Adaptive Network Flow Parameters for Stealthy Botnet Behavior

Machine Learning techniques for providing perturbations to network flow patterns

Torgeir Fladby

Thesis submitted for the degree of
Master in Informatics: Network and System Administration
30 credits

Institutt for informatikk
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2018

Adaptive Network Flow Parameters for Stealthy Botnet Behavior

Machine Learning techniques for providing perturbations to network flow patterns

Torgeir Fladby

Abstract

Machine-learning based Intrusion Detection and Prevention Systems (IDPS) provide significant value to organizations because they can efficiently detect previously unseen variations of known threats, new threats related to known malware or even zero-day malware, unrelated to any other known threats. However, while such systems prove invaluable to security personnel, researchers have observed that data subject to inspection by behavioral analysis can be perturbed in order to evade detection [1].

We investigated the use of adversarial techniques for adapting the communication patterns between botnet malware and control unit in order to evaluate the robustness of an existing Network Behavioral Analysis (NBA) solution. We implemented a packet parser that let us extract and edit certain properties of network flows and automated an approach for conducting a grey-box testing scheme of Stratosphere Linux IPS (Slips). As part of our implementation, we provided several techniques for providing perturbation to network flow parameters, including a *Simultaneous Perturbation Stochastic Approximation* (SPSA) method, which was able to produce sufficiently perturbed network flow patterns while adhering to an underlying objective function.

Our results showed that network flow parameters could indeed be perturbed, which ultimately enabled evasion of intrusion detection based on the detection models that were available for our IDS. Additionally, we demonstrated that it was possible to combine evading detection with stochastic techniques for optimization problems, effectively enabling adaptive network flow behavior.

Acknowledgement

First, I would like to express sincere gratitude to PhD student and supervisor Anis Yazidi for extending his already long work hours whenever I needed support, and to Associate Professor and supervisor Hårek Haugerud, whose commitment throughout this project has been invaluable. Whenever I needed advice on writing, algorithms or relevant research, my supervisors made themselves available. Their support has made me feel like they truly wanted me to succeed, which has been a major motivational factor when working on this project.

Furthermore, I would like to thank Associate Professor Stefano Nichele for triggering my curiosity when it comes to complex systems, machine learning and Data Sciences. Associate Professor Kyrre Begnum also deserves recognition for teaching me how to use and appreciate container technology for building scalable and platform-independent solutions.

Author

Torgeir Fladby

Contents

Glossary

ADNIDS Anomaly Detection based Intrusion Detection System. 13

API Application Programming Interface. 9

APT Advanced Persistent Threat. 8

ASRS Adaptive Step-Size Random Search Algorithm. 19

C2 Command and Control. 9

DDoS Distributed Denial of Service. 9

DNS Domain Name System. 9

GAN Generative Adversarial Network. 23

GO Global Optimization. 18

HIDS Host-Based Intrusion Detection System. 12

HTTP Hypertext Transfer Protocol. 9

IDPS Intrusion Detection and Prevention System. 1

IDPS Intrusion Prevention System. 12

IDS Intrusion Detection System. 8

IoT Internet of Things. 9

IRC Internet Relay Chat. 9

LA Learning Automaton. 21

ML Machine Learning. 10

NBA Network Behavioral Analysis. 1

NIDS Network-Based Intrusion Detection System. 12

P2P Peer-to-Peer. 9

SA Stochastic Approximation. 20

SaaS Software as a Service. 9

SIEM Security Information and Event Management. 12

Slips	Stratosphere Linux IPS. 15
SO	Stochastic Optimization. 18
SPSA	Simultaneous Perturbation Stochastic Approximation. 1
STF	Stratosphere Testing Framework. 16

Chapter 1

1 Introduction

1.1 Motivation

The immense growth of the Internet has over the past three decades laid the foundation for massive technological development in almost all modern industries. However, as digital solutions open up for opportunities for citizens and organizations, the increased accessibility of intellectual property introduces an arsenal of new risks. Advanced persistent threats (APTs) are more frequently targeting critical infrastructure such as government systems, power grids, mobile communication systems, critical manufacturing facilities, water supplies and supply chain systems [2]. As businesses and organizations migrate services and infrastructure to the cloud, and more frequently integrate with third-party solutions, maintaining a responsible overview over data and communications in internal networks becomes increasingly difficult [3]. As a result, companies and nation states are forced to take the risk introduced by increased attack surface, adaptive adversaries and response-driven cyber security into consideration when conducting their daily business.

The last decade has seen a vast increase in the amount of new types of malware; approximately 8.4 million new malware variants were observed in 2017, up from 0.13 million in 2007 [4]. This surge of emerging threats renders traditional signature-based intrusion detection and prevention tools less feasible for protecting digital infrastructure against cyber attacks. Although cyber security specialists and software vendors are disclosing and patching more vulnerabilities than ever, there is a constant struggle to keep up with adversaries who are trivially able to create new types of malware. Once a new type of malware is disclosed, anti-virus software vendors must fetch a signature for it and patch all signature-based Intrusion Detection System (IDS) software installed on their customers' infrastructure. Hence, due to the vast amount of new types of malware, relying on detection of known threats is not effective for organizations whose threat actors are able to continuously adapt.

A key issue related to defending cyber infrastructure is the scale of which new malware can be deployed. Once a new strain of malware is created, the malware author(s) objective is often to infect as many entities as possible, and organize them collectively, in order to gain as much momentum as possible when conducting attacks against target systems. Networks consisting of computers infected by malware with a particular set of network- and host-based properties are known as *botnets*. Such networks are prepared to conduct malicious activities at a large scale in order to abuse or disrupt the operation of its target's services.

Modern botnets typically support a wide range of malicious activities that can be issued on behalf of the infected host. Different malware is typically created for different platforms; while some botnets target personal computers, other might target IoT or Mobile devices [5].

Malware residing on infected hosts typically have the ability to perform a wide range of passive or active attacks, usually as part of a standalone attack against the user of the infected host, or as part of a collective effort to attack other systems. Attacks against users of the specific host can include information gathering, with the goal of acquiring information worthy of protection, or excessive resource consumption, as a means to farm crypto-currencies [2] [4]. Attacks that include the infected host as part of a collective effort can contribute to more large-scale efforts by propagating through established means of communication (phishing through SMS, e-mail and/or chat-services), or by letting the infected device contribute to Distributed Denial of Service (DDoS) attacks [2] [6]. Botnets are typically organized a set of nodes in one of two different network topologies [7] (albeit with sub-topologies): Centralized Command and Control (C2), in which one or more centralized, and possibly tiered, command and control servers are used to control the infected units -, or decentralized C2, where infected units typically organize, communicate and propagate through Peer-to-Peer (P2P) networking protocols.

What most botnets have in common is the requirement of networking capabilities to issue commands from C2 to infected host, or between infected hosts. C2 communication has been implemented on a wide range of networking protocols, including IRC, P2P, DNS, Tor hidden services and HTTP. Moreover, popular Internet services' APIs, such as Twitter, Github, Pastebin, Telegram and Instagram APIs have been abused for managing C2 communication [8] [5] [9]. Because such services encrypt their traffic, and because the use of encrypted traffic in general has increased drastically over the last few years [2], network analysis tools must use intrusion detection techniques that do not rely on evaluating the data that's being transmitted. However, the fact that C2 communication can be implemented on top of such a wide range of networking services could make malicious network traffic generated by infected units hard to separate from benign traffic. Cisco's 2018 Annual Cyber Security Report confirms that malicious users more frequently conceal their malware's communication in channels hosted by legitimate applications, or by closely mimicking such network traffic [2]. Additionally, the increase in use of distributed architectures such as Software as a Service (SaaS) and IoT, in combination with increased use of cloud-based Internet services, gives malicious users a larger attack surface. For the aforementioned reasons, defense communities have started embracing more advanced tools such as machine-learning and artificial intelligence systems that show recent promising results when it comes to detecting malware with unknown characteristics [10].

Because botnets often use complex C2 structures with multiple controlling endpoints, have the potential to perform a large variety of different activities and perform various complex attacks, they are difficult to precisely define. For instance, hosts that are part of a botnet may install and use different layers of services and malware while reporting to a large set of C2 communication channels [11]. The complexity and variable behavior of botnets make signature-based detection and prevention systems that rely on static or dynamic rule sets more prone to false negatives, as they continously lag behind malware that adapts it communicative behavior. In [11], Garcia et. al initiated the Malware Capture Facility Project, with the goal of capturing a vast amount of botnet behavioral patters. After analyzing a variety of such patterns, Garcia et. al identified C2 communication channels as potentially defining characteristic of botnets. By defining the behavior of a botnet through a chain of network flow states based on Markov Chains, they were able to create a "signature" of botnets' behaviors that relied on the malware communication's periodicity rather than traditional identifiers such as content of packet payloads. By comparing the behavior of network flow patterns, Garcia et. al were able to detect other, previously unseen, botnet behaviors [12]. Hence, this type of machine-learning based approach to network behavior analysis is promising because the models generated for a particular type of botnet communication may be used to detect other types of malware with similar behavioral patterns.

According to [2], organizations rely heavily on Machine Learning (ML)-based defense systems; as much as 34% of organizations queried in 2018 were completely reliant on the use of machine learning for defending their digital infrastructure [2]. Such systems provide significant value to organizations because they can efficiently detect a range of known and unknown threats. However, these systems are expensive, somewhat technologically immature and require significant time and resources to implement and maintain [13]. While ML-based systems prove invaluable to security personnel and indeed make it easier to detect abnormal behavior in networks, it has been proven that such systems can be misled into falsely classifying malicious network traffic as benign [14] [15]. This project aims to evaluate the robustness machine-learning based behavioral analysis, and determine wheter adaptive malware can efficiently evade detection.

1.2 Problem Statement

The efficiency of IDS relies on several parameters, including positioning in the network, configuration of preprocessors, event management and more importantly the techniques used for detection. A combination of signature detection, anomaly detection and a variety of ML-based approaches are commonly used in modern IDPS [16]. An example of a system which provides ML-based NBA is Stratosphere Linux IPS (hereby referred to as Slips), an open source implementation of an NBA that uses Markov chains to generate models of malicious network traffic over time. Slips provides IDS functionality by comparing these models with live network traffic and determines whether the recorded network flow behavior matches that of a malicious profile. This project aims to evaluate Slips' behavioral analysis capabilities with respect to its robustness against adaptive network flow behavior. The basic question that we wish to resolve can hence be formulated as the following:

- How can the behavior of malicious network traffic be altered to evade detection by abehavioral analysis tool?

1.3 Thesis Structure

The report is divided into the following chapters:

- **Introduction:** A description of context, problem statement and scope.
- **Background:** A discussion on the theoretical and technological background that this thesis is based upon.
- **Related Work:** A summary of related research that demonstrates the motivation to solve the problem statements.
- **Approach:** A structured overview of implementation strategies and code that contribute to solving the problem statement.
- **Result:** A discussion about expectations of the work and an assessment of the achievements.
- **Discussion:** A brief review of the implementation, approach, challenges and future work.
- **Conclusion:** A summary of the work that was done and contributions made to this project.

2 Background

In order to address the defense mechanisms and attack techniques that are evaluated throughout this thesis, it is important to have an understanding of the technology that these systems are based upon. Consequently, background information and research relevant to IDPS and their underlying detection models will be accounted for. Furthermore, key concepts behind the NBA tool Slips will be explained, and the models Slips uses for detection will be elaborated upon. Additionally, because we want to perform perturbation to a set of parameters, we investigate a select set of optimization techniques that are relevant to such procedures.

2.1 Intrusion Detection and Prevention

Intrusion detection is the process of monitoring network traffic and other events in a network in order to analyze them for indications of malicious or abnormal activity. IDSs are crucial for protecting the security of computer networks because they are responsible for detecting violations, incidents or direct threats to the security policies that determine how information and systems should be accessed. In general, one distinguishes between two different types of IDSs:

- **Host IDS (HIDS)**: Resides on one or more hosts in a network, monitoring dynamic behavior in file systems and processes to detect malicious or abnormal activity.
- **Network IDS (NIDS):** Software that is strategically located on network endpoints such as routers and gateways to monitor network traffic to and from all entities in a network.

There are significant limitations as to what an IDS standalone provides of security because the system is only able to *detect* malicious behavior rather than prevent it. However, the IDS provides highly valuable information to system administrators and software that analyze data and/or provide active response to identified incidents. Software with the ability to analyze, stop and/or block malicious activity are called Security Information Evenent Management Systems (SIEMs). Such systems typically leverage advanced data analysis tools that perform behavioral profiling and classification of network traffic. The intelligence generated by the SIEM can be transmitted to an incident response team or an automated Intrusion Prevention System (IDPS) for active response. The process of both detecting and acting upon violations of security policies is IDPS, while the SIEM conducts security analysis on enterprise-scale networks.

2.1.1 Detection methods and challenges

For an IDS to provide timely and accurate information to an organization's SIEM, it must report and distinguish between a large variety of different attacks in an efficient manner. Because of the large amount and variety of data generated by today's systems, an IDSs challenge is to minimize false positives (benign behavior reported as malicious) and true negatives (undetected malicious behavior). Consequently, the detection methods that are used for identifying threats are of significant importance, and must adapt to the behavior of the systems' adversaries. Generally, one distinguishes between two different high-level types of detection techniques:

- **Signature-based:** Detection of attacks by looking for known patterns (signatures) in static files, kernel behavior, network packets or known malicious instruction sequences in network or process flows. Signature-based detection techniques are typically not applicable to encrypted network traffic and are often prone to true negatives when malicious, but previously unseen malware, is compared to a set of known signatures. However, behavioral network flow analysis does take advantage of some signature-based techniques in that the similarity between time- and size-related parameters of previously recorded traffic plays a more significant role in detection mechanisms than the actual data transmitted. Hence, behavioral IDS can be applied to a network segment despite the communication being encrypted because time-related parameters are not affected by encryption.
- **Anomaly-based:** Detects unknown threats by classifying data flows in the target domain and looking for abnormal activity. Machine learning and behavioral analyses are used in order to create a model of benign behavior - any significant deviation from the modeled behavior is considered anomalous. While some Anomaly Detection based NIDS (ADNIDS) categorize network behavior as either malicious or benign, state-of-the-art techniques attempt to categorize behavior into different classes of attack techniques [17].

To be able to perform behavioral analysis, an ADNIDS must have an idea of what types of network traffic are typical for the segment that it's monitoring. For digital infrastructure whose connected hosts employ a variety of different services, and hence generates a vast amount of different network traffic patterns, an ADNIDS cannot create a single profile for the entire segment and efficiently detect anomalies. Rather, its system administrators must create behavioral profiles for each communication channel that captures the specific service's unique characteristics.

In order for an ADNIDS to scale properly and avoid significant false positive rates, two main challenges typically arise when an efficient and accurate system is to be implemented:

- *Feature selection*, i.e. identifying which subset of features in network traffic are representative to the target domain is hard because it is imperative to the accuracy and efficiency of the model that the significant features are maintained while the irrelevant or redundant features are omitted [18].
- *Correctly labeled and sufficient size of data set* are common challenges, usually because of the vast amount of data needed to be labeled manually. The workload of labeling a data set increases vastly when the feature set increases.

Feature selection is an extremely important element when solving ML-based classification problems. The process of selecting the most relevant features aims to reduce the complexity of models such that they can be interpreted by researchers, reduce the training time and furthermore reduce the chances that a model is overfitted. [19]. Additionally, it is not given that the features selected for one class of attacks fits the patterns of other types of attacks because attack techniques are continuously evolving. For this reason, feature selection techniques can be described as a search technique for proposing new feature subsets. Each subset is typically given a score, which is later taken into consideration when searching for other subsets. Feature selection is however less relevant for this project because we are mostly interested in the timing-aspect of network flows.

2.1.2 Anomaly detection and behavioral analysis

NBA is a technique that can be used to enhance the performance of an ADNIDS by monitoring live traffic and disclosing unusual actions or patterns in periodicity that significantly deviate from normal behavior. NBA's purpose in a cyber defense system is to reduce the amount of resources required by system administrators in resolving network issues and detecting known or unknown threats to the network. Hence, it is an enhancement to already established security infrastructure such as firewalls, antivirus or similar intrusion management software. NBA gives network security analysts the ability to monitor what is is happening in internal or external-facing network connections by aggregating data from a variety of different hosts. Such data is typically analyzed and evaluated using three different categories of rules:

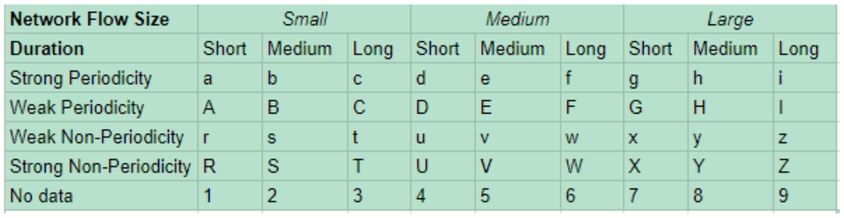
- **Anomaly Rules:** Tests events and network traffic flows for changes in short-term events compared against a longer time-frame. Typically reacts to changes in network services or infrastructure.

- **Threshold Rules:** Tests events and network traffic flows for activity that exceeds or are below specified sets of thresholds.
- **Behavioral Rules:** Detects rate or volume changes that occur in network traffic flows over predefined seasons. A season is defined as the benchmark time consumption for comparison with the traffic that is being evaluated. The length of seasons can be automatically re-evaluated to better fit the model of benign behavior.

In order to determine what portions of network flow is deviating from normal behavior, an NBA must perform an offline comparison of live data where the benign or malicious benchmark is evaluated with respect to the new data. Based on seasonal parameters, offline analysis of network flows may vary significantly in time consumption because malware may distribute its activity over longer periods of time in an effort to evade detection.

2.2 Stratosphere Linux IPS

Stratosphere Linux IPS (Slips) is an open-source network flow analysis tool, developed by Czech researchers Sebastien Garcia et. al [12]. The tool takes as input network flows from a *ra client* and feed them to a set of network behavioral models and detection algorithms. More specifically, the Slips program models network flows as Markov chains by consuming parameters such as size, duration and periodicity [20] of each flow. Based on the chain of states generated, each flow is assigned a letter and a symbol that characterizes the behavior of the connection in that specific state. Periodicity, duration, size and time differences character representations are as listed in Fig. 1 [21]. Character representations of *duration between network flows* are listed in Figure 2.

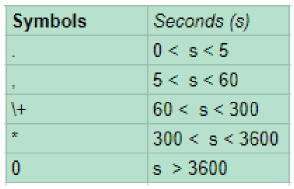


Network Flow Size	*Small*			*Medium*			*Large*		
Duration	Short	Medium	Long	Short	Medium	Long	Short	Medium	Long
Strong Periodicity	a	b	c	d	e	f	g	h	i
Weak Periodicity	A	B	C	D	E	F	G	H	I
Weak Non-Periodicity	r	s	t	u	v	w	x	y	z
Strong Non-Periodicity	R	S	T	U	V	W	X	Y	Z
No data	1	2	3	4	5	6	7	8	9

Figure 1: Character representation of network flow size in bytes and duration of network flow

2.2.1 Behavioral Models and Detection Methods

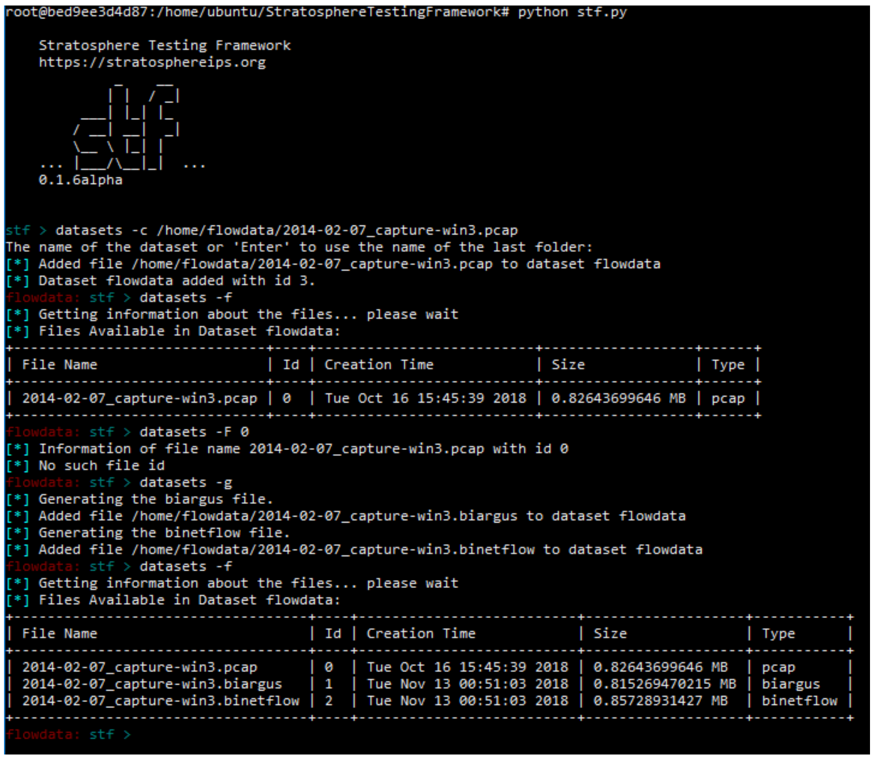
Slips uses a set of symbols to represent the time between network flows (Fig. 2). By concatenating network flows represented as characters and symbols,



Symbols	*Seconds (s)*
.	0 < s < 5
,	5 < s < 60
\+	60 < s < 300
*	300 < s < 3600
0	s > 3600

Figure 2: Symbol representations of time between network flows.

network administrators can gain insight into their network traffic by looking at how periodic different network flows are. In the below example, we use Stratosphere Testing Framework (STF) to generate a network flow file from a packet capture file, in order to further analyze it using Slips. Figure 3 shows how STF can utilize a packet-capture file to generate network flow and argus files. Network flow data is extracted from the network flow files in order to generate parameters for the *models* functionality, which generates a Markov Chain for each network flow in the packet capture. The Markov Chain can be used by Slips to determine similarities between the input pattern and malicious patterns.



```
root@bed9ee3d4d87:/home/ubuntu/StratosphereTestingFramework# python stf.py

    Stratosphere Testing Framework
    https://stratosphereips.org
             _    __
            | |  / _|
         ___| |_| |_
        / __| __|  _|
        \__ \ |_| |
    ... |___/\__|_|  ...
    0.1.6alpha


stf > datasets -c /home/flowdata/2014-02-07_capture-win3.pcap
The name of the dataset or 'Enter' to use the name of the last folder:
[*] Added file /home/flowdata/2014-02-07_capture-win3.pcap to dataset flowdata
[*] Dataset flowdata added with id 3.
flowdata: stf > datasets -f
[*] Getting information about the files... please wait
[*] Files Available in Dataset flowdata:
+------------------------------+----+--------------------------+------------------+------+
| File Name                    | Id | Creation Time            | Size             | Type |
+------------------------------+----+--------------------------+------------------+------+
| 2014-02-07_capture-win3.pcap | 0  | Tue Oct 16 15:45:39 2018 | 0.82643699646 MB | pcap |
+------------------------------+----+--------------------------+------------------+------+
flowdata: stf > datasets -F 0
[*] Information of file name 2014-02-07_capture-win3.pcap with id 0
[*] No such file id
flowdata: stf > datasets -g
[*] Generating the biargus file.
[*] Added file /home/flowdata/2014-02-07_capture-win3.biargus to dataset flowdata
[*] Generating the binetflow file.
[*] Added file /home/flowdata/2014-02-07_capture-win3.binetflow to dataset flowdata
flowdata: stf > datasets -f
[*] Getting information about the files... please wait
[*] Files Available in Dataset flowdata:
+-----------------------------------+----+--------------------------+-------------------+-----------+
| File Name                         | Id | Creation Time            | Size              | Type      |
+-----------------------------------+----+--------------------------+-------------------+-----------+
| 2014-02-07_capture-win3.pcap      | 0  | Tue Oct 16 15:45:39 2018 | 0.82643699646 MB  | pcap      |
| 2014-02-07_capture-win3.biargus   | 1  | Tue Nov 13 00:51:03 2018 | 0.815269470215 MB | biargus   |
| 2014-02-07_capture-win3.binetflow | 2  | Tue Nov 13 00:51:03 2018 | 0.85728931427 MB  | binetflow |
+-----------------------------------+----+--------------------------+-------------------+-----------+
flowdata: stf >
```

Figure 3: Generating datasets using stf

The packet capture used in Fig. 3 (*2014-02-07_capture-win3.pcap*) [22] consists of 4 hours of Zeus malware communication between an infected host and a C2 server. The connection flow of the malicious connection is evaluated as highly periodic with small to medium network flow sizes and short durations between network flows. The sequence of network flows generated by the malware is represented by STF in the following chain of characterizations:

```
44.R+R.U.u.a.a.d.a.d.a.a.d.d.a.d.a.a.a.d.a.a.a.a.a.a.a.d.
   d.a.a.a.a.a.a.a.a.a.a.d.d.a.a.d.a.d.a.a.d.d.a.a.a.a.a.
   d.a.a.a.a.d.a.a.d.d.a.a.d.a.d.a.a.a.a.a.a.a.a.a.d.a.a.
   a.a.d.a.a.d.a.a.a.a.a.d.a.a.a.d.a.d.a.a.a.a.a.d.a.a.d.
   d.a.a.d.a.a.d.a.d.a  (...)
```

When Slips has decided whether the input network flows represent benign or malicious traffic, it offers active response simply by adding an iptables firewall rule. Because this is the only feature of Slips that provides intrusion prevention, and because time-consuming offline processing is required in order to issue any countermeasures, one could argue that Slips by itself does not provide functional intrusion prevention but rather acts as a complement to additional security measures.

Slips' detection algorithms use previously created models of known malicious behavior to detect new suspicious connections in the network. First, models for ground-truth network traffic are generated; these models form a baseline for both normal and malicious behavior and are used as the benchmark for comparison. When Slips captures network traffic, the network flows and their characterizations are calculated and then compared to the known and verified ground-truth models. If the input network flows match an existing model by a given threshold, they are reported as malicious by the detection models and an alert is raised [23].

The *network flow periodicity* implied by a given chain of states aids Slips in identifying similarities between flow patterns, even if the size and duration of network flows varies. The calculation of periodicity is a key aspect of Slips' detection mechanism in that it is able to extract a piece of meta-data from malicious behavior that malware authors may not be aware of. Given three network connections with time stamps t_1, t_2, t_3, in a particular network flow, the flow's periodicity is defined as the difference between $t_3 - t_2$ and $t_2 - t_1$ [11]. Hence, if an attacker can continuously and sufficiently vary the time between connections, perturbation that affects the the periodicity evaluation of each network flow may affect its representation as a chain of states.

2.3 Stochastic Optimization for Parameter Perturbation

In order to provide disturbance to network flow patterns, original patterns must be observed, measured and provided as input to a function that returns a perturbed version of that data. Stochastic Optimization (SO) is a class of statistical methods that can be used to generate random variables based on initial values. In this project, such methods can be employed as a means to provide perturbation for the network flow parameters *flow size, duration* and *time until next flow.* SO methods aim to minimize or maximize an objective function when there is a degree of randomness in input data. One could, for instance, define an objective function as the problem of maximizing throughput in a system, and use SO to provide parameters that solve that particular optimization problem. Given a valid objective function of the aforementioned sort, we may use such a technique to provide optimal perturbation to continuous network flow parameters.

SO has the advantage of being able to work with continuous data rather than discrete. This is a suitable property when working with time units if we interpret network flow duration measurements as continuous data. Since time can be measured down to fractions of a second, we assume that we are working with random continuous and *generative* input values. This subsection elaborates on a select set of optimization strategies that have potential for use within our problem domain.

2.3.1 Random Search

Random search is a simple direct SO search method that samples solutions from the entire search space based on input variables from a uniform probability distribution (i.e. equal probability of selecting any element in the distribution). This Global Optimization (GO) search method does not require derivatives to navigate the search space. Because Random Search selects variables from a uniform distribution, the current sample is independent of those that have been tried before. Random Search *could* provide a reasonable approximation of the optimal solution, but does not scale well when the amount of dimensions increases. The random search algorithm trials a randomly selected candidate solution and adopts it as the best if it improves the result of the objective function [24]:

Algorithm 1 Random Search Algorithm

```
1: procedure RANDOM SEARCH
2:     Arguments : NumIterations, ProblemSize, SearchSpace
3:     Output : Best
4:     Init :
5:     $Best \leftarrow \emptyset$
6:     for $iter_i \in NumIterations$ do
7:         $candidate_i \leftarrow RandomSolution(ProblemSize, SearchSpace)$
8:         if $Cost(candidate_i) < Cost(Best)$ then
9:             $Best \leftarrow candidate_i$
10:        end if
11:    end for
12:    return $Best$
13: end procedure
```

2.3.2 Adaptive Random Search

Adaptive Random Search, or Adaptive Step-Size Random Search (ASRS) is an extension of the Random Search Algorithm designed to reduce the negative impact of random step sizes and address the limitations of fixed step-sizes. The key difference between Random search and ASRS is that ASRS trials a large step-size and adopts the larger step-size if it yields a better result. This approach to optimization both improves the performance of the algorithm and enables evasion of local optima because smaller step-sizes are adopted if no improvement is made after a pre-defined amount of time or iterations [25]. A description of this algorithm can be read as pseudocode in Algorithm 8.

2.3.3 Stochastic Hill Climbing

In contrast to ASRS' Global Optimization approach, the Stochastic Hill Climbing Optimization Algorithm is a type of Local Optimization algorithm that visits a random neighbour and then determines whether the neighbour configuration resulted in an improvement of the objective function or not. For this algorithm to be used in a continuous domain, a step-size must be defined such that a candidate's neighbours can be identified. This algorithm converges, or reaches a local optima, when no neighbour solutions offer improvement or equal cost to the current solution. Because this algorithm should be repeated after convergence to ensure that it has not found a local optima, it is less suited for computations that require many feasible results because it frequently generates overhead trials. The algorithm is quite similar to that of Random Search, except it visits neighbours based on step-size instead of just a random vector and defines its current vector as the best solution when the maximum number of iterations is reached:

Algorithm 2 Stochastic Hill Climbing

```
1: procedure STOCHASTIC HILL CLIMBING
2:     Arguments : $iter_{max}, ProblemSize, StepSize$
3:     Output : Current
4:     Init : $Current \leftarrow RandomSolution(ProblemSize, StepSize)$
5:     for $iter_i \in iter_{max}$ do
6:         $candidate \leftarrow RandomNeighbour(ProblemSize, StepSize)$
7:         if $Cost(candidate) \geq Cost(Current))$ then
8:             $Current \leftarrow candidate$
9:         end if
10:    end for
11:    return $Current$
12: end procedure
```

This algorithm could be useful for improving the result of candidate solutions generated by other algorithms.

2.3.4 Stochastic Approximation

Stochastic Approximation (SA) is a well-known procedure for finding roots of equations when there are noisy measurements. SA uses noisy observations to find the set θ of a function g where $g(\theta) = 0$ [26]. The function g in this case is however not the objective function, but rather the gradient of the *expected* objective function f [27]:

$$g(\theta) = \nabla_\theta f(\theta) \tag{1}$$

When $g(\theta) \equiv \nabla_\theta f(\theta)$, the SA method finds the local optima of f for an unconstrained decision set. The decision set of gradients is determined by trialing noisy subgradient observations on g such that $f(\theta) = g(\theta_i, \zeta_i)$, where ζ is noise. The next decision, θ_{i+1} is then updated by moving a distance a_i in the direction opposite the gradient:

$$\theta_{i+1} = \theta_i - g_i(\theta_i) \times a_i \tag{2}$$

Since it is not given that network traffic has constructive generation, θ_i cannot be observational data. The stochastic approximation function must then use estimates of the gradient. This can be done by using the current decision, θ_t as a starting point and adding a perturbation $e_j c_i$, where e_j is a vector with all 0's except for a 1 in the j_{th} spot and c_i is an iterator of perturbation sizes, or gain sequences, that decreases toward a value between 0 and 1. In [28], Spall et. Al generate an estimate of f by adding and subtracting a random perturbation vector to get an estimate with two observations. The j_{th} entry of the gradient is then approximated by $g(\theta_i, \zeta_i)$ [29], as

$$[g(\theta_i, \zeta_i)]_j = \frac{F(\theta_i + c_i e_j, \zeta_{i,1}) - F(\theta_i - c_i e_j, \zeta_{i,2})}{2c_i}, \tag{3}$$

, where $\zeta_{i,1}$and $\zeta_{k,2}$ is noise for each θ_k measurement. If this approach is to be used for generating perturbations to network parameters, one would have to replace the gradient ∇_θ with a random perturbation vector and furthermore determine the ideal values for gain sequences. Gain sequences are vital to the performance of SA algorithms and must be determined in a case-to-case scenario [28]. To solve this problem, one could run many trials and observe which gain sequences perform well in the problem domain.

2.3.5 Learning Automaton

As a means to provide optimal configuration for an optimization problem, one could employ a simple machine-learning based technique: A Learning Automaton (LA) machine learning algorithm could be used to determine an optimal combination of actions based on an initial set of allowable actions. The LA could iteratively select inputs to the target environment, which after using those parameters would return a stochastic response and a performance score. In the case of parameter perturbation for evading intrusion detection, the stochastic response could be whether the perturbed connections were detected as malicious or not. The performance score could be similar to the objective function used in the perturbation algorithm. If the stochastic response from the target environment was positive, the LA would reward that solution by increasing the probability, $\hat{p}$, that those parameters are again selected. If the stochastic response is negative, the probability vector $\hat{p}$ is left untouched and hence penalized as other solutions gain positive feedback [30]. This algorithm for updating the probability vector would then implement a reinforcement-based learning scheme because it enables recurrence for good solutions. In [30], this was implemented for an LA using the reward-inaction algorithm, which is a simplistic but effective approach in many scenarios. An LA that optimizes two parameters for a variable environment can be formulated as the pseudo-code in Algorithm 3.

Algorithm 3 Learning Automaton

```
1: procedure Pursuit Learning Automaton
2:     Arguments : $iter_{max}$
3:     Output : $best\_index\_a, best\_index\_b$
4:     Init :
5:     $\lambda \leftarrow 0.1$
6:     $N_a \leftarrow 100$ Number of values in probability vector A
7:     $N_b \leftarrow 100$ Number of values in probability vector B
8:     $a_{min} \leftarrow$ Minimum value for a
9:     $a_{max} \leftarrow$ Maximum value for a
10:    $b_{min} \leftarrow$ Minimum value for b
11:    $b_{max} \leftarrow$ Maximum value for b
12:    $A \leftarrow \{a_{min} \times i + 1.0 \times (a_{max} - a_{min})/N_a \text{ for } i \in \{0, ..., N_a\}\}$
13:    $B \leftarrow \{b_{min} \times i + 1.0 \times (b_{max} - b_{min})/N_b \text{ for } i \in \{0, ..., N_b\}\}$
14:    $P_A \leftarrow \{0.5 \text{ for } i \in \{0, ..., N_a\}\}$
15:    $P_B \leftarrow \{0.5 \text{ for } i \in \{0, ..., N_b\}\}$
16:    $B \leftarrow 0$ Value of best performance so far
17:    $I_a \leftarrow 0$ best index for $P_A$ so far
18:    $I_b \leftarrow 0$ best index for $P_B$ so far
19:    $E \leftarrow$ target environment
20:    $\alpha \leftarrow$ action for Environment E
21:    $\gamma(weights) \leftarrow$ roulette selection function
22:    for $iter \in iter_{max}$ do
23:        $index_a = \gamma(P_A)$                                  ▷ Roulette selection
24:        $index_b = \gamma(P_B)$
25:        $improvement \leftarrow False$
26:        $feasible \leftarrow True$
27:        $reward, feasible \leftarrow E(a(index_a, index_b))$
28:        if $reward > B \wedge feasible$ then    ▷ $\alpha_{iter}(index_a, index_b)$ was better
29:            $B \leftarrow reward$
30:            $improvement \leftarrow True$
31:            $I_a \leftarrow index_a$
32:            $I_b \leftarrow index_b$
33:        end if
34:        for $i \in \{0, ..., N_a\}$ do                          ▷ Update weights for $P_a$
35:            if $i \equiv I_a$ then
36:                $P_a^i = P_a^i + \lambda \times (1 - P_a^i)$ ▷ Increase probability of selecting $P_a^i$
37:            else
38:                $P_a^i = P_a^i + \lambda \times (0 - P_a^i)$  ▷ Decrease other indices accordingly
39:            end if
40:        end for
41:        for $i \in \{0, ..., N_b\}$ do                          ▷ Update weights for $P_b$
42:            if $i \equiv I_b$ then
43:                $P_b^i = P_b^i + \lambda \times (1 - P_b^i)$
44:            else
45:                $P_b^i = P_b^i + \lambda \times (0 - P_b^i)$
46:            end if
47:        end for
48:    end for
49:    return $I_a, I_b$
50: end procedure
```

3 Related Work

Techniques using machine learning to detect previously unseen malware have been studied extensively throughout the last century; however, it is only in recent years that adversarial machine learning as a tool for evading such systems has gained significant attention. While many researchers attempt to improve the performance and accuracy of their classification algorithms [31], evaluation of machine learning based intrusion detection *robustness* has been studied far less. Kos, Fischer and Song, et. al proved in [32] that small, but carefully crafted, perturbations to original input images can mislead a neural network classifier to produce incorrect output. Kos et. al used adversarial examples to attack generative models, and were able to mislead neural networks trained on MNIST [33], SVHN [34] and CelebA [35] datasets with high confidence by feeding the target models with adversarial examples. The researchers showed how to break the integrity of an ML-based image classifier, but Kos et. al more importantly demonstrated how a generative adversarial model could be used to attack neural networks trained on a range of different datasets.

The advancement of adversarial techniques that use machine learning to perturb input features has seen development in a range of other domains. Rigaki et. al demonstrate the use of such techniques against the IPS domain by showing that it is possible to use Generative Adversarial Networks (GANs) to mimic network traffic, adapt malware communication and ultimately avoid detection; in their case eliminating blockage of C2 network traffic. Papernot et. al [36] in 2016 showed how adversarial sample attacks against malware classifiers could be constructed, and furthermore evaluated how defensive mechanisms could be improved by training malware classifiers using data gathered from adversarial training. In 2017, Papernot et. al expanded on their research by staging an attack against a malware classifier that ultimately reached a misclassification rate of up to 63%. There has also been research on measures to make systems robust in order to counter adversarial machine learning attacks, both against rule-based IDSs [37] [38] and against deep learning models [36] [39] [40]. In [41], Yuan et. al investigate and summarize approaches for generating adversarial examples, applications for adversarial examples and corresponding countermeasures. This section aims to elaborate on ML-based techniques for adversarial examples, robustness improvement and adaptive malware to provide a basis for the simpler methods used in this project.

3.1 Adversarial Examples

The goal of an attacker attempting to break a classification algorithm is to either force arbitrarty misclassification or to achieve *source-target* misclassification. While the former attempts to achieve *any* misclassification, regardless of category, the latter attempts to map a set of adversarial input features to a specific target class, ultimately breaking the target IDS' *integrity.* In [41], Yuan et. al survey a broad spectre of techniques exploiting deep learning models' vulnerabilities to adversarial examples. Although the aforementioned paper is mainly focused on image classification, it highlights crucial prerequisites and components, and furthermore create a taxonomy for constructing and defending against adversarial examples based on these. Yuan et. al describe the generation of an adversarial example X based on a benign sample x against a deep learning model f as a box-constrained optimization problem:

$$\min_{X} X - xs.t.f(X) = L, f(x) = l, l \neq L, X \in [0, 1] \tag{4}$$

, where l and L are the output labels of x and X, respectively. The objective of this optimization problem is to find the minimal perturbation of sample x (denoted $\|X - x\|$) such that the input X is misclassified by the deep learning model [41].

The model described by Yuan et. al in Fig. 1 is very relevant to this project because it shows that a model used for generating adversarial examples can be applied to a wide range of problems, simply by changing the objective function. Because the objective function can be a minimization or maximization problem, we may create adversarial examples in the network flow domain by perturbing parameters based on objectives such as detection rate, throughput or magnitude of perturbation.

3.2 Taxonomy for adversarial examples

As previosly mentioned, Yuan et. al account for a taxonomy that they use to analyze approaches for adversarial examples, which could also prove useful in the context of this project. The taxonomy is based on three primary areas of interest; threat model, perturbation to input parameters and benchmark adversarial performance. This section will elaborate on the taxonomy by pulling information from a larger variety of sources.

3.2.1 Threat model

Yuan et. al describes *adversarial falsification* as the desired result of feeding an adversarial example to the target model. The two main categories are *false positive* attacks; attacks that generate a negative (benign) sample misclassified as a positive sample; and *false negative* attacks, where a positive (malicious) sample is misclassified as negative and hence not detected by the trained model

[41]. As this projects mainly focuses on evading detection, *false negative* attacks are of particular interest.

Adversary's knowledge speaks to the amount of information an attacker has about the target model, including data, hyper-parameters, architectures, number of layers, activation functions and model weights [41]. White-box attacks have complete knowledge of all the aforementioned parameters; black-box attacks are constructed under the assumption that the adversary has no such knowledge. Malware authors usually have little or no access to the detailed structures and parameters of the machine learning models used by IDS or malware detection systems, and can hence mostly perform black-box attacks in real-life scenarios. When testing the robustness of live machine learning models, it is desirable to use black-box techniques in order to simulate a realistic environment. White-box attacks are useful for testing the fundamentals of which the model is built upon, often in order to test accuracy and performance [42].

Yuan et. al furthermore divide attacks in two different categories of *adversarial specificity* that determine whether the adversarial example's objective is to classify as an arbitrary class or a specific class. While targeted attacks attempt to misguide the victim model into mapping the adversarial example to a specific output category, un-targeted attacks are content with being mapped to any class. Un-targeted attacks have a larger attack space than targeted attacks and are hence computationally easier to execute [41]. Un-targeted attacks are useful when the goal is to evade detection, rather than forcing a specific action from the victim model. Such attacks can be generated through a variety of techniques:

- Running several targeted attacks and choosing the sample that has seen success despite having the least amount of perturbation [41].
- Minimizing the probability of the adversarial example being *correctly* classified [41].
- Maximizing the distance between the probability of the adversarial output prediction vector and the predicted class of the benign input, in order to achieve an adversarial input that is classified as any label except the label of the original benign input [43].

When generating adversarial examples to mimic network traffic, it is not given that minimizing the difference between the original input and the adversarial sample is desireable, as humans will likely not manually inspect the content, nor be able to notice significant difference between adversarial and benign network packets. However, properties related to loss functions and minimal perturbation of parameters are fundamental concepts within a variety of machine learning techniques, and is likely an important aspect when using reinforcement-based learning techniques.

3.2.2 Perturbation

Perturbations to original input is a premise when constructing adversarial examples. Since adversarial examples should be imperceptible to humans or anomaly detection systems, it is imperative that the adversarial examples are as close to the original input as possible, while retaining the property of being misclassified by the target model. Under the taxonomy, Yuan et. al analyze three main aspects of perturbation measurement: perturbation scope, perturbation limitation and perturbation measurements [41]:

- **Scope:** *Individual* and *universal* attacks differ significantly in their attack surface; while the former generates different perturbations for each clean input, the latter generates perturbations for an entire dataset. The former is likely the most relevant for the purpose of this project because perturbations must be based on generative network traffic.
- **Limitations:** *Optimized perturbation* and *constraint perturbation* are two different approaches to setting the goal of the optimization problem (1); while the former aims to minimize perturbations so humans cannot recognize the perturbation, the latter only requires the perturbation to be small enough.
- **Perturbation measurement:** Yuan et. al measure the perturbation of an adversarial example by calculating the Euclidean distance (p-norm) in the p-dimensional L_p space. If x is the magnitude of perturbation by p-norm distance, then:
$$x_p = (\sum_{i=1}^{n} {x_i}^p)^{\frac{1}{p}}$$
, which for p = 0, p = 2 and p = ∞ denotes the maximum absolute column sum, Euclidean distance between adversarial example and original sample, and the maximum change for all features in adversarial examples, respectively [41]. The same formula holds for Wei et. al's definition of Degree of Change, which demonstrates the same properties [43]. The magnitude of a perturbation is important because it can be used as a factor in optimization problems that aim to minimize or maximize this property.

Attack frequency significantly influences the perturbation process when constructing adversarial examples. Perturbations to a sample can be made either once or multiple times. Some computationally heavy algorithms due to their nature optimize their perturbations only once, such as many variants of reinforcement learning algorithms [41], and hence have a less sensitive magnitude of perturbation. Adversarial models that perturb their examples multiple times use a feedback loop to iteratively modify the input features with a higher degree of detail, i.e. a smaller amount of modification to the individual parameters. This multi-step process is typically repeated until the input is successfully misclassified, or a given threshold has been reached [43] [1]. Given a static target model, the formula for adversarial untargeted multi-step perturbation of a benign sample x is denoted as:

$$x^t_{adv} = x^{t-1}_{adv} + \theta R(\frac{\beta h(\overrightarrow{y}_{adv}, y*)}{\beta x^{t-1}_{adv}}), t > 1 \quad (5)$$

, where y* is the arbitrary class C_x and h() is a function that describes the relationship between the prediction vector or *loss function* of the original input or adversarial input in previous iteration. R() is a control function that decides how the adversarial example should be modified, specifically by ensuring that the new values are within the range of the given features' limit values [43]. The multi-step approach will receive less focus throughout this project because of the overhead complexity it introduces [43].

3.2.3 Benchmark and robustness evaluation

Because different machine learning models are based on different datasets and measure performance based on different hyperparameters, it is hard for researchers to establish a general method for benchmarking the performance and robustness of such systems. In order to get reliable results from adversarial machine learning techniques, researchers must train their models on widely used datasets and test their adversarial techniques on well-known models. Benchmark evaluation is a challenge for this project because our approach to parameter perturbation is not based on commonly used techniques for adversarial machine learning.

4 Approach

This chapter provides a high-level overview of the technical implementations that were used to address the problem statements. Choices related to implementation, experiment setup and selection of algorithms are justified to support suggested solutions to the problem statements. As a means to approach the problem statement, two high-level sets of experiments are proposed and described in the sections of this chapter.

4.1 Modifying existing packet captures

In order to test whether perturbation of network flow parameters could assist in evading behavioral analysis intrusion detection, a set of initial experiments were conducted on existing malware capture datasets. It was desirable to see if one could reliably alter Slips' representation of network connections' state by altering the properties of packet capture files that has already been modeled and labeled by the Stratosphere IPS team. Specifically, these experiments aimed to alter properties that affect periodicity in network flows; duration of connections, network flows and time between network flows are parameters that affect this property [44]. The size of network flows also impact the state of network flows and the probability that it matches a model labeled as malicious - hence, this parameter was also included as part of the perturbation scheme. Moreover, the size of network flows could be used as a component in a loss function for an optimization scheme, making it an ideal parameter to include in our approach. It is worth noting that the altered network packets should not be subject to data loss, as that would conceptually strip the malicious capabilities of the captured network packets. If network traffic which has previously been observed, labeled and detected by Slips as malicious can be altered to evade detection, without losing their malicious content, conducting other experiments that perturb network flow properties would be justified.

4.1.1 Objectives

To reach the ultimate objective of confirming whether perturbation to network flow parameters affect the probability that a malicious capture packet file is misclassified as benign, some infrastructure and software had to be implemented:

- (1) A container configuration that installs *Stratosphere Linux IPS*, *argus* and *ra* alongside all relevant dependencies to enable execution of experiments on a platform that is trivial to deploy on any operating system.
- (2) Software that takes as input a packet capture file and a set of *per-*

turbation parameters. All relevant network flows must be identified, and properties such as *duration*, *size of network flow* and *time between current flow and next flow* must be computed. When all network flows and other connections are identified and grouped, the argument perturbation parameters must be applied to the relevant network flows. The program must return the same connections, in the same order, but with timestamps that are modified according to the input perturbation parameters.

- (3) A script that takes as input the packet capture file generated in (2) to create a network flow file compatible with analysis in Slips.
- (4) A module that provides perturbation parameters to the program that changes network flow properties of the selected pcap-files. A set of techniques for computing perturbations are features of this module.
- (5) An implementation of a Learning Automata that enables a large set of iterations to be computed over different configuration options of the algorithm in (4). The goal of the Learning Automata is to find an optimal configuration of parameters provided to the selected algorithm. The configuration could, for instance, minimize the impact that perturbation has to network flow durations, while ensuring that the entire network flow is undetected by Slips. For each iteration, the probability that the Learning Automata selects a feasible solution, must be increased.

4.1.2 Changing packet capture timestamps

Creating an algorithm for changing the timestamps of network flows in a packet capture file according to a set of perturbation parameters required a highly structured approach. First, it was important to establish a set of assumptions that would serve as rules of implementation for this functionality:

- (1) The user that perturbs network parameters using knows which IP addresses are malicious. The user selects these IP addresses as *target_tuples* before parsing the packet capture, such that only the selected IPs are subject to *perturbation* of network flow characteristics. However, packets that do not satisfy the condition of being in target_tuples are indeed important to retain and must be written back to the file unmodified.
- (2) A network flow is defined as a set of packets going from a distinct combination of source port and source IP address to another distinct combination of destination port and destination IP. Incoming packets to the same IP and source port of the IP that initiated the connections, also belongs to that same set of packets, regardless of whether the incoming packet's source port has changed. All packets that share these properties belong in the same flow.

- (3) A network flow can consist of an arbitrary number of packets.
- (4) The *duration* of a network flow is defined as the time between the first packet sent and the last packet sent within a particular network flow.
- (5) The *time between network flows* parameter is defined as the absolute amount of time between the start of a particular network flow, and the start of the next network flow which belongs to the same category of *target_tuples*.
- (6) *Size of network flow* is defined as the sum of the length in bytes of all packets within a particular network flow.
- (7) *Network flow throughput* is defined as the amount of bytes sent through all connections over a particular network flow.
- (8) *Total throughput* is defined as the total amount of bytes sent through all connections in all flows that satisfy the condition of being in *target_tuples*.

Class structures

We began by creating object-oriented data structures around the elementary sets of network connections that occur in a packet capture, and implemented these by creating the classes Packet and Flow. A Packet object consists of a *packet number* that corresponds to the packet's index in the packet capture file, and a Scapy *packet* object, that contains all network packet frames up to - and including - Layer 4, as defined in the OSI model [45]. The entire purpose of the Packet class is to maintain the order of packets. A Flow object consists of a considerably wider range of attributes and methods, some of which are described below:

- **Attributes**: *Source port, source IP, destination port, destination IP*
- **Attributes**: *t1, t2 and td*; timestamp of first packet in flow, timestamp of last packet in flow and the time-delta of t2 and t1.
- **Attribute**: *packets*; a list of Packet objects belonging to this flow.
- **Attribute**: *td_list*; a list representing the time-delta between the packets of this flow.
- **Attribute**: *td_fractions*; a list of float $numbers$ where $0 < number < 1$, which corresponds to each element in td_list. Each float number establishes how much of the *flow duration* each time-delta in this flow retains.
- **Attribute**: *time_until_next_flow*; the amount of seconds until the next flow occurs.

- **Method**: *belongs_in_flow(packet)*; True if packet belongs in this flow.
- **Method**: *add_to_flow(packet)*; adds a Packet object to this flow.
- **Method**: *set_flow()*; Sets the values of t1, t2 and td. Usually called after updating timestamps.
- **Method**: *set_timedeltas()*; fills up the lists td_list and td_fractions. Usually called when a Flow object's packet list is complete.
- **Method**: *perturb_duration(perturbation)*; perturbs the duration of the flow by modifying timestamps according to value provided in argument *perturbation*, and distributes the new timedeltas according to the initial fractions provided in td_fractions. Returns the amount of seconds that were perturbed, a value which can be positive or negative.
- **Method**: *incr_timestamps(pert_td_next)*; increments the timestamp of all packets in the flow. Updates t1, t2 and td. Returns the amount of seconds that were incremented, a value which can be positive or negative.

For this project's python implementation of the Flow class, see Appendix C.

Parsing the packet capture file

Before we could start manipulating timestamps in the given packet capture file, some additional supporting data structure and metrics were necessary. The data structure chosen for storing Flow and Packet objects was Python 2.7's *collections.deque*. The procedure for parsing a packet capture file was implemented as the following sequence of actions: We started by parsing the packet capture file using Scapy's *rdpcap()* method and created a Packet object for each Scapy packet. Next, we identified which packets belonged to Flow objects, and which connections should be considered single Packet objects. For each packet that met the criteria of being part of a Flow object, we searched through our current set of Flows to determine whether this packet met the criteria of belonging to any of the flows. The Packet object was then put into its corresponding Flow object, or added as a new Flow object if it did not belong to any existing flow. If a Packet object did not meet the criteria of being in a Flow, it was simply appended to the deque. When every packet in the packet capture file had been parsed, the time-deltas for each flow were updated, and the set of flows was returned.

Measuring time between network flows

In order to perturb time between the network flows, we had to know the initial time between these flows. We implemented a function that takes as input the target IP address tuples and a deque containing Flow and Packet objects, and set the *time_until_next_flow* attribute to the correct value. When all Flow objects' attributes were updated, the flow was returned. Since we maintained the order of Flow and Packet objects using a deque, we simply incremented the deque index to get the next item. If the next item matched the criteria of being the next flow for this target tuple, as described in Algorithm 4, we set the correct value of *time_until_next_flow* and incremented the queue counter (This saved us for quite a few iterations). We were then able to obtain the correct duration for each network flow time-delta in the current set of Flows and Packet objects. The time between two network flows t_i^{time}andt_{i+1}^{time} that belong to the same subset of target tuples can then be defined as $t_{i+1}^{time} - t_i^{time}$. The functionality of measuring time between network flows was implemented as the method *measure_offline_duration(target_tuple, flows)* in Appendix C. The pseudocode for this procedure can be described as Algorithm 4, with a helper procedure in Algorithm 5.

Algorithm 4 Setting time between flows

```
 1: procedure MEASURE_TIME_BETWEEN_FLOWS
 2:     Arguments:
 3:     $target\_tuples \leftarrow$ The IP addresses subject to measurement
 4:     $malware\_packets \leftarrow$ Deque containing all Flow and Packet objects
 5:     init:
 6:     $flows \leftarrow parse\_pcap\_to\_deque(malware\_packets)$
 7:     $counter = 0$
 8:     for $flow \in flows$ do
 9:         if $isInstance(flow, Flow)$ then                ▷ flow is Flow object
10:             if $flow.src\_ip, flow.dst\_ip \in$ target_tuples then
11:                 $flow.next\_flow, flow.time\_until\_next\_flow =$
    $get\_timedelta\_next\_flow\_in(flows, counter, (flow.src\_ip, flow.dst\_ip), flow)$
12:                 $counter += 1$
13:             end if
14:         end if
15:     end for
16: return $flows$
17: end procedure
```

The helper method *get_timedelta_next_flow_in(flows, counter, tuple, flow)* is described in Algorithm 5:

Algorithm 5 Get timedelta between current for and next flow.

```
1: procedure GET_TIMEDELTA_NEXT_FLOW_IN
2:     Arguments:
3:     $deque \leftarrow$ All Flow and Packet objects
4:     $index \leftarrow$ Index of the flow we are currently processing
5:     $tuple \leftarrow$ The (source IP, destination IP) tuple of interest
6:     $cur\_flow \leftarrow$ The flow object we are currently processing
7:     Init:
8:     $found = False$
9:     $counter = 1$
10:    while $\neg found$ do
11:        if $index + counter >= len(deque)$ then return $None, 0$
12:        end if
13:        $next\_flow = deque[index + counter]$
14:        if $isInstance(next\_flow, Flow)$ then                ▷ next_flow is Flow
15:            if $next\_flow.src\_ip, next\_flow.dst\_ip \in$ target_tuple $\wedge$ $next\_flow \notin cur\_flow.packets \wedge next\_flow.packets[0].packet.time \geq cur\_flow.packets[0].packet.time$ then
16:                $found \leftarrow True$ return $next\_flow, next\_flow.packets[0].time - cur\_flow.packets[0].packet.time$
17:            end if
18:        else if $isinstance(next\_flow, Packet)$ then      ▷ next_flow is Packet
19:            if $next\_flow.src, flow.dst \in$ target_tuple $\wedge$ $next\_flow \notin cur\_flow.packets \wedge next\_flow.packet.time \geq cur\_flow.packets[0].packet.time$ then
20:                $found \leftarrow True$ return $next\_flow, next\_flow.packet.time - cur\_flow.packets[0].packet.time$
21:            end if
22:        end if
23:        $counter += 1$
24:    end while
25:    return $None, 0$
26: end procedure
```

Altering timestamps of network flows

Having access to the *flows* set generated by parsing the packet capture file and measuring distances between flows, we were now able to implement the alteration of Packet objects' timestamps. The pseudocode in Algorithm 5 provides a high-level description of the procedure used to perturb the timestamps of all objects in the *flows* set. The Python implementation of this procedure is located in Appendix C, as the method *redefine_stored_network_flows(target_tuples, evolving_params, iterationNo, durations, flows)*.

The goal of this procedure was to generate a new set of Flow and Packet objects that contained the same packets as the ones located in the original packet capture file. All Packet objects had to be written back to a pcap-file, ordered by timestamp. Each packet in the set of parsed flows could now have their timestamps modified to reflect those given by a set of perturbation parameters. For the implementation of the timestamp modifier, it was detrimental that *all* packets belonging to the same network flow were also modified. The implementation of the Flow class combined with the initial parsing of packets and identification of a flow's next network flow, ensured that such a functionality was possible. We implemented the methods *incr_timestamps()* and *perturb_duration()* for the Flow class to handle perturbation to *td_next_flow* and *flow_duration*, respectively. The incr_timestamps() method, when called on the current flow's *next_flow* attribute, scews the timestamps of all packets in the flow by ingesting an argument which reflects the perturbation. The perturb_duration() method, when called on the current flow object, perturbs the duration of that flow by multiplying the fractions of time-deltas between every packet in the flow and distributes the new durations evenly between all the subjected packets.

Since we wanted to observe whether - and to what extent - perturbation of network flow parameters have an impact on detection rate, we had to collect data about the perturbations that we performed. We chose to collect data at the perturbation stage of this experiment, and updated the source code accordingly. The data points that were subject to collection included *throughput*, *total bytes of perturbed network flows*, *total duration of perturbed network flows*, *total perturbed packet capture delay*, *total increment excluding time between network flows*, *total time between network flows increment*, and finally the *evolving params* used as inputs to the perturbation algorithm. The pseudocode for perturbing network parameters of all flows in a flow set can be seen in Algorithm 6.

Algorithm 6 Changing packet timestamps

```
1: procedure REDEFINE_STORED_NETWORK_FLOWS
2:     Arguments:
3:     $target\_tuples \leftarrow$ IP tuples subject to perturbation
4:     $evolving\_params \leftarrow$ Adaptive parameters fed to perturbation algorithm
5:     $iterationNo \leftarrow$ Iteration parameter from Learning Automata
6:     $flows \leftarrow$ Set of Flow and Packet object subject to perturbation
7:     init:
8:     for $flow \in flows$ do
9:         if not $isInstance(flow, Flow)$ then                ▷ flow is Packet object
10:            $timestamp \leftarrow flow.packet.time$
11:        end if
12:        if $isInstance(flow, Flow)$ then
13:            if $flow.src\_ip, flow.dst\_ip \in$ target_tuples then
14:                $t\_between = flow.time\_until\_next\_flow$
15:                $pre\_perturbation = flow.packets[-1].packet.time$
16:                $flow\_params = [flow.flowsize, flow.duration, t\_between]$
17:                $pert = call\_perturbation\_algorithm(flow\_params, evolving\_params)$
18:
                   $negt = \{\forall elem \| elem \in pert \wedge elem \geq 0\} \neq pert$
19:                while $negt$ do                ▷ Omit negative perturbations
20:                    $pert = run\_spsa([flow.flowsize, flow.duration, t\_between])$
21:                    $negt = \{\forall elem \| elem \in pert \wedge elem \geq 0\} \equiv pert$
22:                end while
23:                $pert\_duration = pert[1]$
24:                $pert\_t\_between = pert[2]$
25:                $next\_t\_between = pert\_t\_between - t\_between$
26:                if $len(flow.packets) > 1$ then        ▷ This flow has a duration.
27:                    if $flow.next\_flow \neq None$ then
28:                        $flow.next\_flow.incr\_timestamps(next\_t\_between)$
29:                    end if
30:                    $flow.perturb\_duration(pert\_duration)$
31:                else        ▷ This flow consists of one Packet has no duration.
32:                    if $flow.next\_flow \neq None$ 0 then
33:                        $flow.next\_flow.incr\_timestamps(next\_t\_between)$
34:                    end if
35:                end if
36:            end if
37:        end if
38:    end forreturn $flows$
39: end procedure
```

4.1.3 Generating network flow files

Slips can read traffic either from a live *ra* process or from a *binetflow* file. If we were to choose the first option when testing our newly generated packet capture files, we would need to replay the newly generated packet capture file to a network interface that *ra* was listening to. In order to get results in a reasonable amount of time, the packet replay would need to be sped up significantly. This would likely need to be integrated with Slips' interpretation and parsing of network flow records, which could be a cumbersome task. By choosing a solution where we replay packets, the experiments we wished to conduct would likely suffer significant delays, making the approach unfeasible with respect to the time we had available. For that reason, we chose to generate binetflow files from the packet capture files generated in *change_pcap_timestamps.py*. With Argus and Ra clients configured correctly, and with *ra.conf* available in our Experiments directory, we used the Python module stf.core.Dataset to genereate a STF dataset, a *biargus* file and finally a *binetflow* file. The implementation of this functionality is included in Appendix C, located in the method *redefine_stored_network_flows.py(target_tuples, evolving_params, iterationNo, durations, flows).*

4.1.4 Procedures for parameter perturbation

After the functionality for perturbing network flow parameters was implemented, we wanted to evaluate to what extent the manipulation of these parameters affected the detection rate of malware. Since Slips computes behavioral patterns for network connections based on network flow size, network flow duration and *periodicity* [44], we wanted to cause some manual disturbance of parameters that affect these properties. If manual perturbations affected Slips' detection rate on a given malware capture, we wanted to repeat the experiment using more scientific approaches where the goal was to minimize the perturbations' negative impact on network connection delays. Finally, if we to some extent were able to minimize the impact on connection delays, we wanted to adopt a type of machine learning algorithm that optimize the parameters used with the given algorithms.

An important aspect when generating random perturbations, and in particular for problems that utilize Stochastic Optimization, is the *objective function.* Since we mainly focused on perturbing the periodicity of network flows, and therefore did not perturb the size of network flows, perturbation to that variable does not really impact our resulting packet capture in any way. However, we could still use this variable in our objective function, whether the goal was to reduce the amount of time used by malicious connections or, to make it as similar to the original traffic as possible. Moreover, we can use the information gained by the objective function as feedback to other optimization problems, for instance as a means to generate parameters for a perturbation algorithm. If we want to increase throughput, we may define our objective function as $\hat{y}$,

$$\hat{y}_i = \underset{D}{\operatorname{argmin}} \frac{flow_i^{size}}{flow_i^D}, \tag{6}$$

where D is the product of the duration of the i_{th} network flow and the time until its next network flow.

In scenarios where it is desirable to minimize the difference in throughput, the objective function may for instance minimize the euclidean distance between the perturbed sample and the original input. The objective function $\hat{y}$ could then be denoted as the 2-norm distance between the two samples x and y, where x is the observed data and y is its perturbed version:

$$\hat{y} = (\sum_{i=1}^{n} \|x_i - y_i\|^2)^{1/2}. \tag{7}$$

Depending on what problem is to be solved, however, it is often hard to obtain a direct gradient of the objective function, particularly if one has continuous or generative data. In such cases it could be a good idea to rely on a technique that does not require measurements of the direct gradient of the objective function.

Randomly generated threshold-based perturbations

This experiment was intended as an initial assessment of the effect that random perturbation had to Slips' detection and classification mechanisms. We created an object of PerturbationOptimizer (see Appendix D) and called the method *random_vector_with_threshold(thresholds)* to receive our perturbation parameters. The generation of a random perturbation vector is described in algorithm 7.

Algorithm 7 Getting threshold-based random perturbation parameters

```
1:  procedure RANDOM_VECTOR_WITH_THRESHOLDS
2:      Argument:
3:      $thresholds \leftarrow$ 2D list of bounds for each parameter
4:      init:
5:      $size_{lower}, size_{upper} \leftarrow thresholds[0][0], thresholds[0][1]$
6:      $duration_{lower}, duration_{upper} \leftarrow thresholds[1][0], thresholds[1][1]$
7:      $t\_between_{lower}, t\_between_{upper} \leftarrow thresholds[2][0], thresholds[2][1]$
8:      $vector = [rnd(size_{lower}, size_{upper}),$
9:      $rnd(duration_{lower}, duration_{upper}),$
10:     $rnd(t\_between_{lower}, t\_between_{upper})]$
11:     return vector
12: end procedure
```

By performing random perturbations to the network flow parameters, we could gain insight into how Slips reacted when the time-aspect of malicious traffic was altered. We investigated whether the newly generated packet capture file was detected as malicious by Slips, and to what extent the new parameters affected the periodicity of network flows.

Adaptive Step-Size Random Search Algorithm

The ASRS Algorithm [25] was implemented to take as input a set of boundaries that defined the maximum amount of distance each step could take. Additional inputs were perturbation factors for each parameter subject to perturbation, as well as one factor for small step sizes and one factor for large step sizes. The ASRS algorithm was implemented to trial a large step size for each iteration and adopt the larger step size if it yielded a better result. The ASRS algorithm is described using pseudo-code in Algorithm 8.

Algorithm 8 Adaptive Step-Size Random Search Algorithm

```
procedure ADAPTIVE_RANDOM_SEARCH
    Arguments :
    $iter_{max} \leftarrow$ Max amount of iterations for one session
    $bounds \leftarrow$ 2D list representing bounds, i.e. problem size
    $init\_factor_{bytes} \leftarrow$ initialization factor for the $bytes$ parameter
    $init\_factor_{duration} \leftarrow$ initialization factor for the $duration$ parameter
    $init\_factor_{between} \leftarrow$ initialization factor for the $t\_between$ parameter
    $pert_{small} \leftarrow$ perturbation factor for small step size
    $pert_{large} \leftarrow$ perturbation factor for large step size
    $iter_{mult} \leftarrow$ how often step_size is multiplied by $pert_{large}$
    $max\_no\_impr \leftarrow$ Max amount of unsuccessful attempts before reducing step size
    init:
    $step\_size = list()$
    $step\_size.append((bounds[0][1] - bounds[0][0]) * init\_factor_{bytes})$
    $step\_size.append((bounds[1][1] - bounds[1][0]) * init\_factor_{duration})$
    $step\_size.append((bounds[2][1] - bounds[2][0]) * init\_factor_{between})$
    $current, count = \{\}, 0$
    $current[''vector''] = random\_vector(bounds)$
    $current[''cost''] = objective\_function(current[''vector''], step\_size)$
    while $iter \leq iter_{max}$ do
        $big\_stepsize = large\_step\_size(iter, step\_size, s\_factor, l\_factor, iter_{mult})$
        $step, big\_step = take\_steps(bounds, current, step\_size, big\_stepsize)$
        $good\_cost\_small\_step = step[''cost''] <= current[''cost'']$
        $good\_cost\_big\_step = big\_step[''cost''] <= current[''cost'']$
        if $good\_cost\_small\_step \vee good\_cost\_big\_step$ then
            if $big\_step[''cost''] \leq step[''cost'']$ then
                $step\_size, current = big\_stepsize, big\_step$
            else
                $current = step$
            end if
            $count = 0$
        else
            $count+ = 1$
            if $count \geq max\_no\_impr$ then
                $step\_size = [x/s\_factorforxinstep\_size]$
            end if
        end if
        $iter+ = 1$
    end while
    return $current$
end procedure
```

This algorithm was implemented according to the description in [25] and modified to include the perturbation of three parameters. The implementation was written in Python 2.7 and added as a function to the class PerturbationOptimizer (see Appendix D).

Simultaneous Perturbation Stochastic Approximation

Simultaneous Perturbation Stochastic Approximation (SPSA) is an algorithm used to solve challenging optimization problems where it is difficult or impossible to obtain a gradient of the objective function [28]. In our case, it was not possible to know the gradient of a certain configuration because we did not know whether the flow would be classified as malicious or not at the time of perturbation. SPSA, in contrast to optimization algorithms that work with discrete noise-free data, relies on two separate measurements of the objective function to obtain an approximation of the gradient, regardless of the number of parameters being optimized. This "two-step", *simultaneous* approach is desirable for solving our optimization problem because the data we attempt to provide perturbations for, while trying to reach a particular objective for each iteration, is both continuous and noisy. SPSA is applicable to a variety of problems within engineering and social sciences, but must be adapted to fit each use case. We chose to implement SPSA because of its versatility with respect to different optimization problems and objective functions, such that we may extend functionality at a later point if desirable.

The goal of SPSA in our case is to minimize the loss function $\hat{y}(\theta)$, where the loss function is a scalar-valued measurement of performance, and *theta* is the vector of parameters to be perturbed. SPSA starts by iterating from an initial guess of *theta*, a vector which in our case is based on measurements of a network flow. We can then obtain measurements $\bar{y}(\theta)$ of the loss function $\hat{y}(\theta)$ by adding noise to the initial loss function:

$$\bar{y}(\theta) = \hat{y}(\theta) + \zeta, \tag{8}$$

where ζ is the added noise. When we have exact measurements, however, adding noise is not necessarily desirable. We implemented support for two different objective functions, namely *throughput maximization* and *Euclidean Distance minimization* and slightly increased the noisiness of the value by adding a random variable chosen from a gaussian distribution of 0 ± 0.1 to θ. The SPSA algorithm was implemented by completing the following steps:

We selected initial values for α, γ, c, a, n and the vector β as 0.602, 0.101, 1, 1, 1000 and [0.1, 0.9], respectively. Then, we defined two iterators, or *gain sequences*, as

$$a_i = \frac{a}{(\beta[1] \times (i+1))^{\alpha}}, \text{ for } i \in \{0, ..., n\} \tag{9}$$

and

$$c_i = \frac{c}{(\beta[0] \times (i+1) * 0.5)^{\gamma}}, \text{ for } i \in \{0, ..., n\}. \tag{10}$$

The gain sequences were created so that they would not produce values of $\hat{\theta}$ with excessively large magnitude of perturbation.

Furthermore, we generated an initial *Simultaneous Perturbation Vector* δ_0 by selecting one Bernoulli-value for each parameter subject to perturbation, such that $\delta_0 = [\pm 1, \pm 1, \pm 1]$. Bernoulli-values were selected due to the requirements stated in [28] Section III. A. We then started iterating over SPSA with n iterations. For each iteration k, we computed the following [28]:

- **Two objective function evaluations**: Two measurements of the loss function $\hat{y}(\theta)$ based on the simultaneous perturbation around $\hat{\theta_k}$. The two perturbations were calculated as $\hat{\theta}_{k1} = y(\hat{\theta}_k + c_k \times \delta_k)$ and $\hat{\theta}_{k2} = y(\hat{\theta}_k - c_k \times \delta_k)$, where c_k is the gain sequence as defined in Formula 9.
- **Simultaneous Perturbation Gradient Approximation**: An approximation to the unknown gradient $g(\hat{\theta_k})$ as

$$g_k(\hat{\theta_k}) = \frac{\hat{\theta}_{k1} - \hat{\theta}_{k2}}{2 \times c_k} \tag{11}$$

- **Estimate update**: The next estimate of $\hat{\theta}$, denoted as $\hat{\theta}_{k+1}$ was then computed as the difference between $\hat{\theta_k}$ and $g_k(\hat{\theta_k}) \times a_k$, where a_k is the k_{th} element of our gain sequence a:

$$\hat{\theta}_{k+1} = \hat{\theta_k} - g_k(\hat{\theta_k}) \times a_k \tag{12}$$

- **Termination**: SPSA terminates if there is little change over several successive iterates or the maximum number of iterations has been reached. If these conditions are not met, k is incremented and a new iteration is initiated.

The implementation of the SPSA algorithm was based on the GitHub Gist user *yantan16*'s implementation of SPSA using python iterators [46]. The solution was first modified to meet the needs of this project and then integrated with the python module perturbation_optimizer.py, as seen in Appendix D. Guidelines for selection of gain sequences in [28] were used in an attempt to select viable parameters for our implementation of the SPSA algorithm.

4.1.5 Learning Automaton

Since we did not know for sure what were the optimal parameters for creating gain sequences while performing SPSA, we wanted to optimize this aspect of our problem as well. We did this by implementing a Pursuit-based Learning Automaton as described in [30] and based on the pseudo-code in Algorithm 3. The values $\beta[0]$ and $\beta[1]$ as seen in in formula 9 and 10, were subject to optimization. In the implementation of the LA, they were given the minimum and maximum thresholds of $a_{min} = 0.05, a_{max} = 0.5$ and $b_{min} = 0.5, b_{max} = 0.1$, respectively. We then defined two vectors with length N, containing 100 different configurations for the parameters a and b as A and B. Then we created two additional vectors also with length N whose indices n were mapped to probabilities that we selected the n_{th} configuration of A and B, respectively. For each iteration, the LA tried the action of SPSA on our environment with different configurations for the vector β. Its ultimate goal was to maximize throughput of network flows while remaining undetected by Slips.

4.1.6 Container Environment

Docker was the obvious choice of container engine selected for this task. A baseline Dockerfile that could be used to configure a container with *all* experiment code can be viewed in Appendix A. Following is a short description of the most important components installed on each platform, along with a short explanation to why they were necessary for the experiment:

- **net-tools**: Includes binaries such as *netstat* and *ifconfig*, which eases the process of diagnosing potential networking issues.
- **git**: Enables us to fetch git resources.
- **python2.7**: The programming language that Slips is based upon. All code is written in Python 2.7.
- **iptables**: The firewall that enables IP-blocking functionality in Slips. Not relevant for the experiments conducted when modifying packet captures, but necessary for Slips to compile and run.

- **From Github: Stratosphere Linux IPS (branch: develop)**: Contains Stratosphere Linux IPS along with the models used for detection.
- **From Github: Stratosphere Testing Framework (branch: master)**: Includes modules that enable us to generate *biargus* files and *binetflow* files. Also includes additional modules that are handy when generating statistics on network flow files.
- **From Qosient: argus and argus-clients**: Audit Record Generation and Utilization System - required by Stratosphere Testing Framework to generate and analyze flow records. Includes the binaries *argus* and *ra*, which are vital components when using STF and Slips.
- **wireshark-common, tshark, tcpdump, libpcap0.8-dev, flex, bison, make, gcc, libncurses5-dev, libgeoip-dev, zlib1g-dev, libreadline7, libreadline6-dev, libssd-dev, libwrap0-dev**: dependencies of argus and argus-clients.
- **python-pip**: Necessary for installing python packets.
- **python-pip: prettytable, zodb, transaction, btrees, persistent-pip**: Python libraries that are dependencies of Slips and STF
- **python-pip: scapy, numpy, matplotlib, pickle, pandas**: additional python libraries necessary for self-implemented code.
- **python-tk**: required to work with the python pandas module.
- **adaptive_random_search.py**: Implementation of adaptive random search for simple parameter perturbation.
- **change_pcap_timestamps.py**: Implementation of the module that modified timestamps of packet capture files.
- **spsa-slips.py**: Implementation of Simultaneous Perturbation Stochastic Approximation algorithm for parameter perturbation.
- **la_slips.py**: Implementation of Learning Automaton for optimization of alogrithmic parameters

In order to get a viable range of results, it was desirable to run the experiments on an infrastructure that let us process multiple experiments simultaneously in isolated and independent environments. Docker Compose as a tool to run multi-container Docker applications was the natural choice for this task, and allowed us to scale with ease and take full advantage of the resources provided by the underlying platform. A Dockerfile with a select set of the above dependencies and file inclusions was created for each experiment. Then, images were created for each experiment and considered ready to be deployed in our multi-application environment. Having the file *docker-compose.yml* (Appendix B) available in our Experiments Directory, we ran *docker-compose up -d* to boot an instance of all experiments. The amount of containers running a particular experiment was then scaled up or down by issuing the command *docker-compose scale experiment-name=*number of desired active containers**. The platform that we ran our experiments on was a virtualized Ubuntu 16.04 operating system with 16GB RAM and 8 Virtual CPUs. This host was running a docker daemon, enabling containerization of experiments. We ran a total of 5 different experiments on the modification of existing packet captures. The experiments were run as 5 different services in a Docker Compose cluster and each service was scaled up to five units. After the services had completed their tasks, data was collected from each container by running the script in Listing 1.

```python
import docker
import collections

client = docker.from_env()
all_containers = client.containers.list(all=True)
relevant_containers = all_containers[:25]
container_dict = collections.defaultdict(list)

for i in relevant_containers:
    print(i.labels)
    if "random" in i.name:
        container_dict["1-random-perturbation"].append(i)
    elif "2-adaptive" in i.name:
        container_dict["2-adaptive-stepsize"].append(i)
    elif "3-simultaneous" in i.name:
        container_dict["3-simultaneous-perturbation"].append(i)
    elif "4-learning-automata" in i.name:
        container_dict["4-learning-automata"].append(i)
    elif "5-learning-automata-euclidean" in i.name:
        container_dict["5-learning-automata-euclidean"].append(i)
for k, v in container_dict.items():
    counter = 0
    for i in v:
        filename = i.name + ".tar"
        f = open(filename, "wb")
        bits, stat = i.get_archive("home/ubuntu/SlipsExperiments/data")
        for chunk in bits:
            f.write(chunk)
        f.close()
```

4.2 Semi-realistic Attack Scenario

The experiments described in this section aimed to test the feasibility of behavioral analysis systems in a semi-realistic environment that has already been compromised by an attacker. The attack scenario could be described as the C&C stage of a read team exercise [47], where it is assumed that a malicious entity already has compromised at least one host within a victim network, and that privileges are escalated to the extent that attackers are able to install arbitrary software on the host. During the C&C stage of a red team exercise, it is detrimental that penetration testers do not give away that they have compromised infrastructure on target systems. For that reason, when a communication channel with the control server is to be established, it is desirable that the malware's communication channel remains hidden from any IPS agents analyzing traffic on internal or external network endpoints.

4.2.1 Objectives

Substantial digital infrastructure was required to simulate the semi-realistic environment and software that this experiment required. The technical aspects of this set of experiments consisted of four main contributions:

- A portable, platform-independent container environment running digital infrastructure and code needed to support the experiments conducted.
- Two Discord bots interacting with each other over the Internet that can dynamically adapt network packet size, connection flow duration and amount of seconds to wait before initiating a new connection flow. The two discord bots implement the server and client functionality needed to conduct C&C acitivity. This structure resembles that of a *centralized C&C architecture* [6].
- Middleware that manages low-level network packet manipulation and communication between discord bots and a malicious trojan.
- A perturbation algorithm that is able to continuously feed the C2 channel with new network flow perturbation parameters.

4.2.2 Container environment

The technology used for creating a containerized environment is *docker-compose* [48]. To simulate a minimal but realistic environment, we deployed two services, both running Ubuntu 16.04 on kernels virtualized in Docker 18.06.1.

- **argus-rdp-client** resembles an organization's workstation and runs a Remote Desktop process for remote access to local resources. The host's only external-facing network interface is being monitored by an argus process running on the host, transmitting network flow logs to port 561. The owner of this workstation frequently uses a Discord client to communicate with peers. RDPClient is also infected by the AdaptiveC2Trojan trojan [49].
- **ips1** is a service dedicated to collecting network flow logs in the organization's internal network and monitoring them for anomalous behavior. The software that monitors network traffic is Slips. Security personnel in the target organization have collected network flows from benign traffic and used these to create models of what type of traffic should be considered normal. More importantly, they have security personnel have recorded data on the network traffic of malware typical to Discord, and use NBA to detect any significant similarity between live network traffic and malicious models. Hence, Slips' behavioral models can detect suspicious activity in Discord network flows over pre-defined periods of time.

Details for the configuration of the above-mentioned services can be found in the GitHub repository for this project [50], but will not be included in this report.

4.2.3 Argus configuration

The monitoring setup used for this project was highly simplified, but nevertheless suitable for the demonstrative purposes required to solve the problem statement. Below are the configuration details for *argus-rdp-client* and *ips1*, respectively.

argus-rdp-client

A comprehensive set of software components were required in order to run argus. All software dependencies were installed as part of the Docker image that is used for deployment of the infrastructure. Argus' configuration was based on the documentation provided by [51] and argus' man page. The apt-binaries, external tarballs, pip packages and configuration commands, as stated in Appendix F.X were added to *ubuntu-xrdp*'s Dockerfile [52] and remain pre-installed on *argus-rdp-client*'s host. This configuration should install any binaries and packages necessary for Argus to be able to run. Furthermore, the latest argus bundles are pulled from their respective sources before they were configured, made and installed.

argus-rdp-client by default exposes network flow logs on port 561, and a set of additional metrics are specified in argus' configuration file. Argus had to be configured to transmit the network flow metrics we expected to receive on *ips1*. The following configuration was set in **/etc/argus.conf** on *argus-rdp-client* to meet those requirements:

```
ARGUS_FLOW_TYPE="Bidirectional"
ARGUS_FLOW_KEY="CLASSIC_5_TUPLE"
ARGUS_ACCESS_PORT=561
ARGUS_INTERFACE=any
ARGUS_FLOW_STATUS_INTERVAL=300
ARGUS_MAR_STATUS_INTERVAL=60
ARGUS_GENERATE_RESPONSE_TIME_DATA=yes
ARGUS_GENERATE_PACKET_SIZE=yes
ARGUS_GENERATE_JITTER_DATA=yes
ARGUS_GENERATE_MAC_DATA=yes
ARGUS_GENERATE_APPBYTE_METRIC=yes
ARGUS_GENERATE_TCP_PERF_METRIC=yes
ARGUS_GENERATE_BIDIRECTIONAL_TIMESTAMPS=yes
ARGUS_CAPTURE_DATA_LEN=600
ARGUS_FILTER_OPTIMIZER=yes
ARGUS_KEYSTROKE="yes"
```

If configured according to the above settings, Argus could then be run as a daemon exposing network flows on port 561 by issuing the command *argus -i eth0 -d*:

```
root@ce2628a055f4:/home/ubuntu# argus -i eth0
     ArgusAlert: 10 Nov 18 21:14:48.414642 started
     ArgusAlert: 10 Nov 18 21:14:48.492937
         ArgusGetInterfaceStatus: interface eth0 is up
```

ips1

Ra was installed on ips1 according to the specification in [51]. On *ips1*, ra was run as a program with its standard output piped to the Slips application. The command *ra -f StratosphereLinuxIPS/ra.conf -n -Z b | python slips.py -c slips.conf -f models/ -d* was issued in order for *ra* to listen to the network interface *eth0* on port 561, transmitting all network flow logs to the Slips application. By applying the models specified in the *models/* directory, Slips performs behavioral analysis on the incoming network flow data and determines whether it matches an existing profile.

4.2.4 Adaptive C&C Trojan

The implementation of this project's C2 malware resembled that of a simplified *centralized C&C* architecture because the bot(s) attempted to establish their communication channel with a limited amount of C2 servers. For the scenario this project's experiment is based upon, a set of assumptions were made:

- The attacker has already overcome *initial infection*, *secondary infection* and *connection* phases and is at a stage of infection where the objective is to **maintain malicious C&C** [6].
- Slips for network behavior analysis is the only IDPS mechanism that exists in the network.
- The attacker has compromised a user of the popular communication service Discord and may communicate over this channel with an arbitrary peer.
- The attacker tolerates significant delay in communication with infected hosts because network parameters related to time must be dynamically adaptable.

Discord bots

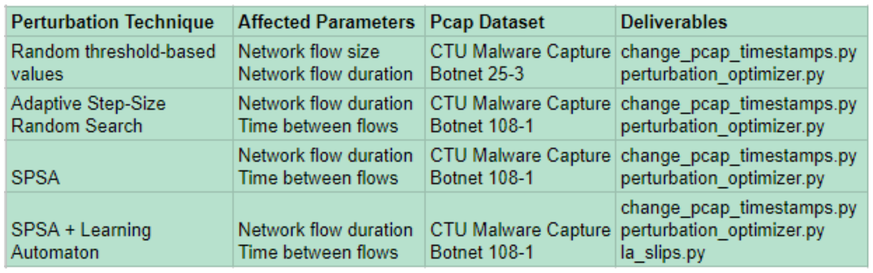
The discord bots used for communication between C2 server and infected host need to be able to perform a set of delay-tolerant core procedures:

- **Client:** Submit heartbeat message to C2 server and wait for a response to confirm that it is online.
- **Client:** Retrieve an instruction set from the C2 server.
- **Client:** Retrieve *perturbation* instruction set from the C2 server. (Or generate this set on the host, transmitting the perturbed parameters to the server)
- **Client:** Execute instruction sets and provide feedback on the results.
- **Client:** For each communication flow, dynamically adjust relevant network parameters.
- **Server:** Transmit partially fragmented instruction sets with perturbations on time- and size related parameters according to payload received from client.

Unfortunately, this is as far as we came with respect to a Semi-realistic Attack Scenario. The ambition of implementing a bot whose behavior changed based on perturbations to continuous network parameters was somewhat out of reach based on the time available for this project. We were able to implement some bot-like features over the Discord API, but lacked the time necessary to produce code for adaptive botnet behavior, middleware between the Discord API and trojan software and low-level packet manipulation modules. Some additional thoughts on this part of the project is provided in chapter 6.

5 Results

This section provides a short description of the experiments that were conducted as part of this project and the results produced by each experiment. The experiments were designed to provide data that reflected the performance of Slips, or to serve as building blocks that justify other experiments conducted in this project. Data collected from the experiments were stored as Pickle objects. The graphs throughout this section are based on pickled objects and *pandas* data frames. Figure 4 represents an overview of the experiments that were conducted as part of this project.



Perturbation Technique	Affected Parameters	Pcap Dataset	Deliverables
Random threshold-based values	Network flow size Network flow duration	CTU Malware Capture Botnet 25-3	change_pcap_timestamps.py perturbation_optimizer.py
Adaptive Step-Size Random Search	Network flow duration Time between flows	CTU Malware Capture Botnet 108-1	change_pcap_timestamps.py perturbation_optimizer.py
SPSA	Network flow duration Time between flows	CTU Malware Capture Botnet 108-1	change_pcap_timestamps.py perturbation_optimizer.py
SPSA + Learning Automaton	Network flow duration Time between flows	CTU Malware Capture Botnet 108-1	change_pcap_timestamps.py perturbation_optimizer.py la_slips.py

Figure 4: Experiments conducted as part of this project.

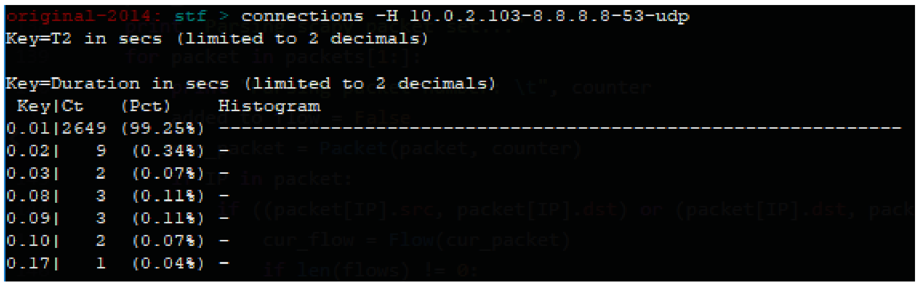
A central part of the experiments involves measuring the performance of our perturbation techniques by evaluating the *magnitude* of perturbation (See 3.2.2 - Perturbation measurement and 4.1.4 Formula 6, 7). The *magnitude* of perturbation is defined as the Euclidean distance between the perturbed vector and the observed vector in the *time, flow size* space. However, since we have two values that affect the *time* space - namely *duration* and *time between network flows*, and since we have objective functions that use the parameter *size* defined, it is important to note that any change in size may also affect the magnitude of perturbation, and for SPSA and LA also the time-related parameters. This is important because it implies that there are dependencies between the size parameter and time-related parameters despite the fact that we do not change the actual size of network flows. Hence, the *magnitude* attribute does not necessarily represent only the performance of the time-related perturbations.

5.1 Malicious Packet Capture Files

The network flow parsing procedure that was implemented as a core functionality of this project required an initial packet capture file as input. We selected two different packet capture files from [53] whose malware show different C&C behavior. The network flows generated by the two different types of malware match models in Slips' *models/* directory and their behavior, if unmodified, is hence evaluated as malicious. The two packet capture files that were subject to processing were CTU25-3 (CTU25-3) [22] and CTU Malware Capture Botnet 108-1 (CTU108-1) [54]. The first packet capture file represents the C&C behavior of Zbot malware [55], which primarily consists of short DNS connections between the IP addresses (10.0.2.103, 8.8.8.8). The second packet capture file likely represent behavior associated with the Cridex Trojan [56], whose malicious behavior is observed as connections between the IP addresses (10.0.2.107, 212.59.117.207) and (10.0.2.107, 91.222.139.45). These IP tuples were used as values for the attribute *target_tuples* in the implementation of packet parsing procedures. In order to extract meaningful information from the results of our experiments, an initial assessment of the two packet capture files will serve as a benchmark for analysis.

5.1.1 Zbot characteristics

We added the original packet capture file from [22] to an STF dataset by navigating to the StratosphereTestingFramework folder and issuing the command *python stf.py* to open the STF console, followed by the command *datasets -c /home/ubuntu/SlipsExperiments/trials/2014-02-07_capture-win3.pcap* to select our packet capture file as part of a dataset. We then generated biargus and binetflow files by issuing the command *datasets -g*, and a list of connections by issuing *connections -g*. The connections related to the generated dataset could then be displayed by issuing *connections -L 0*. From this overview, we observed that there were a total of 2669 connections between the IP addresses 10.0.2.103 and 8.8.8.8. Furthermore, we were now able to investigate some behavioral characteristics of the network flows belonging to those connections by issuing the command *connections -H 10.0.2.3-8.8.8.8-53-udp*. We observed that 99.25% of connections in this packet capture file had a duration of 0.01 seconds, depicted as the screenshot in Figure 5:

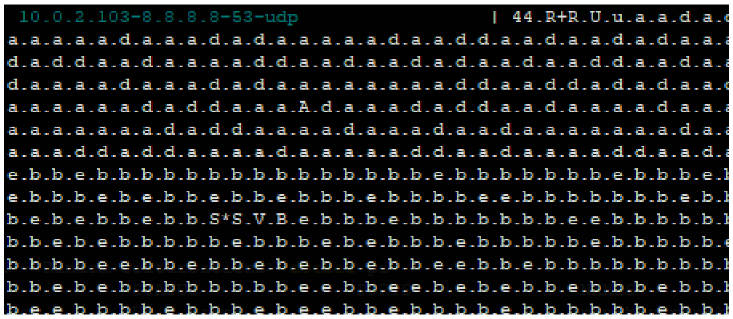


```
original-2014: stf > connections -H 10.0.2.103-8.8.8.8-53-udp
Key=T2 in secs (limited to 2 decimals)

Key=Duration in secs (limited to 2 decimals)
 Key|Ct    (Pct)    Histogram
0.01|2649 (99.25%) ------------------------------------------------------------
0.02|   9  (0.34%) -
0.03|   2  (0.07%) -
0.08|   3  (0.11%) -
0.09|   3  (0.11%) -
0.10|   2  (0.07%) -
0.17|   1  (0.04%) -
```

Figure 5: Duration of connections in CTU25-3

To generate the models that STF uses when comparing incoming network traffic with known malicious patterns, we may issue the STF console command *models -g*. Since connections for the tuple 10.0.2.103-8.8.8.8-53-udp represented a majority of connections in this packet capture, its state representation in STF is quite long. By comparing the state representation values for the tuple 10.0.2.103-8.8.8.8-53-udp in Figure 6 with the character representations in Figure 1, we can see that the flows in these connections have short durations, small- to medium network flow sizes and strong periodicity. Figure 6 is meant to show a minimal representation of our subject tuple's state, as represented in the STF console.



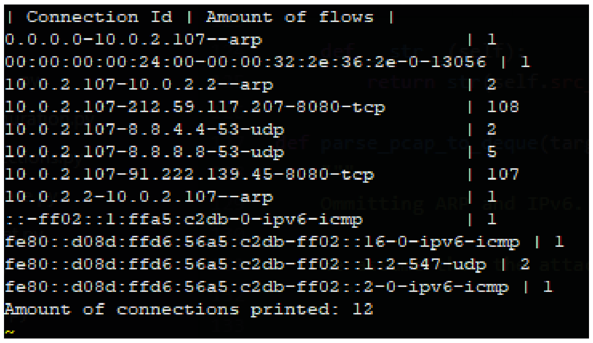
```
 10.0.2.103-8.8.8.8-53-udp                    | 44.R+R.U.u.a.a.d.a.
a.a.a.a.a.d.a.a.a.d.a.d.a.a.a.a.a.d.a.a.d.d.a.a.d.a.a.d.a.d.a.a.
d.a.d.d.a.a.d.a.a.a.a.d.d.a.a.d.a.a.d.a.d.a.a.d.a.a.d.d.a.a.d.a.
d.a.a.a.a.d.a.a.a.d.a.d.a.a.a.a.a.a.a.a.a.d.d.a.a.d.d.a.d.a.d.a.a.
a.a.a.a.a.a.d.a.d.d.a.a.a.A.d.a.a.a.d.a.d.d.a.a.d.a.a.a.a.d.a.a.
a.a.a.a.a.a.a.d.a.d.d.a.a.a.a.d.a.a.a.d.a.a.d.a.a.a.a.a.a.a.d.a.
a.a.a.d.d.a.d.d.a.a.a.a.d.a.a.a.a.a.d.d.a.a.d.a.a.a.a.d.d.a.a.d.
e.b.b.e.b.b.b.b.b.b.b.b.b.b.b.b.b.b.b.e.b.b.b.b.b.b.b.e.b.b.b.e.
e.b.b.b.e.b.e.b.b.e.b.b.e.b.b.b.e.b.b.b.e.e.b.b.b.b.b.b.e.b.
b.e.b.e.b.b.e.b.b.S*S.V.B.e.b.b.b.e.b.b.b.b.b.b.b.e.e.b.b.b.b.b.
b.b.e.b.e.b.b.b.b.b.e.b.b.b.e.b.e.b.b.b.e.b.b.b.b.e.b.b.b.b.b.
b.b.b.b.e.e.b.b.e.b.b.e.b.e.b.b.e.b.b.e.e.b.b.e.b.b.b.b.b.b.b.b.
b.b.e.b.e.b.b.b.b.b.b.b.b.e.e.b.b.e.e.b.e.b.e.b.b.e.b.b.b.e.b.b.
b.e.e.b.b.b.b.e.b.b.b.e.b.e.e.b.b.e.b.b.b.b.e.b.b.b.b.b.b.e.b.b.
```

Figure 6: State representation of 10.0.2.3-8.8.8.8-53-udp in CTU25-3.

This state matches a model in the Slips/models directory, namely the *From-Botnet-UDP-DNS-DGA-17* model, and the IP address 10.0.2.103 is for that reason evaluated as malicious when the binetflow file is provided as input to Slips.

5.1.2 Cridex characteristics

Due to the packet parser's poor performance (See Section 4.1.2 - *Parsing the packet capture file*), we chose to extract the first 1000 packets from this packet capture file, if Slips was able to detect malicious behavior on the pcap's binetflow representation. As with Zbot, we added the sliced packet capture file for the Cridex Trojan to an STF dataset in order to evaluate its baseline characteristics. The new binetflow file was created and added to the *trials/* directory as *2015-03-09_capture-win7-first1k.pcap*. By looking at its connections, we identified two outlier connections as 10.0.2.107-212.59.117.207-8080-tcp and 10.0.2.107-91.222.139.45-8080-tcp, responsible for 108 and 107 flows, respectively. Of a total of 234 flows, the two outlier connections had 215 flows combined, as seen in Figure 7.

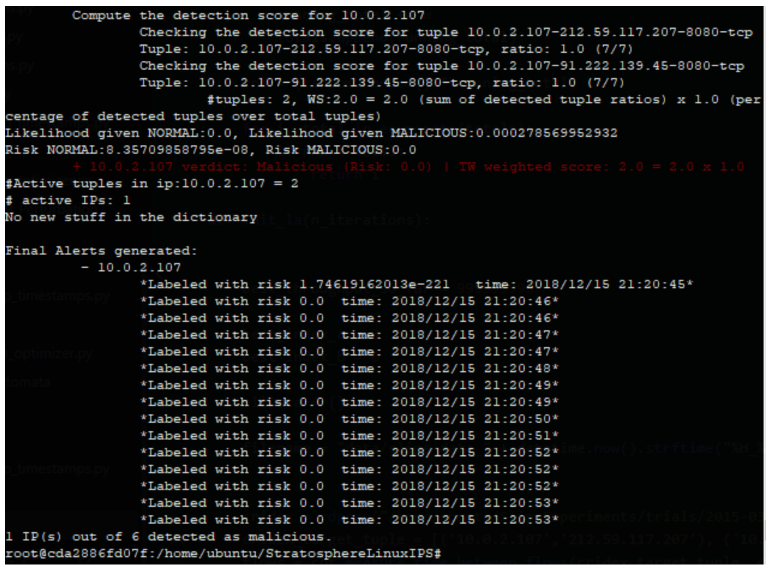


```
| Connection Id | Amount of flows |
0.0.0.0-10.0.2.107--arp                  | 1
00:00:00:00:24:00-00:00:32:2e:36:2e-0-13056 | 1
10.0.2.107-10.0.2.2--arp                 | 2
10.0.2.107-212.59.117.207-8080-tcp       | 108
10.0.2.107-8.8.4.4-53-udp                | 2
10.0.2.107-8.8.8.8-53-udp                | 5
10.0.2.107-91.222.139.45-8080-tcp        | 107
10.0.2.2-10.0.2.107--arp                 | 1
::-ff02::1:ffa5:c2db-0-ipv6-icmp         | 1
fe80::d08d:ffd6:56a5:c2db-ff02::16-0-ipv6-icmp | 1
fe80::d08d:ffd6:56a5:c2db-ff02::1:2-547-udp | 2
fe80::d08d:ffd6:56a5:c2db-ff02::2-0-ipv6-icmp | 1
Amount of connections printed: 12
~
```

Figure 7: Outlier connections

These IPs were chosen as candidates for the value of *target tuples* for experiments using this packet capture file.

We confirmed that Slips was able to evaluate Cridex' new binetflow file as malicious by running the command *python slips.py -c slips.conf -f models/ -r /home/ubuntu/SlipsExperiments/trials/2015-03-09_capture-win7-first1k.binetflow -e 5*, as seen in Figure 8. The -e 5 option was added as an argument to slips.py to display more detailed information about the model that triggered on our binetflow file.



```
        Compute the detection score for 10.0.2.107
                Checking the detection score for tuple 10.0.2.107-212.59.117.207-8080-tcp
                Tuple: 10.0.2.107-212.59.117.207-8080-tcp, ratio: 1.0 (7/7)
                Checking the detection score for tuple 10.0.2.107-91.222.139.45-8080-tcp
                Tuple: 10.0.2.107-91.222.139.45-8080-tcp, ratio: 1.0 (7/7)
                        #tuples: 2, WS:2.0 = 2.0 (sum of detected tuple ratios) x 1.0 (per
centage of detected tuples over total tuples)
Likelihood given NORMAL:0.0, Likelihood given MALICIOUS:0.000278569952932
Risk NORMAL:8.35709858795e-08, Risk MALICIOUS:0.0
        + 10.0.2.107 verdict: Malicious (Risk: 0.0) | TW weighted score: 2.0 = 2.0 x 1.0
#Active tuples in ip:10.0.2.107 = 2
# active IPs: 1
No new stuff in the dictionary

Final Alerts generated:
         - 10.0.2.107
                *Labeled with risk 1.74619162013e-221  time: 2018/12/15 21:20:45*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:46*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:46*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:47*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:47*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:48*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:49*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:49*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:50*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:51*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:52*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:52*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:52*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:53*
                *Labeled with risk 0.0  time: 2018/12/15 21:20:53*
1 IP(s) out of 6 detected as malicious.
root@cda2886fd07f:/home/ubuntu/StratosphereLinuxIPS#
```

Figure 8: After analyzing the first 1000 packets of CTU108-1, the IP address 10.0.2.107 received the verdict *malicious*.

5.2 Modifying existing packet captures

5.2.1 Random threshold-based perturbation

A 2D python list representing thresholds for the random perturbation function was defined as

```
[[flow.length - (flow.length * 0.2), flow.length + (flow.length *
↪  0.2)], [flow.td - (flow.td*0.3), flow.td + (flow.td * 0.35)],
↪  [td_next - (td_next*0.3), td_next + td_next*(0.25)]],
```

and provided as an argument to the random_perturbation() function for each flow subject to perturbation. Recall that the size parameter does not affect the size of flows, but has an impact on magnitude. The packet capture from CTU25-3 was provided as input to the packet parser. We ran a total of two iterations of random threshold-based perturbation using this configuration. By investigating the amount of flows per connection in the newly created binetflow files, we observed no change from the initial network flow capture (See 5.1.1), as seen in Figure 9. This was a strong indication that the packet parsing procedure worked as expected.

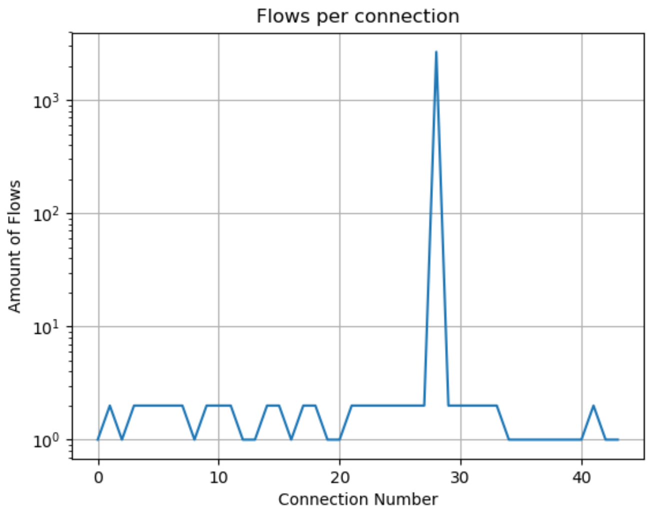


Figure 9: *Amount of network flows per connection after parsing and perturbing network flow parameters on CTU25-3. The connection 10.0.2.103-8.8.8.8-53-udp has Connection Number 28, with 2669 flows.*

Furthermore, we observed that by using the perturbation thresholds defined in $p1$, we produce a large amount of perturbation while not triggering any Slips alerts. Table 1 represents an overview of the first 4 iterations of this experiment, using different sets of threshold parameters. We observed that, by providing considerable or only a slight amount random perturbation to each parameter, we may evade detection.

Iter	**Detected**	**Thresholds Duration**	**Thresholds TD**
0	Yes	$t_{lower} = 0, t_{upper} = 0$	$t_{lower} = 0, t_{upper} = 0$
1	No	$t_{lower} = 0.3, t_{upper} = 0.35$	$t_{lower} = 0.3, t_{upper} = 0.25$
2	No	$t_{lower} = 0.03, t_{upper} = 0.035$	$t_{lower} = 0.03, t_{upper} = 0.025$
3	Yes	$t_{lower} = 0.0003, t_{upper} = 0.00035$	$t_{lower} = 0.0003, t_{upper} = 0.00025$

Table 1: Varying thresholds for Random Perturbation

Since this packet capture file consists of a set of connections where most packets belong to the same flow, a single perturbation to the flows object would affect the timing of all other packets in that flow. CTU25-3 only contains one malicious connection, so a perturbation to that flow object will affect almost all the packets in the capture file.

By comparing the models generated from the initial CTU25-3 binetflow file

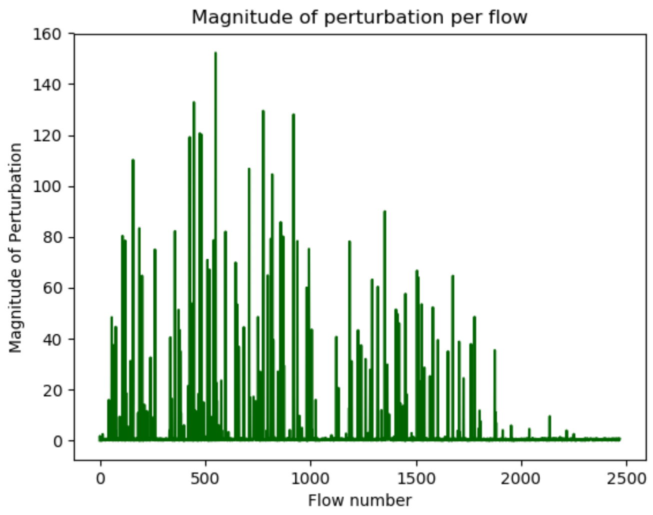


Figure 10: Magnitute of perturbation per flow for iteration 1

with the binetflow file from iteration 3, which was detected by Slips as malicious, we observed that while the connections retained a high degree of periodicity, they increased significantly in duration.

Furthermore, for each of the three runs that had their parameters perturbed, we recorded the magnitude of all perturbations as the average Euclidean distance between the perturbed connections and the original connections. The Euclidean distance was calculated as the p-norm distance between the average of original network flow parameters, and the perturbed version of those parameters.

Iter	**Mean Magnitude**	**Detected**	**Final Timestamp**
0	0	True	5754.224
1	26.68	False	5752.725
2	2.78	False	5752.679
3	0.68	True	5752.954

Table 2: Magnitudes for perturbations with varying thresholds on CTU25-3

By looking at iteration 1, we observe that the magnitude of perturbation does not necessarily correspond with a large change in the final perturbed duration, because we select random values from *around* the initial values from a uniform distribution. This could indicate that, by choosing to only select random values from a uniform distribution, we will approximate the initial value in the long run, which is not desirable if the goal is to reduce the level of periodicity of network flow patterns.

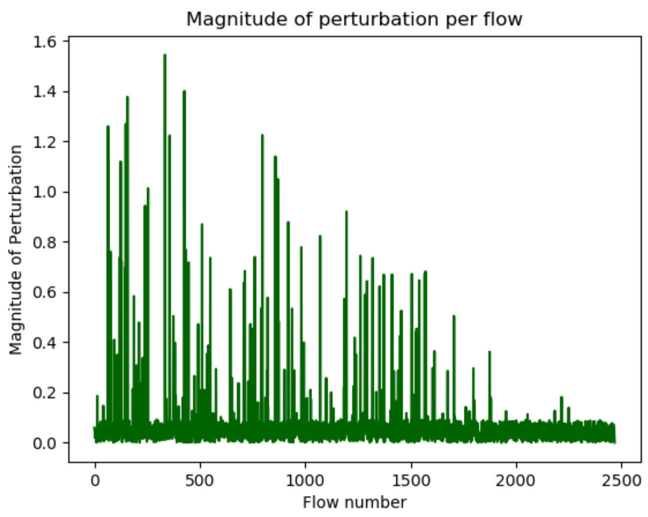


Figure 11: Magnitute of perturbation per flow for iteration 3

5.2.2 Adaptive Step-size Random Search

Since ASRS is also threshold-based, we wanted to run a larger amount of iterations using the same set of parameters as in Iteration 2 of 5.2.1, with the goal of observing whether a slightly more intelligent way of selecting perturbation parameters could result in a smaller magnitude of perturbation. Furthermore, we decided to run the SPSA algorithm with the Cridex dataset, such that we could look at flows with some varying flow characteristics. We defined the following 2D python list:

```
[[flow.length - (flow.length * 0.2), flow.length + (flow.length *
↪  0.2)], [flow.td - (flow.td*0.3), flow.td + (flow.td * 0.35)],
↪  [td_next - (td_next*0.3), td_next + td_next*(0.25)]],
```

With respect to magnitude per flow, ASRS performed a lot better than simple random variables. When using the same parameters as in Iteration 2 of 5.2.1, the average magnitude was *1.96*, compared to the much higher 2.78 using random thresholds.

By observing the magnitudes of perturbation per flow in Figure 12, we spot one outlier network flow, with roughly 5 times the average value:

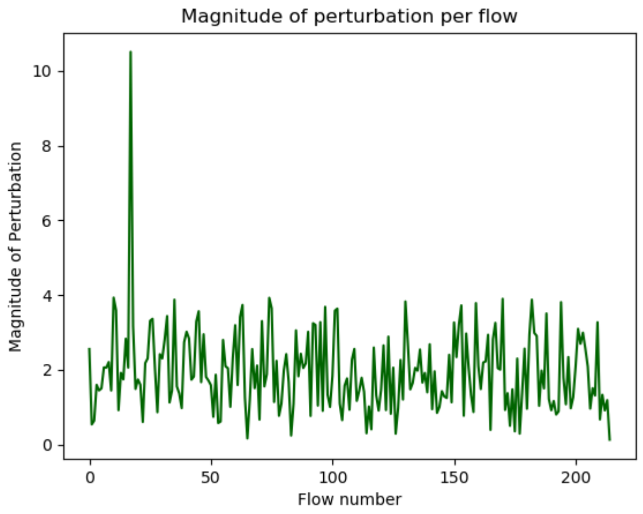


Figure 12: Magnitude of perturbation per flow for CTU108-1 using ASRS

Because ASRS bases perturbation on initial values and thresholds, it is very likely that this is due a larg observational value.

The mean standard deviation of magnitude for CTU108-1 using ASRS was measured to *1.60*, while the mean variance was measured to *2.75*.

If our goal is to minimize magnitude while remaining undetected, ASRS provides us with a consistent improvement in performance because it yields a smaller magnitude of perturbation while remaining undetected in all cases.

Iter	**Mean Magnitude**	**Detected**
0	0	True
1	20.68	False
2	1.96	False
3	0.41	False

Table 3: Magnitudes for perturbations with varying thresholds on CTU108-1

5.2.3 Simultaneous Perturbation Stochastic Approximation

This experiment did not require any bounds or thresholds in order to run the perturbation algorithm. Using the static default parameters for the vector β as $[0.1, 0.9]$ to generate gain sequences, we ran a total of 50 iterations. We used the objective function as stated in Formula 6 for maximizing throughput. All solutions generated by the SPSA were *feasible*, i.e. not detected as Slips as malicious. The mean of the mean magnitudes for all trials was *4.18*, indicating an increase compared to the ASRS algorithm. This is likely because we, for these trials, used the objective function that attempted to maximize throughput by minimizing duration of flows and the time between flows.

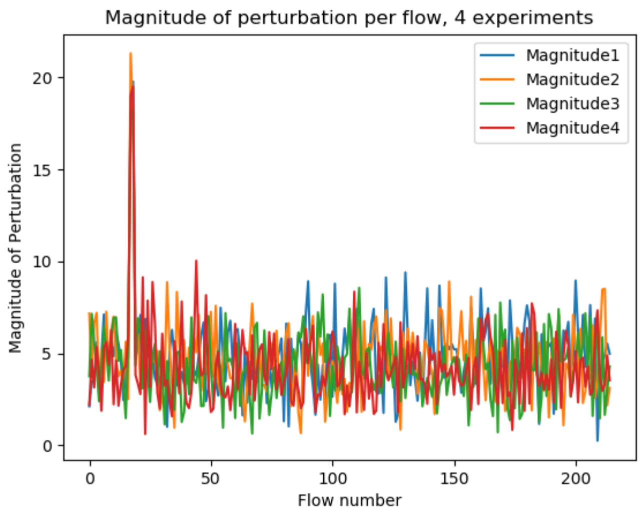


Figure 13: Magnitude of perturbation per flow for 4 iterations on CTU108-1 using SPSA with static gain sequence parameters.

We can observe from the graph in Figure 13 not only that the magnitudes are more severe than when using the ASRS algorith, but that the variance in magnitude is also significantly increased. We measured the mean standard deviation and variance for all 50 iterations of this experiment to *4.78* and *2.18*, respectively. This could imply that the algorithm's gain sequence parameters are not sufficiently tuned to work with our solution.

5.2.4 Simultaneous Perturbation Stochastic Approximation with Learning Automaton

We deployed an experiment on our implementation of the LA by initializing 15 iterations on the Learning Automaton, first using the objective function stated in Formula 6 for maximizing throughput. By looking at Figure 14, we observe that the magnitude of perturbation on average declines in the direction of positive iterations, suggesting that the SPSA algorithm *could* receive improved parameters over time.

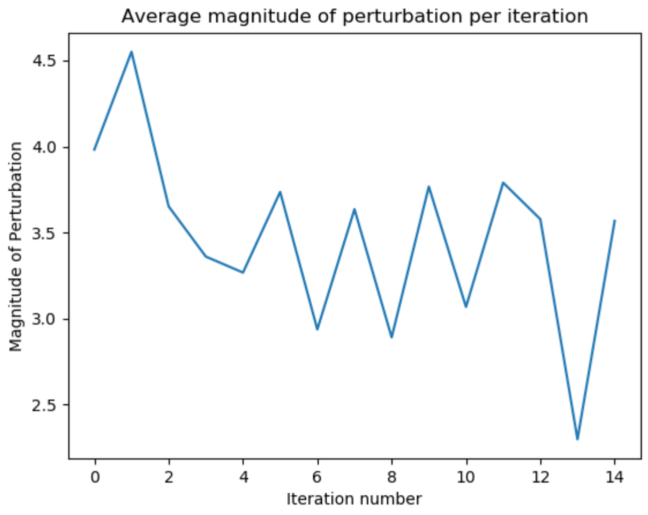


Figure 14: 15 iterations over CTU108-1 using SPSA with dynamic gain sequence parameters and a throughput-optimizing objective function.

However, we also observe a spike in magnitude on the last iteration, indicating that the LA hit a roulette selection with corresponding parameters that yielded a higher magnitude, i.e a reduced reward, for its iteration. If we were to conduct an experiment with a significantly higher amount of iterations, we would have been able to observe a much better overview of the effect that dynamic gain sequence parameters had on the SPSA algorithm.

We ran a final experiment on the LA using SPSA with the Euclidean objective function as stated in Formula 7. We ran a total of 25 iterations and observed the average magnitude of perturbation for each iteration as shown in Figure 15.

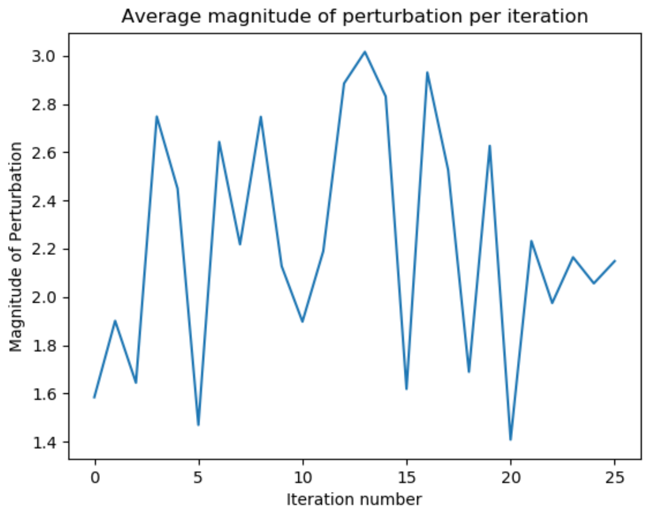


Figure 15: 25 iterations over CTU108-1 using SPSA with dynamic gain sequence parameters and a Euclidean distance-optimizing objective function

The average magnitude of perturbation in Figure 15 is significantly lower than that of Figure 14. This is the expected result because the objective function in this experiment aims to minimize magnitude for each SPSA iteration.

Objective	**Mean**	**Var**	**STD**	**Iterations**
Formula 6	3.472	0.284	0.533	15
Formula 7	2.220	0.236	0.486	25

Table 4: Two experiments using LA with SPSA for the objective functions defined in Formula 6 and 7.

By inspecting the overview of mean, average and standard deviation for the average magnitude in the two different experiments, we can see that the values for mean and variance differ by roughly 16% and 8%, respectively. This is not a significant amount of difference, particularly considering we ran different amount of iterations for each experiment. We can not conclude on the best set of parameters because we have not executed enough iterations on the LA.

5.3 Semi-realistic Attack Scenario

Unfortunately, due to lack of a feasible implementation, we were not able to conduct any experiments on this matter. For that reason, no results are available.

6 Discussion and Future Research

6.1 Problem statement

We initially wanted to investigate whether one could perturb the network parameters of *live* network traffic and hence be able to practically confuse NBA. The packet parsing procedures and their testing against Slips were initially meant as short reconnaissance studies that would serve as justifying factors for further experiments on the manner. However, as these initial investigations proved to be more and more time-consuming, it became natural to make them the main focus of this project. Implementing a tool to use intelligent perturbation techniques for further testing of Slips could serve as an interesting topic for future research, but would require significantly more time than what was available for this project.

Additionally, a project of the above-mentioned nature would absolutely need to focus more on how one could improve current solutions, an aspect this project to some extent lacked. It is detrimental that security-related research has an ultimate goal of hardening systems, which future research on this topic would indeed require.

6.2 Related Work

Finding research and selecting topics for this section was not hard, but structuring the vast amount of different information into a logical build-up with relevant content was not trivial. Most of the research done on adversarial machine learning employed some sort of *deep learning* algorithm, which was out of scope for this project. We extracted the information that was relevant to our problem statement and attempted to apply it to our own approach, to the extent where that was possible. For instance, the concept of providing perturbation to parameters was applicable to our problem domain, so we attempted to use those same concepts on our own simplified model for parameter perturbation. Because of the large amount of research done on adversarial deep learning systems, future projects on the topic of evading behavioral IDS may choose an approach where more advanced artificial intelligence is used as a tool for adapting network parameters.

6.3 Approach

We started the coding of this project by building the elementary components of the packet parsing functionality. We could have had an even more structured approach in this regard, considering that the potential for improvement of the final product's run-time execution was significant.

6.3.1 Key Concepts and Class Structures

Flow class structure: We wanted to maintain a Flow class structure with reduced complexity to ease the workload associated with the implementation. For this reason, we chose to omit packets that were not IPv4 packets from Flow objects. This implies that we categorize packets that are part of protocols such as IPv6, ARP, DHCP, etc, as single packets not belonging to any Flow object. We recognize that by choosing to exclude such packets from Flow objects, we limited ourselves to *only* parse packet capture files whose malicious behavior was generated by applications that only use the IPv4 protocol. When the logic surrounding network flow objects was built, we based the concepts on Cisco's NetFlow standard and extracted the features that we wanted to use [57]. Excluded features of Cisco's NetFlow standard included Ingress interface and IP Type of Service. We ensured that we did not rely on these parameters while parsing any of the files used in the experiments.

Flows set data structure: The data structure used when storing Flow and Packet objects was *not* ideal. In fact, this remained a significant limitation to the implementation of the packet parser because the operation of adding Packet objects to their respective Flows was *far* too complex. The execution time of the *measure_time_between_flows()* method was $T(n) \in O(n^2)$, which rendered this solution not feasible for large packet files. There was no good reason to choose a deque as data structure, especially since we don't use any of its properties. If we had instead used a B-tree, using the Packet object's PacketNo as search index, we could reduce the packet parser's complexity $T(n)$ down to an improved $T(n) \in O(logn)$ execution time. Implementing this modification was desirable, but not feasible due to time constraints.

Excluding the size parameter: There were a few reasons as to why we chose not to include the size parameter for perturbation, despite the fact that all perturbation algorithms support the operations. First of all, we wanted a reduced complexity of the flow class when conducting the initial parsing. Then, after looking into the effects that parameter perturbation of time-related parameters had on the periodicity of flows, we decided it would be sufficient to only cause perturbations to those. This also reduced the complexity of our experiments, which was desirable given the limited scope of this project.

6.4 Experiments

6.4.1 Code and Infrastructure Challenges

Thanks to the brilliant features of Docker and Docker Compose, experiments were trivial to launch. However, as we started running experiments, we quickly discovered small bugs in our code that needed fixing in order to gain integrity in results. This proved to be *extremely* time-consuming, and often resulted in us having to re-implement existing code.

Due to computational exhaustion, we were not able to test our implementation on larger packet captures, effectively rendering us without results for testing of malicious patterns that span over days, weeks or months.

The exclusion of the *size* parameter did to some extent affect the results of our experiments because, by including that parameter when calculating Euclidean distance, we could *not* use the *magnitude (seconds)* unit for axes descriptions.

The LA concept was easy to implement but computationally hard to execute. We initially wanted to run it for thousands of iterations, but because of time limitations and the constant need to update the source code for improvement in efficiency or data collection, we ended up executing fewer iterations than originally planned. However, we still believe that the LA serves its demonstrative purpose of being able to optimize parameters for an SA algorithm.

7 Conclusion

The results of this thesis may contribute to increasing awareness around the importance of comprehensive intrusion detection mechanisms for services that require some degree of security. In a digital society where threats emerge faster than their respective security measures, one may assume that malicious actors and their software is trivially able to adapt to new environments. Information Security specialists must ensure that their security software, and the models they depend on, are at all times up to date and on-par with the current highest standard. However, as we have shown in this project, security professionals should also be careful not to blindly trust existing models.

We set out to answer how the behavior of malicious network traffic could be altered, with the goal of evading detection by a behavioral analysis tool. While we did not prove that evasion was possible for a live solution, we did conceptually show that by creating a packet manipulation scheme supporting perturbations to network flow parameters, we may perturb network flow patterns to effectively evade NBA intrusion detection. Furthermore, we explored and implemented different search techniques that provided perturbed versions of an initial set of network flow parameters. Finally, we demonstrated how a simple reinforcement-based ML method could be used as a tool to provide optimal parameters for the perturbation algorithms that were implemented.

Appendices

A Dockerfile configuration for experiments on Stratosphere Linux IPS

```
#
# Stratosphere IPS Dockerfile
#
# https://github.com/stratosphereips/StratosphereLinuxIPS
#
# Build with:
# docker build -t slips-experiments /path/to/Dockerfile/folder

# Pull base image.
FROM ubuntu:latest

ENV DEBIAN_FRONTEND noninteractive

# Install slips and argus client.
RUN \
  apt-get update && \
  apt-get install -y --no-install-recommends apt-utils
RUN \
  apt-get update && \
  apt-get install -y software-properties-common && \
  apt-get install -y net-tools && \
  apt-get install -y git && \
  rm -rf /var/lib/apt/lists/* && \
  apt-get update && \
  apt-get install -y python2.7 && \
  apt-get install -y iptables && \
  mkdir /home/ubuntu
WORKDIR /home/ubuntu/
RUN \
  git clone --single-branch -b develop
   ↪ https://github.com/stratosphereips/StratosphereLinuxIPS.git
   ↪ && \
  git clone
   ↪ https://github.com/stratosphereips/StratosphereTestingFramework
RUN \
  apt-get install -y wget && \
  apt-get install -y curl
RUN \
  curl https://qosient.com/argus/src/argus-3.0.8.2.tar.gz
   ↪  --create-dirs -o /home/ubuntu/argus-3.0.8.2.tar.gz && \
  curl https://qosient.com/argus/src/argus-clients-3.0.8.2.tar.gz
   ↪  --create-dirs -o /home/ubuntu/argus-clients-3.0.8.2.tar.gz
RUN \
  tar -xzvf /home/ubuntu/argus-clients-3.0.8.2.tar.gz -C
   ↪  /home/ubuntu/ && \
  tar -xzvf /home/ubuntu/argus-3.0.8.2.tar.gz -C /home/ubuntu/
RUN \
  apt-get update && \
  apt-get install --fix-missing -y wireshark-common
RUN \
  apt-get install -y tshark tcpdump libpcap0.8-dev flex bison &&
   ↪  \
  apt-get install -y make && \
  apt-get install -y vim && \
  apt-get install -y build-essential gcc
RUN apt-get install -y libncurses5-dev libncurses5-dev
RUN apt-get install -y make libgeoip-dev zlib1g-dev libreadline7
 ↪  libreadline6-dev libbsd-dev libwrap0-dev
RUN sh /home/ubuntu/argus-3.0.8.2/configure
WORKDIR /home/ubuntu/argus-3.0.8.2/
RUN ./configure && make && make install
WORKDIR /home/ubuntu/argus-clients-3.0.8.2/
RUN ./configure && make && make install
RUN ls -la /home/ubuntu/argus-clients-3.0.8.2/
RUN apt-get install -y python-dateutil
RUN apt-get install -y python-pip
RUN pip install prettytable zodb transaction btrees persistent
 ↪  scapy numpy matplotlib
WORKDIR /home/ubuntu
RUN mkdir SlipsExperiments
WORKDIR /home/ubuntu/SlipsExperiments
RUN mkdir trials
ADD adaptive_random_search.py /home/ubuntu/SlipsExperiments/
ADD change_pcap_timestamps.py /home/ubuntu/SlipsExperiments/
ADD la_slips.py /home/ubuntu/SlipsExperiments/
ADD spsa_slips.py /home/ubuntu/SlipsExperiments/
ADD stf /home/ubuntu/SlipsExperiments/stf/
ADD pcaps/*.pcap /home/ubuntu/SlipsExperiments/trials/
RUN mkdir confs
RUN cp /home/ubuntu/StratosphereTestingFramework/confs/ra.conf
 ↪  confs/
RUN la_slips.py

\end{lstlisting}
```

```
\clearpage
\section{Docker Compose configuration: docker-compose.yml}

\begin{minted}[
    gobble=4,
    frame=single,
    linenos]{yaml}
    version: '3.7'
    services:
        1-random-perturbation:
        build:
            context: "./1 - Random Perturbations"
            dockerfile: Dockerfile
        image: 1-random-perturbation:latest
        tty: true
        entrypoint:
            - python
            - change_pcap_timestamps.py

        2-adaptive-stepsize:
          build:
              context: "./2 - Adaptive Step-Size"
              dockerfile: Dockerfile
          image: 2-adaptive-random-search:latest
          tty: true
          entrypoint:
            - python
            - change_pcap_timestamps.py

        3-simultaneous-perturbation:
          build:
              context: "./3 - Simultaneous Perturbation"
              dockerfile: Dockerfile
          image: 3-simultaneous-perturbation:latest
          tty: true
          entrypoint:
            - python
            - change_pcap_timestamps.py

        4-learning-automata:
          build:
              context: "./4 - Learning Automata"
              dockerfile: Dockerfile
          image: 4-learning-automata:latest
          tty: true
          entrypoint:
            - python
            - la_slips.py
```

B Python2.7: change_pcap_timestamps.py

```
import time, sys, os, random
from scapy.all import *
from collections import deque
import perturbation_optimizer as po
import md5
from stf.core import dataset

import glob, os, re, subprocess, pickle

class Packet:

    def __init__(self, init_packet, packetNo):
        self.packetNo = packetNo
        self.packet = init_packet

class Flow:
    def __init__(self, init_packet):
        self.t1 = init_packet.packet.time
        self.t2 = None
        self.td = None
        self.length = len(init_packet.packet)
        self.src_ip = init_packet.packet[IP].src
        self.dst_ip = init_packet.packet[IP].dst
        self.src_port = init_packet.packet[IP].sport
        self.dst_port = init_packet.packet[IP].dport
        self.ToS = None
        self.packets = []
        self.packets.append(init_packet)
        self.td_list = []
        self.td_fractions = []
        self.time_until_next_flow = 0
        self.next_flow = None


    def get_duration(self):
        return self.td

```

```python
    def belongs_in_flow(self, packet):
        return (packet[IP].src == self.src_ip and packet[IP].dst
        ↪  == self.dst_ip and packet[IP].dport == self.dst_port
        ↪  and packet[IP].sport == self.src_port) or
        ↪  (packet[IP].dst == self.src_ip and packet[IP].src ==
        ↪  self.dst_ip and packet[IP].sport == self.dst_port)

    def add_to_flow(self, packet):
        self.packets.append(packet)
        self.length += len(packet.packet)

    def add_to_flow_not_IP(self, packet):
        self.packets.append(packet)
        self.length += len(packet.packet)

    def set_flow(self):
        self.t1 = self.packets[0].packet.time
        self.t2 = self.packets[-1].packet.time
        self.td = self.t2 - self.t1

    def same_ips(self, flow):
        return ((self.src_ip == flow.src_ip) and (self.dst_ip ==
        ↪  flow.dst_ip))

    def set_timedeltas(self):
        if len(self.packets) > 1 and self.td != None:
            counter = 0
            for packet in self.packets[1:]:
                self.td_list.append(packet.packet.time -
                ↪  self.packets[counter].packet.time)
                frac = self.td_list[counter]/self.td
                self.td_fractions.append(frac)
                counter +=1
        else:
            self.td = 0
            self.td_list.append(0)
            self.td_fractions.append(1)

    def perturb_duration(self, perturbation):

        if len(self.packets) == 1 or perturbation == 0:
            return 0

        counter = 1
        pert_sum = 0

        old = self.t2
        pert_seconds = perturbation - self.td
        for td_f in self.td_fractions:
            print td_f
            pert = pert_seconds * td_f
            pert_sum += pert
            if len(self.td_fractions) == counter:
                self.packets[counter].packet.time += pert_seconds
            else:
                self.packets[counter].packet.time += pert_sum
            counter +=1
        self.set_flow()
        return self.packets[-1].packet.time

    def incr_timestamps(self, pert_td_next):
        prev_time = self.packets[-1].packet.time
        for packet in self.packets:
            packet.packet.time += pert_td_next
        print "The last packet in this flow was changed from
         ↪  time: %f to %f" % (prev_time,
         ↪  self.packets[-1].packet.time)
        self.set_flow()
        return self.packets[-1].packet.time - prev_time

    def scew_flow_by(self, seconds):
        for packet in self.packets:
            packet.packet.time += seconds
        self.set_flow()

    def get_flow_size(self):
        return self.length

    def __str__(self):
        return str(self.src_ip) + "\t" + str(self.src_port) +
         ↪  "\t" + str(self.dst_ip) + "\t" +  str(self.dst_port)

def parse_pcap_to_deque(target_tuples, packets):
    flows = deque()
    counter = 0
    prev_flow = None
    print "Parsing scapy packet set..."
    for packet in packets[1:]:
        added_to_flow = False
        cur_packet = Packet(packet, counter)
        if IP in packet:
            if ((packet[IP].src, packet[IP].dst) or
             ↪  (packet[IP].dst, packet[IP].src)) in
             ↪  target_tuples:
                cur_flow = Flow(cur_packet)
                if len(flows) != 0:
                    for flow in flows:
                        if isinstance(flow, Flow) and not
                         ↪  added_to_flow:
                            if flow.belongs_in_flow(packet):
                                flow.add_to_flow(cur_packet)
                                added_to_flow = True
                                break
                    if not added_to_flow:
                        flows.append(cur_flow)
                else:
                    flows.append(cur_flow)
            else:
                flows.append(cur_packet)
        else:
            flows.append(cur_packet)
        counter +=1
    print "Added packet set to flows."
    print "Setting flow parameters..."
    for i in flows:
        if isinstance(i, Flow):
            i.set_flow()
            i.set_timedeltas()
    return flows

def get_timedelta_next_flow_in(deque, index, tuple, cur_flow):
    found = False
    counter = 1
    while not found:
        if index + counter >= len(deque):
            return None, 0
        next_flow = deque[index+counter]
        if isinstance(next_flow, Flow):
            if next_flow.src_ip == tuple[0] and next_flow.dst_ip
             ↪  == tuple[1] and (next_flow not in
             ↪  cur_flow.packets and
             ↪  next_flow.packets[0].packet.time >
             ↪  cur_flow.packets[0].packet.time):
                found = True
                return next_flow,
                 ↪  next_flow.packets[0].packet.time -
                 ↪  cur_flow.packets[0].packet.time
```

```python
        elif isinstance(next_flow, Packet):
            if (next_flow.packet.src == tuple[0] and next_flow.packet.dst == tuple[1]) and (next_flow not in cur_flow.packets and next_flow.packet.time > cur_flow.packets[0].packet.time):
                found = True
                return next_flow, next_flow.packet.time - cur_flow.packets[0].packet.time
        counter += 1
    return None, 0

def measure_time_between_flows(target_tuple, malware):
    flows = parse_pcap_to_deque(target_tuple, malware)
    counter = 0
    for flow in flows:
        if isinstance(flow, Flow):
            if(flow.src_ip, flow.dst_ip) in target_tuple:
                flow.next_flow, flow.time_until_next_flow = get_timedelta_next_flow_in(flows, counter, (flow.src_ip, flow.dst_ip), flow)
                counter +=1
    return flows


def write_flows_to_pcap_ordered_by_time(flows, outfile):
    tmp_flows_dict = {}
    for flow in flows:
        if isinstance(flow, Flow):
            for i in flow.packets:
                tmp_flows_dict[i.packet.time] = i.packet
        else:
            tmp_flows_dict[flow.packet.time] = flow.packet
    print "Finished indexing Packets. Sorting Packets and Flows..."
    od = collections.OrderedDict(sorted(tmp_flows_dict.items()))
    print "Finished sorting Packets and Flows. Writing pcap..."
    for k, v in od.iteritems():
        wrpcap("trials/" + outfile, v, append=True)


def write_flows_to_pcap_ordered_by_number(flows, outfile):
    tmp_flows_dict = {}

    for flow in flows:
        if isinstance(flow, Flow):
            for i in flow.packets:
                tmp_flows_dict[i.packetNo] = i.packet
        else:
            tmp_flows_dict[flow.packetNo] = flow.packet
    print "Finished indexing Packets. Sorting Packets and Flows..."
    od = collections.OrderedDict(sorted(tmp_flows_dict.items()))
    print "Finished sorting Packets and Flows. Writing pcap..."
    for k, v in od.iteritems():
        wrpcap("trials/" + outfile, v, append=True)



def redefine_stored_network_flows(target_tuples, evolving_params, iterationNo, flows):
    outfile = "2015-03-09_capture-win7-first1k-" + str(iterationNo) + ".pcap"
    total_bytes = 0.0
    total_duration = 0.0
    total_duration_perturbation = 0.0
    pert_incr = 0
    pert_data = {}
    optimizer = po.PerturbationOptimizer()

    print "Altering duration of packets..."
    counter = 0
    for flow in flows:
        original_timestamp = 0
        if not isinstance(flow, Flow):
            original_timestamp = flow.packet.time
        else:
            original_timestamp = flow.packets[-1].packet.time
        if isinstance(flow, Flow):
            if (flow.src_ip, flow.dst_ip) in target_tuples:
                positive = True
                td_next = flow.time_until_next_flow
                pre_pert_tu = flow.time_until_next_flow
                pre_pert_td = flow.td
                pre_perturbation = flow.packets[-1].packet.time
                total_loss = 0
                pert, loss = po.run_spsa([flow.length, flow.td, td_next], beta_vector=evolving_params)
                #omit negative perturbations
                positive = all( i >= 0 for i in pert)
                while not positive:
                    pert, loss = po.run_spsa([flow.length,
                        ↪  flow.td, td_next],
                        ↪  beta_vector=evolving_params)
                    pert_incr += 1
                    positive = all( i >= 0 for i in pert )
                total_bytes+=pert[0]

                 ↪  total_duration_perturbation+=(pert[1]-flow.td)+(pert[2]-td_next)
                pert_td_next = pert[2]
                pert_td = pert[2]
                flow_incr = 0
                if len(flow.packets) > 1:
                    # We may perturb duration
                    if flow.next_flow != None:
                        flow_incr =
                         ↪  flow.next_flow.incr_timestamps(pert_td-td_next)
                        total_duration += flow_incr
                    new_timestamp =
                     ↪  flow.perturb_duration(pert[1])
                    pert_data[counter] = [counter,
                        ↪  pert_td_next-td_next, pre_perturbation,
                        ↪  new_timestamp, flow_incr, pert,
                        ↪  [flow.length, pre_pert_td, pre_pert_tu]]
                    total_duration += flow.td
                else:
                    # We may not pertub duration, only
                        ↪  time_until_next_flow and optionally byte
                        ↪  size
                    if flow.next_flow != None:
                        flow_incr =
                         ↪  flow.next_flow.incr_timestamps(pert_td-td_next)
                        total_duration += flow_incr
                        total_loss += loss
                    new_timestamp = pre_perturbation
                    pert_data[counter] = [counter,
                        ↪  pert_td_next-td_next, pre_perturbation,
                        ↪  new_timestamp, flow_incr, pert,
                        ↪  [flow.length, pre_pert_td, pre_pert_tu]]
                cur_packet = flow.packets[-1]
                counter += 1
            else:
                cur_packet = flow.packets[-1]
                pert_data[counter] = [counter, 0,
                    ↪  cur_packet.packet.time,
                    ↪  cur_packet.packet.time,
                    ↪  cur_packet.packet.time, [0,0,0], [0, 0, 0]]
                counter += 1
        else:
            cur_packet = flow
            pert_data[counter] = [counter, 0,
             ↪  cur_packet.packet.time, cur_packet.packet.time,
             ↪  cur_packet.packet.time, [0,0,0], [0, 0, 0]]
            counter += 1



    write_flows_to_pcap_ordered_by_time(flows, outfile)
    print "Generating binetflow file.."
    current_datasets = dataset.Datasets()

     ↪  current_datasets.create("/home/ubuntu/SlipsExperiments/trials/"
     ↪  + outfile)
    current_datasets.generate_argus_files()
    print "Binetflow file generated."
    throughput = 0.00001
    if total_bytes > 0 and total_duration > 0:
        throughput = total_bytes/(total_duration)
    print total_bytes, total_duration
    #Total throughput for the modified flows
    return throughput, [total_bytes, total_duration_perturbation,
     ↪  total_duration, pert_data, evolving_params], total_loss


def test_binetflow_using_slips():
    list_of_files = glob.glob("trials/*.binetflow")
    latest_file = max(list_of_files, os.path.getctime)
    try:
        return subprocess.call("python
         ↪  /home/ubuntu/StratosphereLinuxIPS/slips.py -f
         ↪  /home/ubuntu/StratosphereLinuxIPS/models/ -d -r
         ↪  /home/ubuntu/SlipsExperiments/" + latest_file[0],
         ↪  shell=True)
    except Exception as e:
        return None

def was_feasible():
    list_of_files =
     ↪  glob.glob("/home/ubuntu/SlipsExperiments/logs/*")
    latest_file = max(list_of_files, os.path.getctime)
    with open(latest_file[0], "r") as f:
        line = f.read()
        if "detected as malicious" in line:
            theline = line
    alert = theline.split("\n")[-1]
    m = re.match(r'\d+', alert)
    det = m.group(0)
    print det
    if det == '0':
        return True
    return False
```

C Python2.7: perturbation_optimizer.py

```
import numpy as np
import random
from collections import deque
from itertools import islice, izip, tee, count


class PerturbationOptimizer:

    def __init__(self,
        problem_size = 2,
        max_iter = 40,
        bounds = [[300.0, 1200.0], [1000.0, 2000.0], [5000.0,
         ↪  12000.0]],
        init_factor_bytes = 0.1,
        init_factor_duration = 0.1,
        init_factor_offline = 0.9,
        s_factor = 1.2,
        l_factor = 3.0,
        iter_mult = 10,
        max_no_impr = 10):

        self.problem_size = problem_size
        self.bounds = bounds
        self.max_iter = max_iter
        self.init_factor_bytes = init_factor_bytes
        self.init_factor_duration = init_factor_duration
        self.init_factor_offline = init_factor_offline
        self.s_factor = s_factor
        self.l_factor = l_factor
        self.iter_mult = iter_mult
        self.max_no_impr = max_no_impr

```

```python


    def run_spsa(self, vector=[204, 7.9, 50.1]):
        run_spsa(vector)

    def random_vector_with_thresholds(self, thresholds):
        return [
        (random.uniform(thresholds[0][0], thresholds[0][1])),
        (random.uniform(thresholds[1][0], thresholds[1][1])),
        (random.uniform(thresholds[2][0], thresholds[2][1]))
        ]

    def add_solutions_to_deque(self, amount):
        for i in range(amount):
            self.best_params.append(self.adaptive_random_search(
                self.max_iter,
                self.bounds,
                self.init_factor_bytes,
                self.init_factor_duration,
                self.init_factor_offline,
                self.s_factor,
                self.l_factor,
                self.iter_mult,
                self.max_no_impr
            ))

    def objective_function(self, vector, pert_vector=None):
        return (vector[2] + vector[1])/float(vector[0])

    def random_vector(self, minmax):
        """
        Returns a random 2D vector that represents a step in a random direction, based on thresholds.
        """
        return [self.rand_in_bounds(minmax[i][0], minmax[i][1]) for i in range(len(minmax))]


    def rand_in_bounds(self, min, max):
        return min + ((max-min) * random.uniform(0, 1))

    def large_step_size(self, iter, step_size, s_factor, l_factor, iter_mult):
        if iter > 0 and iter % iter_mult == 0:
            return [x * l_factor for x in step_size]
        return [x * s_factor for x in step_size]

    def take_steps(self, bounds, current, step_size, big_stepsize):
        step, big_step = {}, {}
        step["vector"] = self.take_step(bounds, current["vector"], step_size)
        step["cost"] = self.objective_function(step["vector"])
        big_step["vector"] = self.take_step(bounds, current["vector"], big_stepsize)
        big_step["cost"] = self.objective_function(big_step["vector"])

        return step, big_step

    def take_step(self, minmax, current, step_size):
        position = current
        for i in range(len(position)):
            min_i = max(minmax[i][0], current[i]-step_size[i])
            max_i = min(minmax[i][1], current[i]+step_size[i])
            position[i] = self.rand_in_bounds(min_i, max_i)
        return position

    def adaptive_random_search(self,
        max_iter=100,
        bounds=[[300, 400], [1.0, 2.0], [5.0, 12.0]],
        init_factor_bytes=0.1,
        init_factor_duration=0.1,
        init_factor_offline=0.9,
        s_factor=1.2,
        l_factor=3.0,
        iter_mult=10,
        max_no_impr=10):
        """
        Description:
            Init function for the Adaptive Random Search algorithm

        Args:
            max_iter: the maximum amount of iterations one session will conduct.
            bounds: 2D list representing the bounds for each parameter on the form [[min_a, max_a], [min_b, max_b], [min_c, max_c]]
            init_factor_bytes: initialization factor for maximum amount of bytes.
            init_factor_duration: initialization factor for maximum duration of flow (ms)
            init_factor_offline: initialization factor for maximum offline duration (ms)
            s_factor: factor for the small step size
            l_factor: factor for the large step size
            iter_mult: how often step_size should be multiplied by l_factor. iteration % iter_mult == 0
            max_no_impr: how many attempts should be tried unsuccessully before reducing the step_size by s_factor fractions.
        """
        step_size = list()
        step_size.append((bounds[0][1]-bounds[0][0]) * init_factor_bytes)
        step_size.append((bounds[1][1] - bounds[1][0]) * init_factor_duration)
        step_size.append((bounds[2][1] - bounds[2][0]) * init_factor_offline)
        current, count = {}, 0
        current["vector"] = self.random_vector(bounds)
        current["cost"] = self.objective_function(current["vector"], step_size)

        #print current["cost"]
        #print current["vector"], current

        for iter in range(max_iter):
            big_stepsize = self.large_step_size(iter, step_size, s_factor, l_factor, iter_mult)
            step, big_step = self.take_steps(bounds, current, step_size, big_stepsize)
            #print step, big_step
            # step, big_step are dictionaries that contain the estimated cost for that step
            if step["cost"] <= current["cost"] or big_step["cost"] <= current["cost"]:
                if big_step["cost"] <= step["cost"]:
                    step_size, current = big_stepsize, big_step
                else:
                    current = step
                count = 0
            else:
                count += 1
                count = 0
                if count >= max_no_impr:
                        step_size = [x / s_factor for x in step_size]
                #print(" > iteration %d \t best=%f" % (iter+1,
                ↪  current["cost"]))
            return current


class Bernoulli:
    def __init__(self, r=1, p=3):
        """
        The bernoulli distribution of +/- 1 is the distribution
    ↪  we choose for our delta-k vector
        This is in order to meet the requirements for the
    ↪  algorithm as described in
    ↪  https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=705889,
        in that uniform or normal random variables are not
    ↪  allowed. Each variable should have 1/2 probability of
    ↪  occurrence.
        """
        self.p = p
        self.r = r

    def __call__(self):
        return [random.choice((-self.r, self.r)) for _ in
        ↪  range(self.p)]

def nth(iterable, n, default=None):
    """Returns the nth item or a default value"""
    return next(islice(iterable, n, None), default)


def y(theta):
    """The loss function that returns seconds per Byte sent.
    ↪  Adding noise to time_between_network_flows."""
    return ((theta[1] + theta[2]) / theta[0]) - ((theta[2] *
    ↪  0.01) + random.gauss(0, 0.1))

def identity(id):
    return id

def SPSA(y, t0, a, c, delta, constraint=identity):
    """
        Creates an Simultaneous Perturbation Stochastic
    ↪  Approximation iterator.
        y - a function of theta that returns a scalar
        t0 - the starting value of theta
        a - an iterable of a_k values
        c - an iterable of c_k values
        delta - a function of no parameters which creates the
↪  delta vector
        constraint - a function of theta that returns theta
        """
    theta = t0

    # Pull off the ak and ck values forever
    for ak, ck in izip(a, c):
        # Get estimated gradient
        gk = estimate_gk(y, theta, delta, ck)

        # Adjust theta estimate using SA
        theta = [t - ak * gkk for t, gkk in izip(theta, gk)]

        # Constrain
        theta = constraint(theta)

        yield theta # This makes this function become an iterator

def estimate_gk(y, theta, delta, ck):
    '''Helper function to estimate gradient approximation'''
    # Generate Delta vector
    delta_k = delta()
    # Get the two perturbed values of theta
    ta = [t + ck * dk for t, dk in izip(theta, delta_k)]
    #print "Perturbed t_a: "
    #print ta
    tb = [t - ck * dk for t, dk in izip(theta, delta_k)]
    #print "Perturbed t_b: "
    #print tb

        # Calculate g_k(theta_k)
    ya, yb = y(ta), y(tb)
    #print "Result of y(ta): %f \t Result of y(tb): %f" % (ya,
    ↪  yb)
    #print "Calculating G_k ..."
    gk = [(ya-yb) / (2*ck*dk) for dk in delta_k]

    return gk

def standard_ak(a, A, alpha, beta):
    '''Create a generator for values of a_k in the standard
    ↪  form.'''
    # Parentheses makes this an iterator comprehension
    # count() is an infinite iterator as 0, 1, 2, ...
```

```python
    return (((a / ((k+1)*.5) ** alpha) + beta*alpha) for k in
    ↪  count())

def standard_ck(c, gamma, beta):
        '''Create a generator for values of c_k in the standard
        ↪  form.'''
        return (((c / ((k+1*0.5)) ** gamma) + beta) for k in
        ↪  count())


def run_spsa(init_theta, beta_vector=[0.1, 0.9], n=1000,
↪  replications=40):
    dim = 3
    theta0 = init_theta
    c = standard_ck(c=1, gamma=0.101, beta=beta_vector[0])
    a = standard_ak(a=1, A=100, alpha=0.602, beta=beta_vector[1])
    delta = Bernoulli(p=dim)

    # tee splits an iterator into n independent runs of that
    ↪  iterator
    # iterators let us create a "lazy" list that we can just pop
    ↪  values from.
    # this is a quite efficient way to do it
    ac = izip(tee(a,n),tee(c,n))

    losses = []
    for a, c in islice(ac, replications):
        theta_iter = SPSA(a=a, c=c, y=y, t0=theta0, delta=delta)
        terminal_theta = nth(theta_iter, n) # Get 1000th theta
        terminal_loss = y(terminal_theta)
        losses += [terminal_loss]
    return terminal_theta
```

D Python2.7: la_slips.py

```python
import change_pcap_timestamps as cpt
import random
import math
from datetime import datetime
import glob, os, re, subprocess, pickle
from scapy.all import *
import pandas as pd
import matplotlib.pyplot as plt
```

```
from matplotlib import style
import numpy as np
style.use('ggplot')

niterations = 100

target_tuples = [('10.0.2.107','212.59.117.207'),
↪  ('10.0.2.107','91.222.139.45')]

lmba=0.1

N_a = 100
min_a = 0.5
max_a = 1

N_b = 100
min_b = 0.05

max_b = 0.5

A = [min_a+i*1.0*(max_a-min_a) / N_a for i in range(N_a)]
B = [min_b+i*1.0*(max_b-min_b) / N_a for i in range(N_b)]

P_A = [0.5 for i in range(N_a)]
P_B = [0.5 for i in range(N_b)]


"""
def was_feasible():

    list_of_files =
↪  glob.glob("/home/ubuntu/SlipsExperiments/logs/*")
    latest_file = max(list_of_files, os.path.getctime)

    with open(latest_file[0], "r") as f:
        line = f.read()
        if "detected as malicious" in line:
            theline = line
    alert = theline.split("\n")[-1]
    m = re.match(r'\d+', alert)
    det = m.group(0)
    if det == '0':
        return True
    return False
"""
def throughput_function(a, b, iter, flows):

    tmp_flows = flows
    throughput, data =
     ↪  cpt.redefine_stored_network_flows(target_tuples, [b, a],
     ↪  iter, tmp_flows)
    res = cpt.test_binetflow_using_slips()
    data.append(throughput)
    if res == '0':
        print "Successfully tested binetflow using Slips...
         ↪  Checking if result is feasible"
        data.append(True)
    else:
        data.append(False)
    return throughput, cpt.was_feasible(), data

def roulette_selection(weights):
    totals = []

    running_total = 0

    for w in weights:
        running_total += w
        totals.append(running_total)

    rnd = (random.random() * running_total)

    for i, total in enumerate(totals):
        if rnd < total:
            return i

def init_la(n_iterations):


    best_throughput_so_far = 0

    best_index_a = 0
    best_index_b = 0

    allData = {}

    filename = "data/slips_data_" +
     ↪  datetime.now().strftime("%H_%M_%h_%m_%s") + ".dat"


    cridex =
     ↪  rdpcap("/home/ubuntu/SlipsExperiments/trials/2015-03-09_capture-win7-first1k.pcap")
    cridex_target_tuple = [('10.0.2.107','212.59.117.207'),
    ↪  ('10.0.2.107','91.222.139.45'), ('91.222.139.45',
    ↪  '10.0.2.107'), ('212.59.117.207', '10.0.2.207')]
    flows = cpt.measure_time_between_flows(cridex_target_tuple,
    ↪  cridex)

    allData["iteration_number"] = []
    allData["feasible"] = []
    allData["throughput"] = []
    allData["total_bytes"] = []
    allData["total_duration_perturbation"] = []
    allData["pert_data"] = []
    allData["total_duration"] = []
    allData["evolving_params"] = []
    allData["best_throughput_so_far"] = []
    allData["current throughput"] = []
    allData["best_index_a"] = []
    allData["best_index_b"] = []
    allData["improvement"] = []

    tmp_flows = flows

    for iter in range(n_iterations):
        # Pick index for the value a according to probability
        ↪  vector P_A
        index_a = roulette_selection(P_A)
        # Pick index for the value b according to probability
        ↪  vector P_B
        index_b = roulette_selection(P_B)

        improvement = False
        feasible = True

        current_throughput, feasible, data =
        ↪  throughput_function(A[index_a], B[index_b], iter,
        ↪  flows)

        if current_throughput > best_throughput_so_far and
        ↪  feasible == True:
            print "The throughput was nice, and we were not
            ↪  detected!"
            best_throughput_so_far = current_throughput
            improvement = True
            best_index_a = index_a
            best_index_b = index_b

```

```python
        for index in range(N_a):
            if index == best_index_a:
                P_A[index] = P_A[index] + lmba*(1-P_A[index])
            else:
                P_A[index] = P_A[index] + lmba*(0-P_A[index])

        for index in range(N_b):
            if index == best_index_b:
                P_B[index] = P_B[index] + lmba*(1-P_B[index])
            else:
                P_B[index] = P_B[index] + lmba*(0-P_B[index])

        if iter % 10 == 0:
            print "---"*10
            print "---"*10
            print "Probablity for choice of A", P_A
            print "Probablity for choice of B", P_B
            print "Best so far", best_throughput_so_far

        allData["iteration_number"].append(iter)
        allData["feasible"].append(feasible)
        allData["throughput"].append(current_throughput)
        allData["improvement"].append(improvement)
        allData["total_bytes"].append(data[0])
        allData["total_duration_perturbation"].append(data[1])
        allData["pert_data"].append(data[3])
        allData["total_duration"].append(data[2])
        allData["evolving_params"].append(data[4])
        allData["best_throughput_so_far"].append(best_throughput_so_far)
        allData["current throughput"].append(current_throughput)
        allData["best_index_a"].append(best_index_a)
        allData["best_index_b"].append(best_index_b)

        flows = tmp_flows

    stats = pd.DataFrame(allData)
    iter_stats = stats.set_index('iteration_number')

    stats.to_pickle("data/pandas-dataset-" + datetime.now().strftime("%H_%M_%h_%m_%s") + ".pk1")
    with open(filename, 'wb') as output:
        pickle.dump(allData, output, pickle.HIGHEST_PROTOCOL)

    print "execution finished"
    print "The perturbed pcap-file was feasible:",
    ↪  allData["feasible"][-1]
    print "Final Probablity for choice of A", P_A
    print "Final Probablity for choice of B", P_B
    print "---"*10
    print "---"*10
    print "best index of A value ", best_index_a," which
    ↪  corresponds to", A[best_index_a]
    print "best index of B value ", best_index_b," which
    ↪  corresponds to", B[best_index_b]
    print "---"*10
    print "Given these two best values, the Optimal found
    ↪  throughput (max in all iterations so far) %f Bytes per
    ↪  second" % (best_throughput_so_far)

init_la(niterations)
```

References

[1] Maria Rigaki and Sebastian Garcia. Bringing a gan to a knife-fight: Adapting malware communication to avoid detection. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 70–75. IEEE, 2018.

[2] Cisco Systems Inc. Cisco 2018 Annual Cyber Security Report. `https://www.cisco.com/c/en/us/products/security/security-reports.html`, 2018. [PDF; accessed 17-September-2018].

[3] McAfee. Navigating a Cloud Sky. `https://www.mcafee.com/enterprise/en-us/assets/reports/restricted/rp-navigating-cloudy-sky.pdf`, 2018. [PDF; accessed 11-October-2018].

[4] Ralf Benzmüller. Malware numbers 2017. `https://www.gdatasoftware.com/blog/2018/03/30610-malware-number-2017`, 2017. [Online; accessed 17-September-2018].

[5] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[6] Ibrahim Ghafir, Jakub Svoboda, and Vaclav Prenosil. A survey on botnet command and control traffic detection. 5:75–80, 10 2015.

[7] Ahmad Karim, Rosli Bin Salleh, Muhammad Shiraz, Syed Adeel Ali Shah, Irfan Awan, and Nor Badrul Anuar. Botnet detection techniques: review, future trends, and issues. *Journal of Zhejiang University SCIENCE C*, 15(11):943–983, Nov 2014.

[8] Github: jgamblin. Mirai source code, 2017.

[9] Technologyreview.com. Cybersecurity experts uncover dormant botnet of 350,000 twitter accounts, 2017.

[10] H. R. Zeidanloo, Mohammad Jorjor Zadeh Shooshtari, Payam Vahdani Amoli, M. Safari, and M. Zamani. A taxonomy of botnet detection techniques. In *2010 3rd International Conference on Computer Science and Information Technology*, volume 2, pages 158–162, July 2010.

[11] Garcia Sebastian. Botnets behavioral patterns in the network, 2014.

[12] Sebastian Garcia and Michal Pechoucek. Detecting the behavioral relationships of malware connections. In *Proceedings of the 1st International Workshop on AI for Privacy and Security*, page 8. ACM, 2016.

[13] EY. Ey 20th global information security survey, 2018.

[14] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. Evading machine learning malware detection. *Black Hat*, 2017.

[15] J. Kargaard, T. Drange, A. Kor, H. Twafik, and E. Butterfield. Defending it systems against intelligent malware. In *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, pages 411–417, May 2018.

[16] Chirag Modi, Dhiren Patel, Bhavesh Borisaniya, Hiren Patel, Avi Patel, and Muttukrishnan Rajarajan. A survey of intrusion detection techniques in cloud. *Journal of network and computer applications*, 36(1):42–57, 2013.

[17] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31, 2016.

[18] Ahmad Javaid, Quamar Niyaz, Weiqing Sun, and Mansoor Alam. A deep learning approach for network intrusion detection system. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (Formerly BIONETICS)*, BICT'15, pages 21–26, ICST, Brussels, Belgium, Belgium, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[19] Wikipedia contributors. Feature selection — Wikipedia, the free encyclopedia, 2018. [Online; accessed 24-September-2018].

[20] Wikipedia. Markov chains, 2018.

[21] Stratosphere IPS. Stf, 2018.

[22] Stratosphere IPS. Zbot-pcap, 2018.

[23] Stratosphere IPS. Stf technology, 2018.

[24] clerveralgorithms.com. Random search, 2015.

[25] clerveralgorithms.com. Adaptive step-size random search, 2015.

[26] Herbert Robbins and Sutton Monro. A stochastic approximation method. In *Herbert Robbins Selected Papers*, pages 102–109. Springer, 1985.

[27] Colombia University Lauren A. Hannah. Stochastic optimization, 2014.

[28] J. C. Spall. Implementation of the simultaneous perturbation algorithm for stochastic optimization. *IEEE Transactions on Aerospace and Electronic Systems*, 34(3):817–823, July 1998.

[29] James C Spall et al. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE transactions on automatic control*, 37(3):332–341, 1992.

[30] Hamid Beigy and Mohammad Reza Meybodi. Utilizing distributed learning automata to solve stochastic shortest path problems. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 14(05):591–615, 2006.

[31] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[32] Jernej Kos, Ian Fischer, and Dawn Song. Adversarial examples for generative models. *arXiv preprint arXiv:1702.06832*, 2017.

[33] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[34] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 5, 2011.

[35] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3730–3738, 2015.

[36] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick D. McDaniel. Adversarial perturbations against deep neural networks for malware classification. *CoRR*, abs/1606.04435, 2016.

[37] Daniel Lowd and Christopher Meek. Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 641–647. ACM, 2005.

[38] Roberto Perdisci, Guofei Gu, and Wenke Lee. Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems. In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 488–498. IEEE, 2006.

[39] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 372–387. IEEE, 2016.

[40] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[41] Xiaoyong Yuan, Pan He, Qile Zhu, Rajendra Rana Bhat, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *arXiv preprint arXiv:1712.07107*, 2017.

[42] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on GAN. *CoRR*, abs/1702.05983, 2017.

[43] Wenqi Wei, Ling Liu, Stacey Truex, Lei Yu, and Mehmet Emre Gursoy. Adversarial examples in deep learning: Characterization and divergence. *arXiv preprint arXiv:1807.00051*, 2018.

[44] Sebastian Garcia. Modelling the network behaviour of malware to block malicious patterns. the stratosphere project: a behavioural ips. *Virus Bulletin, number September*, pages 1–8, 2015.

[45] Wikipedia. Osi model, 2018.

[46] yantan16. Simultaneous perturbation stochastic approximation using iterators, 2018.

[47] Red Team Security Consulting. Red teaming methodology, 2018.

[48] Docker Inc. Docker compose, 2018.

[49] Torgeir Fladby. Discord malware, 2018.

[50] Torgeir Fladby. C2-stratosphere-evasion, 2018.

[51] NSM. Nsm-wiki, 2018.

[52] Daniel Guerra. Ubuntu-xrdp, 2018.

[53] Stratosphere IPS. Malware capture facility project, 2018.

[54] Stratosphere IPS. Cridex dataset, 2018.

[55] Symantec. Trojan.zbot, 2010.

[56] Symantec. Trojan.cridex, 2010.

[57] Wikipedia. Netflows, 2018.