SUPERFLUIDITY

A SUPER-FLUID, CLOUD-NATIVE, CONVERGED EDGE SYSTEM

RESEARCH AND INNOVATION ACTION GA 671566

DELIVERABLE I6.3:

MODELLING AND DESIGN FOR SYMBOLIC EXECUTION AND MONITORING TOOLS

DELIVERABLE TYPE:	R/OTHER
DISSEMINATION LEVEL:	PU
CONTRACTUAL DATE OF DELIVERY TO THE EU:	01.07.2016
ACTUAL DATE OF DELIVERY TO THE EU:	11.07.2016
WORKPACKAGE CONTRIBUTING TO THE DELIVERABLE:	WP6

EDITOR(S):	COSTIN RAICIU (UPB)
AUTHOR(S):	COSTIN RAICIU (UPB), RADU STOENESCU (UPB), MATEI POPOVICI (UPB), OMER GUREWITZ (BGU)
INTERNAL REVIEWER(S)	

ABSTRACT:	
KEYWORD LIST:	

INDEX

Superfluidity H2020
I6.3: Modelling and Design for Symbolic Execution and Monitoring Tools

Costin Raiciu, Matei Popovici,
Radu Stoenescu - University Politehnica of Bucharest
Omer Gurewitz, Asaf Cohen - Ben Gurion University

July 1, 2016

1 Introduction

DOW Description: This internal deliverable will include the modelling and design for Task 6.3: the symbolic execution checking tool and the automatic monitoring and anomaly detection.

5G networks will accentuate the growing industry trend to shift away from static policies and hardware implementations to dynamic instantiation of software network functionality. Software networking functionality could be requested by the operator itself or by paying third-parties such as web content proiders and mobile applications.

Superfluidity proposes the concept of a Reusable Functional Block as the API for 5G networking. Reusable Functional Blocks have a specification described in a higher level language that allows their users to compose them correctly, and can be implemented in multiple ways as long as they obbey their specification: software (as a monolithic block), hardware (ASICs), or a composition of other RFBs. How can we secure the resulting 5G networks?

Enforcing network security has two major phases: the network operator specifies higher-level policies and then implements them using (low-level) networking functionality typically provided by third-party vendors. High-level policies could include access control lists (who can talk to whom), firewall rules, routing protocol configurations, and so forth. Networking functionality includes switches, routers, middleboxes, etc. In traditional networks, high-level policies are fairly static and thus easy to manually check and deploy infrequently. Traditional networking hardware (switches, routers, simple firewalls) processes packets on custom-made ASICs that are thoroughly verified and seldom updated; such implementations give a fairly strong low-level security guarantee.

In 5G networks, network functions will be instantiated dynamically, and the network will run services configured not only by the operator, but also by third parties. Running third-party processing will be a major revenue source for operators and is

thus very attractive; however it can subject operator networks to many security risks that must be addressed. The great benefit of software network functions is that it can be easily upgraded and it avoids vendor lock-in. On the downside, diverse software with increasingly complex functionality and developed by many vendors is much more susceptible to low level exploits such as buffer overflows.

To enforce security in 5G networks SUPERFLUIDITY takes the three main directions:

1. **Describe operator policies in a high-level specification language**.

2. **Describe RFBs in a way that is amenable to static analysis**.Current informal specifications are not enough, and more precise descriptions are needed. Such descriptions could range from specifying the types of packets expected between boxes to a more complete specification in specialized languages such as SEFL [32] or NoD [17]. The language depends on the analysis we plan to perform: SEFL allows symbolic execution and is quite powerful; P4 [3] allows one to express expected header types using a finite-state-machine abstraction which makes it an ideal candidate if type safety is the only thing that is verified.

3. **Perform static analysis of RFB configurations to ensure policy is obbeyed before deployment**. We will use static analysis tools to analyze network processing described as a graph of RFBs and check whether it matches the high level policies. Existing tools include Header Space Analysis [13], Network Optimized Datalog [17] or our own SymNet tool [32]. SymNet is the most powerful tool in terms of properties verified, and is easiest to use because SEFL, its associated modeling language, is imperative and simple.

   SymNet can run reachability checks over network models by injecting symbolic packets and tracking their path through the network. The output is a list of paths explored in json format, and for each path the list of boxes visited, instructions executed and constrained applied to each header field (including exact values if that is the case). Using this output we can verify properties such as reachability, loop detection, header field changes, header visibility, and so forth. The operator policies can be translated into constraints on the output of SymNet, and verified (this is future work that needs to be done).

4. **Ensure that the implementation conforms to the specification at deployment time.** There is a gap between the abstract model of the network that we can statically analyze (steps above) and the actual implementation. This gap appears because it is impossible to verify large C implementations of networking code in useful time [9].

   Even if we assume the implementations of networking functionality are correct, similar gap appears when multiple abstractions are applied in a network, for instance network virtualization. We give a thorough example in this document based on OpenStack Neutron, where the tenant API static analysis is fairly simple but the instantiation may be faulty because the OpenStack drivers—control plane software that instantiates the tenant networking configuration—are faulty.

To bridge this gap, we rely on multiple techniques, some of which are known and tested but have limited coverage, and some which are novel and are still subject of ongoing work. We list them here:

- **Active testing.** This is the obvious solution, and it can be guided from the results of static analysis as proposed by [39] and also implemented in SymNet. Intuitively, once the static analysis results are known, packets are generated for each path resulting from static analysis and injected in the real network; the outcome is then checked to see if it matches that predicted by static analysis. As all other active techniques, this is lightweight and useful (it helped us uncover many bugs in our SEFL models), but it is not sufficient on its own because it has poor coverage and cannot give any type of strong guarantees.
- **Monitoring and anomaly detection.** This is another runtime technique that applies machine learning to understand the standard behaviour of the network and detect attacks when the behaviour deviates from standard. A snapshot of Superfluidity work in this space is given in Section 2.
- **Static analysis at lower layers.** Symbolic execution can be run on the model, and on the lower level implementation of the model; : if the resulting outputs are equivalent, then the implementation is correct. Defining equivalence is not easy, and different definitions capture different parts of the problem we want to solve—an initial exploration is provided in section 4.

  In certain cases the lower level implementation is C code, which means we will need to resort to traditional symbolic execution (e.g. Klee [5]); for simple scenarios this will work, but it will not scale for complex pieces of networking. In other cases, we can also model the lower level implementation in SEFL: our OpenStack work presented in section 4, the results of running static analysis on the resulting data plane after a tenant's configuration has been instantiated can be compared to the analysis of the abstract tenant view.
- **Automatic generation of C implementations from SEFL code.** SEFL code is memory-safe and easy to symbolically execute which allows us to prove it satisfies the high level properties of the operator. Is it possible to generate C code from these models? As SEFL is imperative, it is fairly easy to translate it to C code, and possibly manually audit the code blocks used in generation to ensure safety.

  However, SEFL models that are optimized for symbolic execution have low branching factor and large constraint sets while fast networking code requires high branching factor and low constraint sets per branch (see examples from [32] and [35] for more insights into why this is the case). Is it possible to *transform* SEFL models optimized for symbolic execution into ones optimized for actual deployment? Our experience in modeling for SymNet shows a few manual techniques can be applied, but we have started to analyze automated ways to do so. A brief snapshot of this incip-

ient work that will be presented in the NetPL Sigcomm workshop can be found in 5.

2 Monitoring and detection

Network monitoring and intrusion detection are basic, indispensable tools for operators. In 5G networks their importance will only grow because of the increasing complexity and an never-ending stream of network configuration changes (i.e. NFV function instantiation and removal).

A crucial part of this work is accurate traffic classification: deciding what application is generating a given packet or set of packets.

2.1 Light-Weight Traffic Classification for the NFV platform

Network traffic classification is vital for many network management tasks such as traffic design, bandwidth allotment, accounting, security (e.g., filtering and anomaly detection), QoS and policy enforcement, etc. Obviously identifying applications is becoming harder and harder to attain as more and more applications such as video/audio streaming, social networking, and gaming are moving to the cloud, especially since many of these applications are using encrypted communication. All the more so, labeling applications in Network Functions Virtualization (NFV) platforms, where network services are running on independent hardware, while the resources are shared between many different network services, makes monitoring even more challenging, especially when enormous data volumes are traversing and processing capabilities are shared between vast number of services. Identifying applications based on passive observations of individual or streams of packets traversing the network, typically relies on two main costly procedures: *capturing and classification*.

The *capturing* procedure intercepts data packets traversing the network and stores them for further inspection. Typically, network administrators utilize monitoring tools, such as NetFlow, sFlow, IPFIX and various other packet level capturing tools to collect flow information and export it for further analysis. Nevertheless, utilizing such tools, which typically rely on substantial packet capturing, consumes precious resources, and hence hinders the scalability of the NFV. Specifically, data centers and NFV networks involve enormous data volumes, thus, the resources required for the monitoring are overwhelmed as the number of flows and link capacities grow. Moreover, utilizing complex lookup tables for finding rule matches limits the performance even further. To address this issue, statistical sampling strategies are used, which lower the CPU and storage requirements. The sampled data is gathered into flows according to predefined criteria (e.g., 5-tuple, QoS marks, VLAN (inner, outer or both), MPLS labels, etc.), on which classification is performed.

However, due to the non-uniform (typically heavy-tailed) distribution of IP flows for packets and bytes, utilizing uniform sampling techniques can result in inadequate traffic estimation. Hence, studies have suggested to utilize more sophisticated sampling techniques. Yet, these solutions require complex mechanisms and *many lookup table filtering rules* in order to obtain a representative sample set. Such mechanisms are

highly expensive and consume a lot of time and computing resources, thus, it is a great challenge to minimize the required filtering ruleset.

The *traffic classification* process labels traffic based on a predefined set of classes (e.g., applications). A commonly used classification approach is Deep Packet Inspection (DPI), which classifies traffic by inspecting the *payload* of each packet traversing the inspection point. DPI attains remarkably high accuracy for unencrypted traffic. Nonetheless, since DPI relies on the visibility of the payload to the classifier, its effectiveness diminishes with the usage of data encryption, which is the conventional wisdom, or when examining the content of the packets traversing the network is forbidden or limited due to privacy or complexity concerns. Furthermore, DPI requires knowledge of the latest syntax each application exploits in its payload, which imposes high storage and computational resources, further hindering the utilization of DPI. Consequently, utilizing Machine-Learning (ML) techniques for traffic classification, which overcome most of the aforementioned hindering factors (with an appropriate *feature set*) is an attractive solution, especially in the NVF environment.

Typically, ML involves three main steps. First, in the feature selection phase, flow statistical attributes, such as minimum, maximum or average packet size, flow duration, or inter-arrival time, by which previously unseen flows will be classified, are selected. Then, in the learning phase, the classifier is trained to map the attribute set to classes, according to the selected learning method. Finally, in the actual testing phase, unknown traffic can be classified, utilizing the rules learned. Note that every classification method may apply different priorities to different attributes, leading to different training results, hence, different performance in the testing phase.

In this study, we suggest a novel set of features, which relies on an extremely simple sampling strategy, namely, a single filtering rule to capture the required data and demands only sampling a minimal fraction of the traffic volume in Bytes (2-3%), yet results in very high classification accuracy. The suggested technique relies on the following key observation: since each application adheres to a proprietary standard message exchange, the data exchanged between two applications should follow a typical pattern which distinctively characterizes the application that generated it. Specifically, when examining the sequence of Application Protocol Data Units (APDU) exchanged between the entities running the application, one can identify an exclusive pattern which is typical to the associated application and is different from one application to the next.

For example, as illustrated in Figure 1, a typical client-server application will follow a typical sequential pattern of request and response APDUs, wherein each such client-server application is characterized by its unique control messages (application-level control messages), typical response APDUs (i.e., typical sizes and variances) and typical patterns of responses which are correlated with the application's proprietary requests.

Inspired by the aforementioned observations, one can utilize these distinct patterns within the network's premises to identify the generating application. Moreover, since the suggested system tracks the traffic generated at the Application layer, such application fingerprints are completely transparent to lower layers and especially transparent to side effects at these layers such as fragmentation, retransmissions, timing, congestion, packet loss, location, delays, etc. Figure 2 shows a sample of ten accumulated APDUs (a-APDU) exchanges for three different applications (SSH, RDP and eMule).

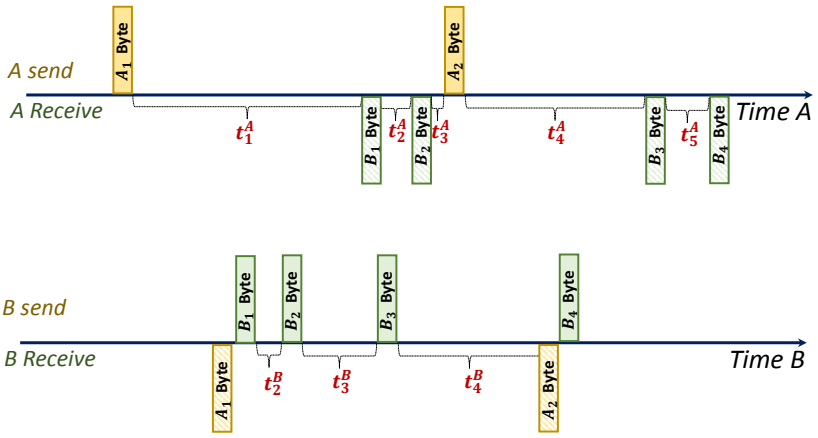

Figure 1: Illustrated example of data exchanged between two hosts

The graph depicts the interlaced number of Bytes sent by A (X-axis) vs. the number of Bytes sent by B (Y-axis) on a log-scale, based on captured TCP ACKs. For example, as can be seen in the figure, the difference between the first and second SSH entries indicates that in the first a-APDU A has sent around 1000 Bytes and B has sent around 500 Bytes; the difference between the second and third entries indicates that B has sent around 100 Bytes while A has sent no data (a-APDU (0,100)). Note that as stated earlier since each fingerprint entry only counts the accumulated number of bytes traversed throughout the network since the last change in direction, i.e., no time-stamps, number of exchanged packets or duplicates, our fingerprints are not sensitive to network noise (e.g., retransmissions, fragmentations, inter-arrival time, losses, etc.).

The challenge is hence, twofold. First, identifying the unique meta-data patterns (signatures) constructed by each application. Note that the application pattern recognition should be performed continuously in order to identify new applications and/or new legitimate patterns generated by already known applications. Furthermore, a specific application can generate several typical patterns, yet each pattern characterizes a specific application. Second, reconstructing the original APDU pattern, generated by the Application layer prior to the intervention of the lower layers, and especially, prior to the impact of network side effects. Note that this task can be done by acquiring all the log files or traces generated by each application, or by collecting and assembling all fragments generated by the two participants. However, since the classification is performed as a virtual network service, without relying on any collaboration from other virtual services or the hosts, and necessarily in real time, on an enormous number of applications traversing the network simultaneously, such solutions hinder the great potential of the NFV architecture.

We address both challenges by introducing a simple strategy that samples a minimal amount of traffic, yet attains a representative sample which enables characterization as well as reconstruction and classification of the unique APDU exchange between

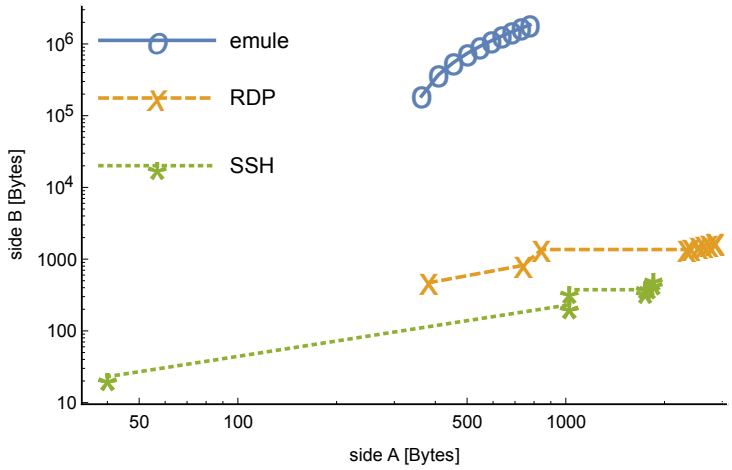

Figure 2: Ten a-APDU exchanges for three different applications: SSH, RDP and eMule. The axes depict the interlaced number of Bytes sent by A (X-axis) vs. the number of Bytes sent by B (Y-axis) in a flow, on a logarithmic scale.

each participating application entity. We devise a unique low-dimension attribute set, which allows the utilization of ML techniques to classify traffic while the flows are ongoing, with high accuracy for a large variety of applications. We show that our scheme requires very low sampling rate for TCP traffic and a slightly higher one for UDP traffic, yet provides very accurate traffic classification for both kinds of traffic.

Specifically, we extract TCP flow attributes only by sampling *zero-length packets*, i.e., packets that contain control bits, but do not contain any payload (e.g., SYN, ACK, etc.). Although zero-length packets are apparently frequent, we show experimentally that they comprise only 2-3% of the total traffic volume in bytes. Moreover, zero-length packets contain critical information on the connection state, yet, are easy to process, and more importantly, are resilient to the network parameters, such as congestion, fragmentation, delays, retransmissions, duplications and losses. Consequently, since each application behaves differently with respect to the amount of data it requires in order to work properly, it is likely to extract unique fingerprints that produce accurate classification for a large variety of applications. Leveraging the same approach, we extend our method to handle UDP flows as well; instead of inspecting the ACKs we inspect the UDP length field of each monitored flow.

In Figure 3 we depict the algorithm flowchart for TCP traffic. Specifically, upon a zero-length packet arrival of unseen flow, the collector creates a new a-APDU record and stores it in the database, where the flow's 4-tuple is used as a key. Each a-APDU record comprises the first and last ACK# and SEQ# which indicates on the a-APDU boundaries, and the direction of the APDU relative to the flow initiator, which is used for SEQ# and ACK# alignment. When a zero-length packet arrives, and its 4-tuple already exists in the collector's database, the collector first checks that the ACK# and SEQ# are relevant (e.g., this is not a retransmission, or out of order packet) then it

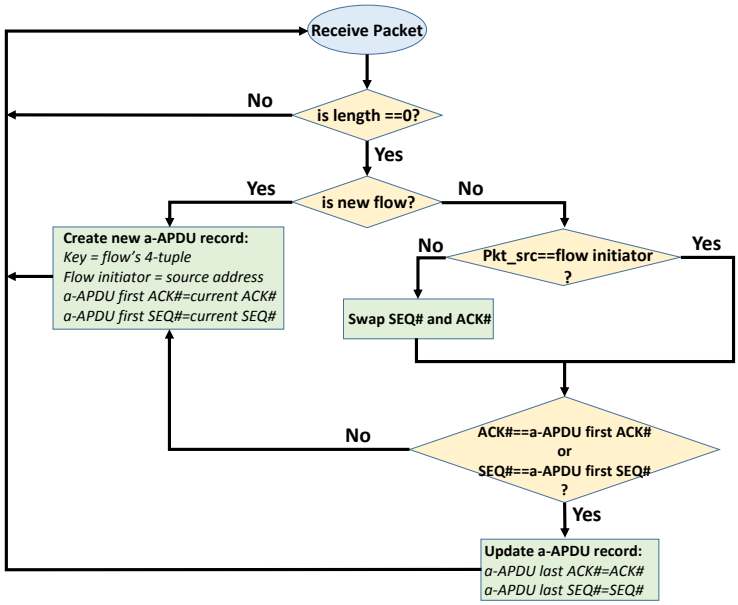

Figure 3: Algorithm flowchart.

checks if either the current packet SEQ# or ACK# is greater than the stored SEQ# or ACK#, respectively. Note that if only one of these fields grows and it is the same field as the last update for this particular flow, we need only to *update* the last ACK# and SEQ#, as no information passed from the other side during this period. Otherwise, the collector creates a new a-APDU record with first and last ACK# and SEQ# equal to the value of the packet's ACK# and SEQ#, respectively. Then, the collector adds the new a-APDU to the corresponding flow entry, and sends the updated a-APDU sequence for classification.

3 Detection of Anomalies in Virtual Network Functions

Cyber-attacks are a disturbing security threat in today's communication- and computer-based systems. They affect a wide range of domains, including electricity and water infrastructures, financial markets, medicine and healthcare, armies, enterprises and universities around the world.

Detecting and blocking such attacks, however, is becoming more and more challenging. While older computer viruses could be easily identified by locating known pieces of their code on the computer hard-drive or in email attachments, modern attacks are distributed [36], they utilize legitimate protocols and communication channels [33, 12, 8, 22, 11, 34, 4], and constantly change their code, servers, and attack strategies [31, 29].

As a result, the current literature includes numerous *anomaly detection techniques*,

which focus on detecting abnormal behaviour, rather than locating signatures of known attacks. Indeed, anomaly detection has been found useful in several cases, being able to detect *zero-day* attacks, previously unknown viruses and Trojans, and even unknown program behaviour on embedded devices [40].

Nevertheless, currently known detection techniques are still based on a few hindering assumptions, restricting our ability to cope with some of the current and future attacks. First, for successful anomaly detection, one must have a good model for the *normal data*. That is, when learning, for example, normal tenant behaviour, one has to correctly and robustly capture the essence of such behaviour, in order to identify deviations on the one hand, yet not to over-fit on the other, as this would result in high false alarm rates. Most of the known techniques today assume some statistical model for the data, and, in essence, estimate the parameters of such a model. A few examples include Markov [22] or ARMA models [6]. Simpler (only statistical-modeling wise) techniques base their detection on memoryless features, e.g., frequencies of events or proportions of packets of a given size, etc. For example, in [38], an Evil-Twin attack is successfully detected by identifying the average number of wireless hops. However, the key questions that arise are the following. What if, as detection systems designers, we *do not know* of any good statistical model for the normal data? Moreover, what if *there is no statistical model*? Are memoryless features, which disregard *the context* of the events in the system always sufficient? What if an anomalous behaviour can only be identified by the *order of the events*, and not necessarily by a single anomalous value of a certain feature?

A second important aspect is the detection complexity and the captures necessary for the anomaly detection system to perform properly. Managing a modern network, for example, requires employing numerous monitoring tools, and collecting huge amounts of data every second. Clearly, a deep inspection of all data is not feasible in most circumstances. What are, then, the main features of the data which can be *efficiently captured*, yet allow for good enough anomaly detection?

3.0.1 Anomaly Detection In SUPERFLUIFITY

To monitor system activity and alert in cases of mal-usage and malicious activity, SUPERFLUIDITY requires novel monitoring and detection tools. As mentioned above, such tools should be applicable without any prior knowledge on the normal behaviour of tenants in the system, and definitely without any assumption regarding the abnormal or malicious behaviour. Specifically, in SUPERFLUIDITY we will monitor derivatives of the tenants activities, such as memory requirements, processing time, traffic patterns and communication requirements, and use this data, without any prior model, to learn the normal activity of tenants and sub-systems. Note that this way monitoring can be done without sacrificing tenants privacy, and, in fact, will be possible even in cases of encrypted data or protected computation and storage. Then, using the learned data, activity in the system will be continuously compared to the normal structure. This way, the monitoring tools in SUPERFLUIDITY will be able to alert for abnormal behaviour. Moreover, changing trends in the system behaviour can be identified, allowing us to better prepare for times of higher demand, required re-allocation of resources and re-placements of services.

Therefore, in task 6.3, we developed a novel universal anomaly detection technique for VNFs, which does not require any a-priori information about neither the normal behaviour patterns nor the abnormal ones, yet efficiently learns the normal behaviour in order to generate a statistical model to which tested behaviour can be compared. The technique is based on the celebrated Lempel Ziv algorithm [41], the probability assignment induced by the prediction algorithm [10], and the learning technique we initiated in [30]. The technique inherits key useful aspects of the universal compression algorithm, that is, it performs optimally (in terms of estimating the model) even when there is no statistical model, and is extremely efficient to implement in practice. It offers a new look on the way to use data in the classification process, suggesting the *context* of the data sequence as the key characteristic used in the classification, rather than actual values. That is, the system does not rely on memoryless features of the data, such as specific times, sizes or other signatures. In contrast, it builds a *context tree* for the learned data. Then, when a new data sequence is tested, the order of values or events in it has the main impact on the classification performance.

3.1 Technical Preliminaries

Classification refers to the problem of labeling unknown (new) instances to the most appropriate class among a set of (known) predefined classes. When the underlying probability distributions for the classes $\{p_i(\cdot)\}_{i=1}^{M}$ are known, and we wish to decide which generated a given data sequence $\mathbf{y}$, a decision rule of the form

$$\hat{i} = \text{argmax}_{1 \leq i \leq M} p_i(\mathbf{y})$$

is optimal in the sense of minimizing the probability of error [16]. In unary-class classification, however, information is available only on one type of instances, namely, there is only one class, $p(\cdot)$. The goal, then, may be to either identify instances belonging to this class, or, taking the opposite viewpoint, usually referred to as *anomaly detection*, identify instances which *do not belong to the class*.

Specifically, assume for now a *given* probability distribution (of a single class) $p(\cdot)$. From this point on, we refer to this class as *normal*. In anomaly detection, the goal is then to identify whether a new data instance $\mathbf{y}$ belongs to the normal class, or, alternatively, is *anomalous*. Since, in most applications, the anomalous instances are threats one wishes to identify, we refer to a correct identification of an anomalous $\mathbf{y}$ as *detection* and for an incorrect identification of normal data as *false alarm*. The optimal decision rule in terms of maximizing the detection probability given a fixed false alarm probability (in the Neyman-Pearson sense) is to compare $p(\mathbf{y})$ to a *threshold*, and decide that $\mathbf{y}$ is normal if $p(\mathbf{y})$ is above the threshold and anomalous otherwise [20]. The threshold is determined according to the required false alarm probability.

In practice, the probability distributions governing the data (either multiple classes or a single one) are unknown, and there is only a limited amount of data to learn from. Furthermore, in most security-related applications, only few, if any at all, anomalous instances to learn from exist, yet more instances of normal behaviour are available. This asymmetry strengthens the need to take the anomaly detection approach in such circumstances, that is, build a behavioural model *only based on the normal instances*, and classify any instance deviating from that model as anomalous [7].

Thus, a reasonable approach is to estimate the probability distribution of normal data using the previously observed sequences and use the resulting estimate, $\hat{p}(\cdot)$, in the detection algorithm. Note, however, that the estimation problem differs significantly if a statistical model for the normal data is given, e.g., i.i.d. or Markovian of a certain order, in which case only a few parameters should be estimated; if, in a more complex scenario, the only knowledge is that the sequences are governed *by some* stationary and ergodic source; or, in the "worst" case, the data constitutes of *individual sequences*, that is, deterministic sequences with no pre-defined statistical model.

3.1.1 Universal Probability Assignment

The Lempel Ziv algorithm [41], LZ78, is a universal compression algorithm with a vanishing redundancy. Consequently, it can also be used as an optimal *universal prediction* algorithm [10], using the appropriate probability assignment. We briefly describe the compression method and the associated probability assignment algorithm.

The LZ78 algorithm is a dictionary-based compression method. For a given sequence of data symbols, a dictionary of phrases parsed from that sequence is constructed based on the incremental parsing process as follows. At the beginning the dictionary is empty. Then, during each step of the algorithm, the smallest prefix of consecutive data symbols not yet seen, i.e., which does not exist in the dictionary, is parsed and added to the dictionary. By that, each phrase is a unique phrase in the dictionary, that may extend a previously seen phrase by one symbol.

Given a sequence $s_1^n = (s_1 s_2 \dots s_n)$, a *parsed phrase*, P, is the smallest prefix of consecutive data symbols that has not been seen yet. This can also be considered as suffix concatenation of symbol s_i (from the sequence) with a previously seen phrase P' (from the dictionary), i.e., $P = (P' s_i)$. A *dictionary*, D, is a collection of all distinct phrases parsed from a given data sequences_1^n, i.e., $D = \{P_1, P_2, \dots, P_i, \dots P_n\}$. For example, the sequence $aabdbbacbbda$ is parsed as $a|ab|d|b|ba|c|bb|da|$.

A common representation of the dictionary is a rooted-tree, where each phrase in the dictionary is represented as a path from the root to an internal node in the tree according to the set of symbols the phrase consists of. In addition, leaf-nodes are added as suffix for each phrase in the tree. A statistical model can be defined for a given data sequences during the construction of a phrase-tree [10], as described next.

At the beginning, an initial tree is constructed including only a root node and k leaf-nodes as its children, where k is the size of the alphabet. Then, for each new phrase parsed from a sequence, the tree is traversed, starting from the root, following the set of symbols the phrase consists of, and ending at the appropriate leaf-node. Once a leaf-node is reached, the tree is extended at this point by adding all the symbols from the alphabet as immediate children nodes to that leaf, making it an internal node. In order to define a statistical model, each node in the tree, except for the root node, maintains a node traversal counter, where each leaf-node's counter is set to 1 and each internal node's counter is equal to the sum of its immediate children's counters.

For a probability assignment, as all leaf-nodes' counters are set to 1, they are assumed uniformly distributed with a probability $1/i$, where i is the total number of leaf-nodes. Each internal node's probability is defined as the sum of its immediate children's probabilities, which also equals the ratio between its counter and current i.

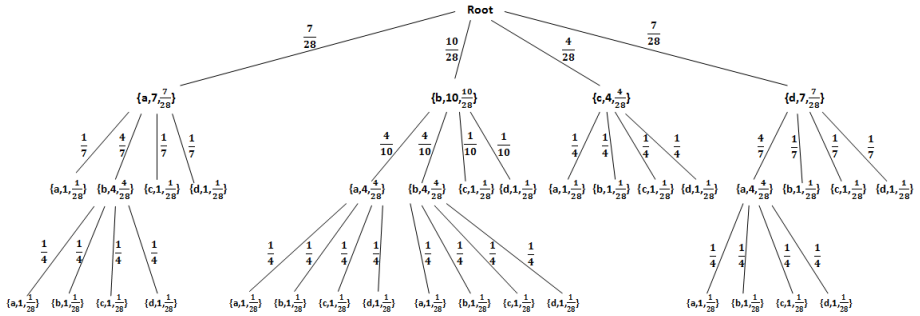

Figure 4: An LZ78 Statistical Model for sequence "aabdbbacbbda".

For example, Figure 4 demonstrates the resulting statistical model for the sequence "aabdbbacbbda". Each node in the tree is represented by the 3-tuple {symbol, counter, probability}. In addition, the probability of an edge is defined by dividing the nodes' probabilities. Note that the probabilities of edges connected directly to the root are equal to the appropriate root-children's counter divided by the total number of leaf-nodes, i, at each step of the algorithm. The probability of a phrase $P_i \in D$ is calculated by multiplying the probabilities of the edges along the path defined by the symbols of P_i. Moreover, note that for each phrase P_i, there exist a specific node in the tree whose probability represents the probability of that phrase. For instance, from the example shown in Figure 4, it can be seen that $P(ba) = \frac{10}{28} \times \frac{4}{10} = \frac{4}{28}$. Considering a sequence S, if during the traversal a leaf-node is reached before all the symbols of S are finished then the traversal return to the root and continue until all the symbols of that sequence are consumed [21]. For example, the probability of the sequence "bdca" given the same statistical model above, is defined as the following traversal probabilities multiplication: Root→b→d→Root→c→a and is calculated as:

$$P(bdca|M_{aabdbbacbbda}) = \frac{10}{28} \times \frac{1}{10} \times \frac{4}{28} \times \frac{1}{4} = \frac{1}{784}.$$

This stems from the conditional probability $\hat{P}(s_{t+1}|s_1^t)$, where s_{t+1} is the next symbol after the (sub-)sequence s_1^t, which is calculated as the ratio between the counter of symbol s_{t+1} and the counter of symbol s_t. We consider s_1^t as *the context* of s_{t+1} at time $t+1$.

3.2 Anomaly Detection Via Universal Probability Assignment

We now describe the building blocks of the anomaly detection system.

3.2.1 Preprocessing

A data sequence on which the anomaly detection algorithm operates is simply a sequence of values over some finite or infinite alphabet. Such a sequence may represent timing of events (e.g., times a certain service was requested), amounts of memory

required, or any other sequence of values. The strength of the algorithm is in its generality - the ability to adapt the algorithm to various data sequences, simply by changing the preprocessing stage. To keep this description in context, consider a certain service a tenant requests, e.g., networking, Input/Output or a usage of a certain piece of hardware. We will keep track of the *timings* of these requests.

The ith *event*, denoted by $e_{i,T}$, is defined by a tuple

$$e_{i,T} = (t_i, \ldots),$$

where t_i is the time the event occurred for tenant T, followed by maybe some additional data. A *flow*, denoted by f_T, is series of events for tenant T, sorted by their time of occurrence, t_i. That is,

$$f_T = \{e_{1,T}, e_{2,T}, \ldots, e_{n,T}\}.$$

For actual learning and testing, it is not required to use all features (fields) in the data. Good detection capabilities can be achieved even when focusing on a single feature. For example, timing data can be characterized by the *difference between two consecutive events* of the same flow, denoted by Time-Difference (TD) and defined by

$$TD_{i,T} = e_{i+1,T}(t_{i+1}) - e_{i,T}(t_i).$$

Consequently, a *single-feature data sequence* is a serialization of one of the features, e.g., with respect to Time-Difference, a sequence is defined as:

$$f_{T,TD} = \{e_{2,T}(t_2) - e_{1,T}(t_1),\ e_{3,T}(t_3) - e_{2,T}(t_2), \ldots, e_{n,T}(t_n) - e_{n-1,T}(t_{n-1})\}.$$

It is important to mention that the above procedure may result in a sequence over a *large alphabet*. For example, times may be given with a very high precision. Such a high alphabet size may significantly increase the learning complexity. Hence, to reduce the range of values, quantization should be performed. For k quantization levels, a set of k centroids $\{c_1, c_2, c_3, \ldots, c_k\}$, is used. The centroids are extracted from the available data during the training phase. Clearly, the number of centroids and the method for extracting them may affect the overall results.

Sequences of the above form, and $f_{T,TD}$ as an example (after quantization), *are the sequences we will use for both learning and, later on, detection*. While simple and one-dimensional, in the sense of tracking only a single feature, in this case, the time differences, these sequence are powerful as they capture *the context of the events for the tenant*. In other words, we will see that what matters the most is not necessarily a specific value of a feature, i.e., a single time difference being higher or lower, but, rather, the sequence of values and their relations. For example, when a tenant performs a certain computation, its memory access pattern might have a certain structure. This structure encompasses the relations between the time it takes to compute something and the time it takes to access memory. However, when a tenant is accessing memory only to identify cache misses, and learn about the memory usage of another tenant, its memory access pattern can be completely different.

Next, based on the sequences above, we describe the learning phase, where a model for the normal behaviour is build, and the testing phase, where *new sequences* are tested against this model in order to decide whether they are anomalous or not.

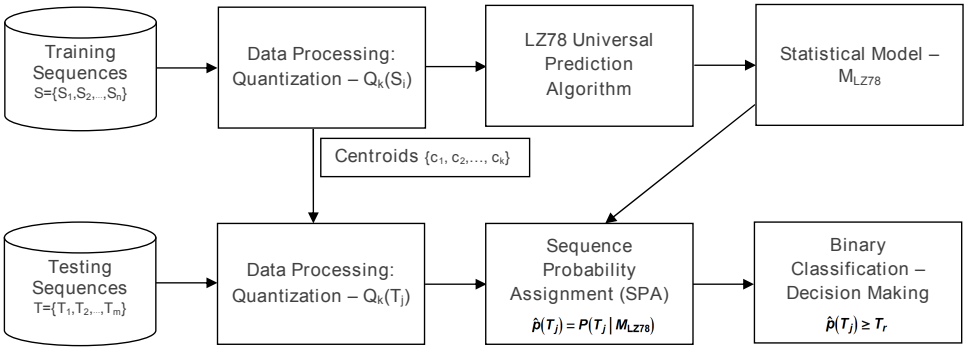

Figure 5: A Classification Model based on the LZ78 universal compression algorithm and its associated probability assignment.

3.3 Learning and Testing

The classification model is divided into a *learning phase* and a *testing phase*. In the learning phase, available data of normal behaviour is preprocessed (according to the concepts described in the previous section) and an LZ78 probability assignment tree is build, to serve as the statistical model for the normal data. In the testing phase, or the actual operation of the detection system, new, unclassified data arrives. This data goes through *the same preprocessing*, resulting in quantised sequences over the same alphabet. The sequences are then tested against the model. Figure 5 depicts the key building blocks.

Specifically, in the learning phase, an LZ78 statistical model is built according to the algorithm in Section 3.1.1, based on a given training set of discrete, quantized sequences over a finite alphabet $S = \{S_1, S_2, \ldots, S_k\}$. Training is done only on normal, benign behaviour. Moreover, it is important to note that in practical cases, one might not have a long enough sequence from a single, normal flow. In such cases, normal sequences can be concatenated together to generate one long sequence from which the tree is built. The size of the alphabet has the following impact. On the one hand, small alphabet size results in low complexity and a robust model (without over fitting). On the other hand, it might group together different types of events, loosing some of the important context in the data (e.g., treating minor differences as equal, hence loosing subtle changes which might reflect distinct phenomenons).

In the testing phase, first, each suspected testing sequence is separately quantized using *the same quantization method and the same set of centroids* $\{c_1, c_2, \ldots, c_k\}$ which were extracted in the learning phase. This is a critical point in the testing phase, as trying to re-calculate optimal quantization for the tested sequences might result in most of them wrongly classified as anomalous. Then, the probability of each suspected sequence (testing sequence) T_j, from a given testing set $T = \{T_1, T_2, \ldots, T_m\}$ is estimated based on the constructed statistical model and classified respective to a predefined threshold T_r. Namely, the probability of each testing sequence, $\hat{p}(T_j)$, is estimated using the sequential probability assignment technique described in Section 3.1.1 *given the LZ78 statistical model built in the learning phase*. Testing sequences for

which $\hat{p}(T_j)$ is greater than or equal to T_r are classified as normal (as they "fit" the model) while a lower than threshold value is classified as anomalous.

It is important to note that for the testing phase the design of the classifier has a few degrees of freedom, based on the required trade-off between complexity, accuracy, and the availability of data. That is, while the optimal decision should be made based on a long enough sequence, in practice, one may combine a few decisions together, based on a few sequences which are known to belong to the same tenant. For example, when a tenant is suspected in malicious behaviour, one might test a few sequences of its data, even if belonging distant, possibly unrated time spans, or based on different features, and make a decision based on all results together.

3.4 Performance Evaluation

The performance of the classifier is measured by the false alarm and detection probabilities, also known as false positive rate (FPR) or Type 1 error and true positive rate (TPR), respectively, and is usually demonstrated using a Receiver Operating Characteristic (ROC) curve. The false alarm probability (or ratio) reflects the number of negative (in this case, normal) instances incorrectly classified as positive (anomalous) in proportion to the total number of negatives in the test, whereas hit detection ratio measures the proportion between the number of positive instances correctly classified and the total number of positives in the test. The ROC curve is generated with respect to a set of thresholds. Each threshold results in a point on the ROC curve, that is, it results in fixed false alarm and detection probabilities. Changing the threshold changes the trade-off between the two probabilities.

3.5 Adaptive Coding and Parallelization

The model described above includes a serial, single pass process of learning and modeling. That is, it builds a single LZ tree which serves as the model. One might wonder if the LZ tree can be updated or enhanced over time. For this, we refer the reader to a few adaptive-LZ techniques, which allow fast rebuilding of the tree in case the *normal data is non-stationary* and changes its form, e.g., [27]. Moreover, the current literature also includes a few algorithms for fast and parallel construction of the tree, e.g., [14].

3.6 Anomaly Detection as a Virtual Network Function

Till now, we have described the anomaly detection algorithm and how it can be used to detect anomalous tenant behaviour and alert in cases of mal-usage or changes and trends in normal behaviour. However, it is important to develop the learning and detection sub-systems of this thread *as virtual network functions* in their own right.

Specifically, consider the learning phase of this algorithm. In this phase, one is required to monitor tenant behaviour and to build a sequence of events for it. Clearly, this should be done either for one specific tenant, or for a group of tenants performing similar tasks. However, as these tenants might migrate, the monitoring tool needs to migrate as well. Moreover, one might choose not to follow a certain tenant, but locate the monitoring tool *at a place which is more convenient system-wise*, as long as the

new location can allow for the same learning process. New tools should be develop to decide how to migrate the monitoring tool, where to migrate to and how to consolidate the data it collects. The same holds for detection. An orchestrator needs to decide where to prob for mal-usage, how often to do it, and, maybe, do it distributively and using aggregated detection techniques.

4 Enabling security in OpenStack Neutron (based on Lanman 2016 paper)

The cloud is taking over the world of computing. Public clouds such as Amazon EC2, Microsoft Azure or Rackspace are widely used, and smaller clouds are being built pretty much everywhere. Network operators are building miniature clouds in their core networks (e.g. DT is deploying racks collocated with PoPs) while mobile operators are deploying processing close to the edge to enable mobile edge computing [19]. Deploying a cloud is no easy task. Major public clouds providers have each developed their own custom cloud management software, but the software is deployment-specific and not available publicly. New cloud players are very numerous and eyeing smaller deployments; having each of them develop cloud software makes no sense.

OpenStack is the leading community effort to build a production-quality, open source platform that enables building public and private clouds with ease. OpenStack has a lot of momentum, with major companies investing human and capital resources, and is reaching maturity. Hundreds of OpenStack clouds have already been deployed [2]. We focus on Neutron, the networking component of OpenStack, that allows users to specify their high-level networking configuration and deploys it. Neutron is notoriously unreliable, to the point where it has become known as the weakest link in OpenStack and bashed in popular media by company executives [26]. Neutron has certainly improved recently, but it is still far from perfect.

In this position paper we propose to use network static analysis, in particular symbolic execution [32], to improve Neutron. Rather than provide a definitive solution, we provide a high level approach to solving Neutron's woes. Our key idea is to use symbolic execution to analyze the properties of a) the tenant virtual network configuration before deployment and b) the actual network dataplane after deployment.

We have built a prototype to showcase our approach and check its validity, and performed a preliminary evaluation. Our initial results are promising: verification is fast (seconds) and can detect common problems with Neutron deployments.

4.1 OpenStack Networking with Neutron

Network virtualization in OpenStack is enabled by Neutron [23]. It offers an API to tenants allowing the creation of virtual networks that are decoupled from the underlying networking topology and the configuration chosen for deployment. The tenant network is then instantiated on the physical topology, and this mapping is influenced by the way the cloud provider has deployed OpenStack. Neutron layering is captured in Figure 6 and it aims to achieve the following goals:

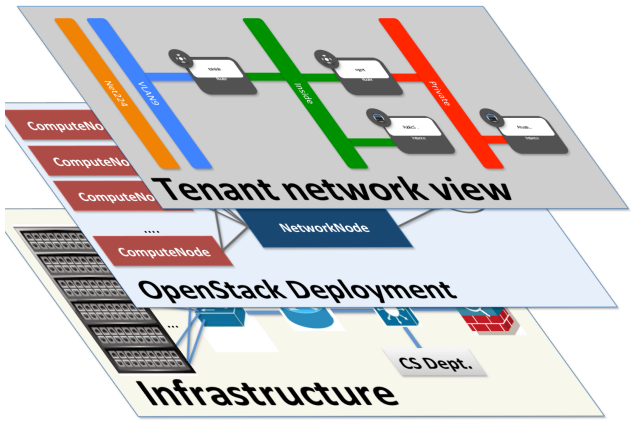

Figure 6: OpenStack Networking Layers.

- **Policy Compliance.** Neutron aims to allow tenants to configure networks that satisfy their high-level security policies, for instance separation of public and back-end traffic, reachability, etc.
- **Implementation Correctness.** The properties of the virtual network configured by the tenant should be matched by the actual deployment. For instance, Internet packets that can reach an instance in the virtual network should also reach the instance in the instantiated network.
- **Traffic Isolation.** A tenant's traffic should not reach other tenants unless Neutron is explicitly configured to do so.
- **Performance.** Neutron must allow cloud providers and tenants to effectively utilize fast interconnects including 40Gbps and 100Gbps Ethernet.

Achieving all these properties simultaneously is very tricky. Achieving tenant policy compliance appears simple: the tenant only has to correctly configure its virtual network (given the appropriate configurationAPI), however tenants are not networking professionals usually, so many configuration errors are possible. Furthermore, the short and inexpensive configure-deploy cycle facilitates the introduction of configuration bugs.

Implementation correctness and traffic isolation are achieved through coding best practices in the open-source community; however this implies that only very popular approaches will be heavily scrutinized, and many bugs will exist in less popular code. Achieving correctness and isolation in the context of proprietary drivers for third-party networking hardware is even more challenging. Even with code review, there is no guarantee that these properties are met in practice. Finally, achieving high networking performance implies using pass-through technologies such as SR-IOV for virtual machines, and relying on networking hardware to implement Neutron. SR-IOV traffic bypasses the local hypervisor stack (e.g. iptables and Openvswitch) and only basic security is provided, meaning that other tenant-specified functionality (such as fire-

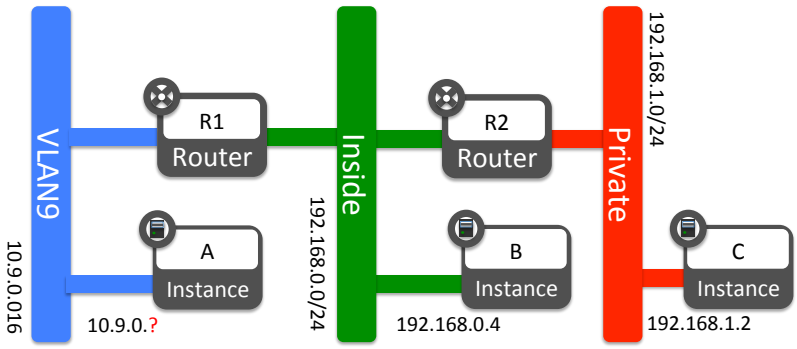

Figure 7: Example of a tenant network topology in the Horizon GUI.

walling) may not be provided. We now provide a more detailed view of OpenStack networking and highlight the difficulties when trying to meet the OpenStack goals.

Tenant network view. Tenants use an virtual network view to configure the way their instances are connected. To this end they can use layer 2 networks (flat or VLANs), routers, firewalls, VPNs and load balancers. In more detail, tenants can use the Horizon GUI or the Neutron API as follows:

- The simplest choice is to use a flat network where virtual machine instances are assigned IP addresses from a DHCP server run by the cloud provider. The DHCP server also provides a gateway for outgoing connections and performs network address translation. Incoming connections are dropped by default.
- Create VLAN(s) and associated subnet(s) where the connected instances are assigned, at configuration time, distinct IP addresses from the subnet's range. One instance can be connected (have interfaces in) multiple VLANs.
- Create routers that can interconnect different VLANs and provide Internet connectivity. All addresses assigned (either with subnets or DHCP) are private and thus not reachable from the Internet. Outgoing Internet connectivity can be provided by using NAT.
- Assign floating IPs if incoming connectivity is desired (e.g. for the tenant to be able to ssh into its instance). A floating IPs is a public IP addresses that is associated to a private IP address of the tenant. Neutron perform address translation between the floating address and the private one, transparent to the VM.
- Specify firewall rules to be applied to specific ports.
- Further constructs include VPNs and traffic load balancing, and this list is likely to increase in the future.

In Figure 7 we show an example tenant networking configuration in the Horizon GUI of our university's OpenStack deployment. The tenant wants to deploy a web server connected to a database server. Its policy is that that its web server should be

publicly accessible on port 80, and that the database is only reachable from the web server, and not the Internet.

The tenant configures two VM instances: B will run the web server and is connected to the green VLAN, and C is the database server and is connected to the red VLAN. The two VLANs are connected via router R2, and the green VLAN is connected via router R1 to the Internet. R1 is setup as an Internet gateway. The blue VLAN (VLAN9) is created automatically by this particular OpenStack deployment and is where the Internet gateway resides. The tenant also starts A for testing purposes and attaches it to VLAN9.

Is this network configuration a correct implementation of this tenant's policy? An experienced OpenStack network administrator will, most likely, be able to debug this configuration quite easily and find that, for instance, there is no incoming connectivity to the web server since a floating IP has not been assigned. The average OpenStack tenant, however, will not be a networking specialist. Such a person needs a fairly large set of skills: they need to be able to create a VM image, they need to install and manage various servers (e.g. web and database), setup the tenant network and, finally, develop their application logic. In the pre-cloud era, multiple people were needed maintain such a website: e.g. a networking administrator, a web admin, a web developer. In the cloud era, it is possible (and expected) that a single person will fulfill all these roles, but they will not be networking experts.

Ensuring that a tenant policy is met by a network configuration requires more than manual debugging - we need tools that help the tenant quickly find the problems and fix them.

Cloud provider view of networking. When deploying Neutron, the cloud provider has to meet the above goals (correctness, isolation, performance) in the context of its local network policy, and with the added constraint of isolating tenant traffic from local traffic and treating tenant traffic as "outside" traffic when deciding access to local machines.

When the tenant starts the deployment of its configuration, virtual machine instances are placed on the available OpenStack Compute Nodes and the tenant networking configuration is instantiated; the instantiation depends on the way the cloud provider has deployed Neutron.

One of the most popular networking deployments is to use Openvswitch [25] on every Compute Node, use iptables for firewalling and have a Network Node running on one of the servers to implement NAT and routing. Traffic leaving from the Compute Nodes is encapsulated in VXLAN tunnels and carried to the Network Node, which also enables Internet connectivity. This configuration meets all the requirements, except the performance one.

There is however great flexibility in how a cloud provider can configure its network to work with Neutron, including:

- Using fault-tolerant Network Node implementations called VRRP.
- Deploying routing and firewalling on every Compute Node, in a configuration called DVR.

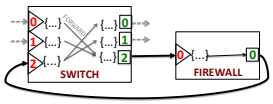

Figure 8: SymNet network models: elements and interconnections

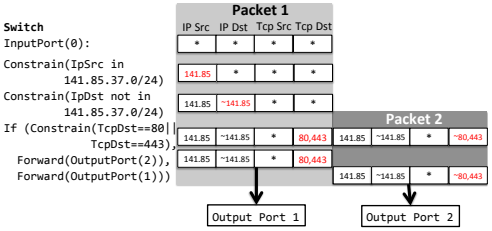

Figure 10: Symbolic execution example: injecting a symbolic packet in the switch model on input port 0. Two packets are generated with different constraints, one that exits on port 1 and one on port 2.

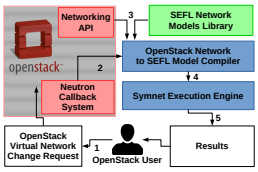

Figure 9: Verifying the tenant network

- Using hardware switch support and VLANs to ensure the tenant traffic isolation instead of VXLAN.
- Using merchant hardware to implement routing and firewalling (called firewall as a service).

A performance-oriented deployment would use SR-IOV for each tenant, bypassing the hypervisor stack, and apply VLAN encapsulation on the NIC. VLAN support in switches would then be used to carry traffic to a hardware firewall (e.g. a CISCO ASA box), where firewall rules belonging to the cloud provider and tenants would be applied. Routing between VLANs and NATting could also be implemented in hardware. In such a deployment, ensuring the isolation and correctness properties hold is trickier: tenant firewall rules could clash or overlap with provider rules, and the correct order in which they should be applied is not obvious. Installing a set of predefined rules for all tenants is feasible, however allowing per-tenant firewall policies is difficult to do.

This versatility of Neutron makes it adaptable for a wide range of uses and requirements. However, it also raises questions about the correctness of any non-trivial individual deployment, especially when less scrutinized third-party software or hardware are used.

4.2 Symbolic Network Execution

We propose to use static network analysis to improve Neutron. There are many static network analysis tools such as [37, 18, 13, 17, 24, 32]; they all require as input a model of network functionality including the processing performed in different boxes, such as switches and routing, as well as a snapshot of the network state, for instance router forwarding table snapshots or switch dynamic MAC tables. Then, the tools "simulate" what happens when certain packets are injected at different parts of the network: given a packet with specific header fields, static analysis tools tracks the path of the packet through the network and the evolution of its header fields.

The strength of static analysis stems from its ability to quickly test a wide range of possible packets (e.g. all possible headers destined to a server) without having to iteratively test all possible combinations of concrete header fields. How this is achieved

depends on the tool being used. The different tools offer different tradeoffs in terms of speed of analysis and properties checked, and an overview of all these tools can be found in [32]. A very good option is to use network symbolic execution, as enabled by the SymNet symbolic execution tool. We use SymNet in this paper and provide a brief overview below; please refer to [32] for more details.

In SymNet, network boxes are modeled as modular elements having an arbitrary number of input and output ports. Network links are modeled as directed edges between the output port of an element to an input port of another one. To describe the functionality of a box, each port has associated a set of instructions that are executed when a packet reaches that port. The set of instructions associated to each port is written in a language called SEFL and described in detail in [32]. SEFL is a simple imperative programming language and offers usual instructions e.g. assignment, if and basic expressions (addition, subtraction). SEFL is optimized to allow scalable network symbolic execution as follows:

- The `constrain` instruction adds restrictions on header fields, such as firewall rules.
- The `forward` instruction makes a packet go to a specified output port.
- The `fork(p1,p2,...)` instruction sends a copy of the packet to each of the specified output ports (p1, p2, ...).
- There are no unbounded loops in SEFL.

We give an example in Figure 8 where we have two network elements: a three-port Openflow switch and a firewall connected to port 2 of the switch. Element input ports are numbered in red and shown as triangles; output ports are numbered in green, and shown as squares. Port 0 is connected to an inside network, and port 1 is connected to the Internet. The configuration aims to ensure that all packets are checked by the firewall, and performs ingress filtering for the ports connected to the two networks. Packets from the firewall port are sent to the local network or the Internet based on their destination address.

```
InputPort(0):
   Constrain(IpSrc in 141.85.37.0/24)
   Constrain(IpDst not in 141.85.37.0/24)
   If (Constrain(TcpDst==80||TcpDst==443),
      Forward(OutputPort(2)),
      Forward(OutputPort(1)))

InputPort(1):
   Constrain(IpSrc not in 141.85.37.0/24)
   Constrain(IpDst in 141.85.37.0/24)
   If (Constrain(TcpSrc==80||TcpSrc==443),
      Forward(OutputPort(2)),
      Forward(OutputPort(0)))

InputPort(2):
```

```
If (Constrain(IpDst in 141.85.37.0/24),
   Forward(OutputPort(0)),
   Forward(OutputPort(1)))
```

Below we provide the model for the firewall which only allows HTTP traffic from our local network and the associated return traffic. To keep per-flow state the model creates a metadata called `FirewallState` and sets in the packet. When the return traffic arrives, it will have the same variable set and will be allowed through.

```
InputPort(0):
If (Constrain(FirewallState==1),
   Forward(OutputPort(0)), //allow seen flows
   InstructionBlock(//outgoing HTTP traffic
     Constrain(IpSrc in 141.85.0.0/16),
     Constrain(IpDst not in 141.85.0.0/16),
     Constrain(TcpDst==80||TcpDst==443),
     Allocate(FirewallState),
     Assign(FirewallState,1),
     Forward(OutputPort(0))
  ))
```

To understand symbolic execution, we inject a packet on input port 0 of the switch and trace its evolution in Figure 10 (only switch processing on port 0 is captured in the figure). The packet has all header fields initialized to symbolic values:

1. The values of IpSrc and IpDst are constrained. Then, the If instruction results in two packets (or symbolic execution paths).

2. The "else" branch packet, packet 2, captures non-HTTP traffic and is forwarded to the switch output port 1; at this point symbolic execution of packet 2 stops, as output port 1 is not connected in our model.

3. On the If branch, the TCP destination port is constrained to be HTTP(S), and the packet is forwarded on output port 2 and onto the firewall on port 0.

4. The packet is processed at the firewall. There is no FirewallState in the packet so the else branch runs, and constraints are added for TcpDst. The state is allocated and assigned and the packet sent to output 0.

5. The packet enters the switch via input port 2, the else branch is run and the packet finally reaches output 1.

The output of symbolic execution is 1) a set of packets that have reached unconnected output ports, together with 2) a set of packets that have failed en-route, because some of the constraints applied to their header fields did not hold. The first category is more interesting for network verification. For each path, SymNet reports in a JSON-formatted output file all the instructions executed, all the ports visited, the values and/or constraints for all header fields.

In particular, if we examine the output from SymNet for the example above, we can conclude that outgoing reachability is permitted for all packets with correct IP

addresses. We also find that the packet headers are not modified by our configuration: all header fields are bound to the same symbolic variable at the beginning and end of the execution. To gain more insights in this configuration, we also inject a purely symbolic packet at input port 1, and also add a symbolic value for the FirewallState variable. The results show that all HTTP response traffic is dropped unless firewall state is set to 1, and all other traffic is allowed unmodified.

4.3 Analyzing OpenStack with SymNet

We provide a high level description of how SymNet can be used to improve Neutron to achieve all the its goals, namely tenant policy compliance, implementation correctness, traffic isolation and performance. To this end, we will rely on SymNet for network verification on two layers: the abstract tenant network and the deployed network.

4.3.1 Checking the abstract tenant network.

To check policy compliance, the tenant can use SymNet to run reachability from all instances to all other instances and the Internet before the configuration is deployed. The SymNet output allows the tenant to quickly check whether the reachability matches their expected behavior.

We have implemented support for such testing in Neutron and our implementation is shown in Figure 9. Whenever a tenant uses Neutron to create or modify a virtual network topology (step 1) and prior to the effective deployment, we insert an additional verification step that uses SymNet to analyze the tenant configuration. The deployment process is stalled until the analysis process ends.

To receive information about the creation of new topologies or the modification of existing ones, SymNet registers to the Neutron callback system. Every time such a callback is executed (step 2), SymNet queries the OpenStack HTTP Networking API [1] for the Neutron network configuration currently in place. Next, we need a SEFL model of the tenant network. We have modeled the functionality offered by Neutron including routers, firewalls, NATs and created a library of SEFL models. For every network function used by the tenant, we obtain the corresponding SEFL element by configuring the generic SEFL library component with the parameters provided by the tenant (step 3). The element's input and output ports are then interconnected according the virtual links provided by the tenant.

Finally, symbolic execution is performed by injecting symbolic packets at all the instances' network attachment points, as well at the Internet gateway. The result is detailed reachability for all VMs in the abstract tenant network. Currently, the result of the analysis is provided to the tenant in JSON format, which manually checks whether it obeys its policy. In the future, we plan to automatically check simple popular policies, such as:

- All-to-all VM communication for ICMP, UDP and TCP traffic.
- Outgoing ICMP and TCP reachability from all instances.
- Incoming SSH reachability (TCP on port 22) for all instances.

4.3.2 Checking the deployed network dataplane.

The second step of our verification happens *after the tenant's configuration is deployed*, and its goal is to ensure *isolation of tenant traffic* and *correctness of implementation*. To this end, we use two complementary approaches: guided testing and symbolic execution.

Guided testing[32, 39] is very simple: for every path resulting from the tenant network analysis, generate a matching packet and inject it in the actual network, observing the packets reachable at other instances or in the Internet. The big advantage for guided testing is that it can be run by the tenant, without cloud provider support and is independent of the deployed network. We have implemented a simple version of guided testing by using SymNet to generate test packets for the provided paths, generating packets using the Click modular router [15] and using `tcpdump` for reception.

Guided testing is not exhaustive: even if it reports success for the tested packets, there are no guarantees the deployment is indeed correct, or that tenant traffic is correctly isolated.

To get hard guarantees we resort to symbolic execution of the deployed network. Compared to the abstract tenant network, the setup needed to symbolically analyze the real network is much more complex: we need accurate SEFL models and snapshots of the dataplane state for all the boxes (hardware and software) deployed in the cloud-operator network that interact with tenant traffic. This includes network ports for *all* instances of all tenants, hypervisor functionality (software switching, tunelling and local filtering), OpenStack network nodes, hardware switches and routers.

Creating a solution that is applicable to all networks is an extremely challenging task, given the heterogeneity of deployed infrastructure and the pervasive use of middleboxes [28]. Generating accurate SEFL models of middlebox functionality is not trivial and requires a lot of expert effort. There is currently no generally applicable recipe to all networks. Modeling real networks, however, is feasible. In prior work we have developed a model of our department's network [32]. This model relies on the following building blocks: a) a switch model that can be automatically created when given a snapshot of the dynamically-learned MAC table; b) a router model created from a snapshot of the forwarding table, obtained via standard commands on Cisco routers, and c) a CISCO firewall model (Application Security Appliance) that is created automatically given the configuration file. We are currently using this model as a basis to implement OpenStack deployment checking in our network.

Given an accurate model of a deployed network, we can use symbolic execution to ensure key properties for Neutron. We initiate reachability checks from the new tenant's instances and from the Internet and use the SymNet output to check isolation and correctness, as described next.

Checking isolation. If any VM from any other tenant is reachable from or reaches the instances of the new tenant, we report a violation of the isolation properties. Symbolic execution enables this analysis, but it is complex because its runtime depends grows linearly with the number of tenants/VMs. Optimizations are needed to ensure it scales to large clouds and may include only checking outgoing connectivity from the new tenant, or defining tenant equivalence classes and running reachability between equivalence classes.

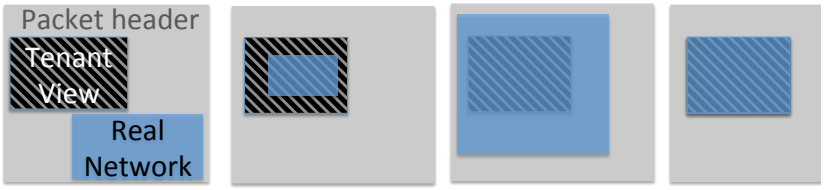

Figure 11: Checking for equivalence between the tenant abstract network configuration and the deployment

Checking correctness. We can compare the sets of paths resulting from the abstract tenant view and the deployment view to decide whether the deployment correctly implements the tenant network. In essence, we want to decide whether the two configurations are equivalent. Equivalence is undecidable for general programs, however we restrict our definition to reachability: we want to ensure the same packets are reachable in the two configurations.

Before we describe the algorithm, we give a few definitions. We assume all packets contain the same set of headers H_1, H_2, etc., for which we care to verify equivalence; all other headers are ignored. Let $C_{H_i}(P_j)$ denote the set of constraints of header field H_i in packet P_j at some reference port in the network. Let $C(P_J) = C_{H_1}(P_J) \cap C_{H_2}(P_J) \cap \cdots$ denote the conjunction of all constraints applied to all headers of packet P_J at the same port.

Output Equivalence Algorithm. *Input:* The set of symbolic packets P_i, $i = 1...N$ and Q_j, $j = 1...M$ obtained by checking reachability between ports a and b in the tenant network and real network, respectively.

Algorithm: To test for equivalence, first we compute the disjunction of constraints of all packets in the two networks at port b: $X = C(P_1) \cup C(P_2) \cdots \cup C(P_N)$ and $Y = C(Q_1) \cup C(Q_2) \cdots \cup C(Q_M)$. The two sets of paths are equivalent if and only if the expression $(X \cap \neg Y) \cup (\neg X \cap Y)$ is not satisfiable.

The intuition for this algorithm is given in Figure 11 where we show a packet with two header fields. The hashed areas corresponds to the reunion of constraints at node b resulting from symbolic execution of the tenant network and the actual network, respectively. The output equivalence algorithm aims to determine if the two areas overlap perfectly.

We have four cases: first, the tenant reachability does not overlap at all with the deployment reachability, resulting in a completely broken instantiation of the tenant network: the tenant has no reachability for its traffic, while unwanted traffic is allowed. In the next case we have partial reachability, but at least the configuration is safe: unwanted traffic is stopped. The third configuration allows full reachability, however also allows other packets through—this could indicate that the firewall rules have not been instantiated properly. Finally, the last case is when there is perfect overlap, and the two configurations are equivalent from an output port point of view.

Note that output equivalence is fairly weak in networks that modify header fields. There, additional constraints could be applied on the initial values of the header fields that influence reachability, yet they are not captured by output equivalence. In our

future work we plan to explore stricter notions of equivalence such as input-output equivalence.

4.4 Preliminary Evaluation

We ran reachability analysis for the tenant configuration in Figure 7 by injecting a symbolic packet at all the tenant instances and the gateway. It takes 5 to 7 seconds to run the reachability analysis, for which 3 to 5 seconds are spent in Neutron API calls and 2s to generate the SEFL code and run the reachability analysis.

Our first analysis showed no connectivity at all because the tenant configuration didn't have static IPs assigned and no DHCP server was enabled either. We changed the configuration by assigning static IPs to all the instances; the analysis found that all instances can communicate directly. Further, A and B have outgoing Internet connectivity, and that connections initiated from the Internet are not allowed. Finally, C has no Internet connectivity.

Next, we deployed the configuration and ran our guided testing implementation. Surprisingly, guided testing showed that instance C had Internet connectivity but it couldn't access instances A or B; this is the exact opposite of the tenant configuration, where C has no Internet connectivity but it can reach A and B. It turned out that this problem was transient: when we reran the guided testing scheme, the results were as expected. The culprit was identified to be the propagation latency between high level commands and driver implementation in Neutron, which we measured to be on the order of minutes in our deployment.

4.5 Conclusions

OpenStack Neutron is a complex piece of software, providing different views to different stake-holders and incorporating code from multiple parties. It has been anecdotally called the weakest link in OpenStack [26].

We argue this happens because debugging currently only relies on standard best practices for code development, without taking into account the particularities of Neutron. As a complement to existing approaches, we propose using static network analysis to improve OpenStack Neutron. We have shown how network symbolic execution can be used on two levels: to check the abstract tenant network and its deployment in the actual network. We have an initial implementation of these ideas that we have integrated with the Neutron API. Our preliminary evaluation has found interesting nuggets: a propagation delay bug in OpenStack (that was known) and can help tenants more easily deploy their networks.

This is a work in progress, and many refinements are needed to our prototype until it can be applied to a wide range of configurations automatically. On the algorithmic side, we intend to explore stronger version of equivalence. Finally, we intend to fully model our department's network (including Openvswitch, Neutron nodes, etc) and perform full symbolic analysis of the deployed network.

5 Symbolic execution - model equivalence & applications

There is a fundamental tension between the runtime speed and the symbolic execution speed of a program [35].When analyzing a simple switch model, symbolic execution can take orders of magnitude less time when the code has been optimized for symbolic execution. However, executing that same code in practice will result in very poor performance [32]. Many researchers have observed this and produced optimized models that enable symbolic execution; unfortunately, there is gap between the model and the actual code.

We take a principled approach to understand what transformations are correct and safe when optimizing models for symbolic execution. Intuitively, we want the optimized model to behave in the same way as the original code.

One approach for checking model equivalence is to look at the domain sets for each program variable. Suppose symbolic execution for model M yields n paths and on each path i, the variable v is bound to symbolic expression e_i. The *domain set* for variable v is thus $d_{M_1} = e_1 \vee e_2 \vee \ldots \vee e_n$.

Two models M_1, M_2 are *output-equivalent* iff the domain sets for all variables are equivalent ($d_{M_1} \wedge \neg d_{M_2} = \mathit{false}$).

Output equivalence is limited in capturing input-output dependence. For instance, programs `if (x > 0) x = 1; else x = 0` and `if (x > 0) x = 0; else x = 1;` are output equivalent, but do not behave in the same way.

A more conservative alternative is to define equivalence in terms of execution paths: M_1, M_2 are equivalent if each path from M_1 is equivalent to some path from M_2 and vice-versa.

However, this approach leaves little room for symbolic execution optimizations which may target path reduction.

A tradeoff between the two is to consider *state-dependent equivalence*. Say P is a *state-predicate* (e.g. $P \equiv$ `x != 0`). Two models are P-equivalent if they produce precisely the same output on any input that satisfies P. State predicates offer flexibility in deciding how strong requirements we place on model equivalence.

5.1 Implementation

In order to optimize models for symbolic execution, we trade off between: (i) the number of execution paths and (ii) the number of constraints per path, for each variable.

Optimization consists in a sequence of equivalence-preserving transformations similar to compiler code optimizations.

First, we perform general optimizations e.g. removing conditionals which result in only one successful path, removing dead code, merging consecutive variable constraints in a single one (thus having fewer invocations on the constraint solver). For some transformations, we may only preserve equivalence with respect to specific state predicates.

Second, we apply transformations aimed at optimizing for criteria (i) or (ii). For instance, in `if (x > 0)` p_1 `else` p_2, if we can determine that `x > 0` holds (univer-

sally, or w.r.t. our given state predicate), we can transform the program to `assert (x > 0);` p_1, thus reducing program branching.

References

[1] OpenStack Networking API 2.0 Specification. `http://developer.openstack.org/api-ref-networking-v2.html`.

[2] OpenStack users share how their deployments stack up. `http://superuser.openstack.org/articles/openstack-users-share-how-their-deployments-stack-up`.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[4] P. Burghouwt, M. Spruit, and H. Sips. Towards detection of botnet communication through social media by monitoring user activity. In *Information Systems Security*, pages 131–143. Springer, 2011.

[5] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI'08*.

[6] M. Celenk, T. Conley, J. Willis, and J. Graham. Predictive network anomaly detection and visualization. *Information Forensics and Security, IEEE Transactions on*, 5(2):288–299, 2010.

[7] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.

[8] S. Chang and T. E. Daniels. P2p botnet detection using behavior clustering & statistical tests. In *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*, pages 23–30. ACM, 2009.

[9] M. Dobrescu and K. Argyraki. Software dataplane verification. In *Proc. NSDI'14*, NSDI'14.

[10] M. Feder, N. Merhav, and M. Gutman. Universal prediction of individual sequences. *Information Theory, IEEE Transactions on*, 38(4):1258–1270, 1992.

[11] J. Francois, S. Wang, W. Bronzi, R. State, and T. Engel. Botcloud: detecting botnets using mapreduce. In *Information Forensics and Security (WIFS), 2011 IEEE International Workshop on*, pages 1–6. IEEE, 2011.

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

[13] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI'12*.

[14] S. T. Klein and Y. Wiseman. Parallel lempel ziv coding. *Discrete Applied Mathematics*, 146(2):180–191, 2005.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[16] A. Lapidoth. *A foundation in digital communication*. Cambridge University Press, 2009.

[17] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *Proc. NSDI'15*.

[18] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Sigcomm*, 2011.

[19] Michael Till Beck and Martin Werner and Sebastian Feld and Ludwig Maximilian and homas Schimper. Mobile Edge Computing: A Taxonomy. In *AFIN 2014*.

[20] J. Neyman and E. S. Pearson. *On the problem of the most efficient tests of statistical hypotheses*. Springer Series "Breakthroughs in Statistics", 1992.

[21] M. Nisenson, I. Yariv, R. El-Yaniv, and R. Meir. Towards behaviometric security systems: Learning to identify a typist. In *Knowledge Discovery in Databases: PKDD 2003*, pages 363–374. Springer, 2003.

[22] S.-K. Noh, J.-H. Oh, J.-S. Lee, B.-N. Noh, and H.-C. Jeong. Detecting p2p botnets using a multi-phased flow model. In *Digital Society, 2009. ICDS'09. Third International Conference on*, pages 247–253. IEEE, 2009.

[23] Openstack. Neutron Networking. `https://wiki.openstack.org/wiki/Neutron`.

[24] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying Isolation Properties in the Presence of Middleboxes. Tech Report arXiv:1409.7687v1.

[25] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. G. s, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *NSDI*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[26] T. Register. HP: OpenStack's networking nightmare Neutron 'was everyone's fault'. `http://www.theregister.co.uk/2014/05/13/openstack_neutron_explainer/`.

[27] G. Seroussi and A. Lempel. Lempel-ziv compression scheme with enhanced adapation, Sept. 7 1993. US Patent 5,243,341.

[28] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. y. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *SIGCOMM*, 2012.

[29] S. Shin, G. Gu, N. Reddy, and C. P. Lee. A large-scale empirical study of conficker. *Information Forensics and Security, IEEE Transactions on*, 7(2):676–690, 2012.

[30] S. Siboni and A. Cohen. Botnet identification via universal anomaly detection. In *2014 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 101–106. IEEE, 2014.

[31] S. S. Silva, R. M. Silva, R. C. Pinto, and R. M. Salles. Botnets: A survey. *Computer Networks*, 57(2):378–403, 2013.

[32] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: scalable symbolic execution for modern networks. In *Sigcomm*, 2016.

[33] W. T. Strayer, D. Lapsely, R. Walsh, and C. Livadas. Botnet detection based on network behavior. In *Botnet Detection*, pages 1–24. Springer, 2008.

[34] R. Villamarín-Salomón and J. C. Brustoloni. Identifying botnets using anomaly detection techniques applied to dns traffic. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 476–481. IEEE, 2008.

[35] J. Wagner, V. Kuznetsov, and G. Candea. Overify: Optimizing programs for fast verification. In *Proc. HotOS'13*.

[36] Y. Xiang, K. Li, and W. Zhou. Low-rate ddos attacks detection and traceback by using new information metrics. *Information Forensics and Security, IEEE Transactions on*, 6(2):426–437, 2011.

[37] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *Proceedings of Infocom*, 2005.

[38] C. Yang, Y. Song, and G. Gu. Active user-side evil twin access point detection using statistical techniques. *Information Forensics and Security, IEEE Transactions on*, 7(5):1638–1651, 2012.

[39] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT'12*.

[40] X. Zhai, K. Appiah, S. Ehsan, G. Howells, H. Hu, D. Gu, and K. McDonald-Maier. A method for detecting abnormal program behavior on embedded devices. *Information Forensics and Security, IEEE Transactions on*, 10(8):1692–1704, Aug 2015.

[41] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.