UNIVERSITATEA POLITEHNICA BUCUREŞTI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE DIPLOMA

Infrastructuri honeypot folosind rețele comandate
și infrastructuri containerizate

Coordonator ştiinţific:
Prof.Dr.Ing. Răzvan Victor Rughiniș
Drd. Ing. Ioan-Mihail Stan

Absolvent:
Carol Sebastian Bontaș

BUCUREŞTI

Iulie 2022

POLITECHNICA UNIVERSITY OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

Bachelor Thesis

Honeypot generator with network controllers and containerized infrastructure

Thesis supervisor:
Prof.Dr.Ing. Răzvan Victor Rughiniș
Drd. Ing. Ioan-Mihail Stan

Carol Sebastian Bontaș

Bucharest

July 2022

Abstract

The domain of cyber security represents a constant struggle between researchers and hackers, a continuous competition where the global digital infrastructure is at stake. Throughout the years, various protective measures have been developed to prevent against an increasing amount of exploits that are becoming more and more complex and refined.

Instead of concentrating on general mitigating techniques, our proposal orients towards creating performant honeypots which are safe and isolated environments that attract malicious users for the purpose of studying their invasive techniques. The proposed implementation may represent a promising framework used as a developing platform for more complex honeypots used in either researching malicious human behavior or in the IT industry as a defensive measure. Moreover, we designed an intelligent orchestration pipeline, focused on pruning low-priority components throughout the honeypot's lifetime, which can be integrated in existing containerization technologies outside the security-driven systems.

Regarding achieved results, the proposed implementation accomplishes notable results in deceptive techniques, isolation and effortless configurability. In addition, this work also provides a case-study general comparison between existing intrusion detection algorithms in order to select suitable candidates which satisfy the honeypot's requirements.

1 Introduction

The domain of honeypots still represents a researching area in which various subdomains intertwine such as cyber security, networking and machine learning in order to fully understand the natural thinking process behind modern exploitation techniques and how to either properly mitigate or predict them. Nowadays, security systems used by governmental institutions or corporations are hidden from the eye of the public for obvious reasons, making them be accessible to a limited number of developers resulting in a slower process of improvement than an open-source project. Moreover, by accepting this closed-source behavior, such projects would not benefit from cooperation between specialists, consultants and researchers that are not part of the owning organization. Considering this drawback, we propose an open-source implementation that provides a more dynamic behavior used in increasing the overall authenticity of the honeypot and providing more complex challenges that convince the attacker to invest higher and higher periods of time.

Our implementation can be perceived as an initial platform for developing more complex honeypots by enabling different organizations to utilize at full potential our flexible architecture.

The rest of the paper is organized as follows. Section 2 presents the state of the art, related works and our main motivation. Section 3 presents the architecture and key concepts regarding the implementation. Section 4 shows the testing cases and results from evaluation experiments, while section 5 and 6 present the future works and several conclusions.

2 State of the art

Honeypots represent a versatile approach in researching the continuous evolution of malicious methods and tools by enabling hackers to interact with pre-configured services and environments.

A brief classification of honeypots can be made depending on their purpose:

- Low-interaction (production) honeypots: emulated services integrated along production servers within the same network, obfuscates a genuine attacker by adding various potential targets with the objective of wasting their time. Common features include low-interaction behavior but easier configuration and maintainability.
- High-interaction (research) honeypots: complete virtual environments with the sole purpose of attracting various attackers. Unlike production honeypots, which are deployed as a defensive measure, the research category memorizes malicious actions and builds patterns of behaviors as data sets for further investigations. Common features are represented by the total opposite of their production counterparts.

Choosing between low and high interaction depends on various factors:

- purpose:
    - is my honeypot used in production or research?
- return of investment:
    - is the cost of potential losses of data and production infrastructure when we do not have a honeypot, higher than the cost of maintaining one?
    - is my system prone to security issues in order to employ complex defensive measures such as honeypots?
- the risk potential:
    - should I risk deploying a honeypot which, by compromising it, can become a proxy node used for attacking other systems worldwide?

There are no general successful recipes for ensuring security, there may be situations where honeypots would bring issues instead of solutions.

Due to the requirement of adaptability based on a particular scenario, honeypots are not always categorized in the binary manner explained above. The level of interaction can be placed on a spectrum dating from low to high, the ability of changing this type of behavior is a desired feature in the rapid development of the nowadays software.

Since security experts do not possess an honest “review” from genuine attackers, they are forced to employ various methods of analyzing malicious behavior through indirect means, which makes difficult to evaluate, through quantifying parameters, which honeypots are better than the others.

2.1 Related work

For this paper, we use as a development and architectural benchmark, the solution named T-Pot developed by the telecommunication and internet service provider, Deutsche Telekom (T-Mobile). What is remarkable about T-Pot is that it represents a platform for integrating multiple honeypot solutions resulting in an aggregated system which can react to a wide variety of attacks at a global scope.

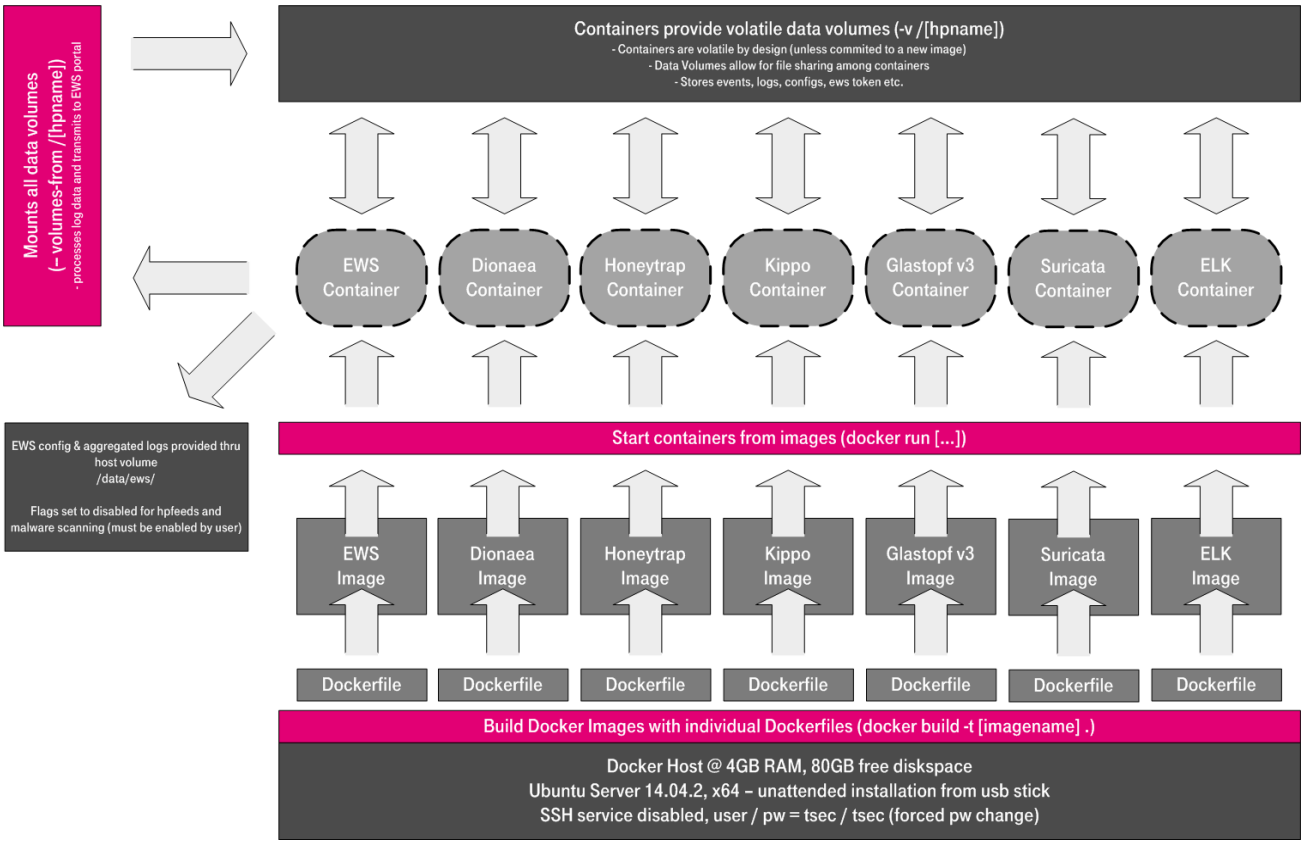

Fig. 1: General architecture of T-Pot[1]

Figure 1 presents the primary components of T-Pot. It utilizes various honeypots (Kippo: used for SSH intrusions, emulates bash commands and filesystems; Dionaea: low-interaction honeypot used for detecting injected shellcodes; Suricata: network intrusion detection system).

Even though T-Pot provides a multi-honeypot platform with easy configurability it also has major flaws:

All the above-mentioned honeypots belong to the low-interaction category, which means that any update to the emulated service forces the developer to update their respective honeypot

There is no interoperability between honeypots, for example a malicious SSH attempt would only trigger a response from the SSH-designated honeypot instead of crafting a more complex countermeasure from other areas of services (networking, web applications etc.)

[1] https://github.com/telekom-security/tpotce

2.2 Motivation

In order to overcome these issues presented in section 2.1 we are contributing with a high-interaction honeypot framework which brings serious improvements:

It abstracts the types of services used at application level, the types of networking protocols and types of parameters and metrics used to quantify malicious behavior. A developer can choose from a repository such as Vulhub[2] what vulnerable applications will be integrated.

- enables the behavior of the honeypot to be defined by heuristics instead of a pre-determined recipe
- enables the honeypot to be hybrid (low or high interaction) depending on the situation: the administrator can prioritize the purpose of the honeypot either being a defense mechanism (by building a heuristic which provides services from a different area of interest for the attacker but still be exploitable with the aim of wasting their time as much as possible) or being a research tool in studying malicious behaviors by collecting a wide variety of actions taken by the attacker
- elegant way of introducing genuine production systems along the honeypot network in order to increase its authenticity in the eyes of experienced exploiters by utilizing an intrusion detection system along with software-defined networks and controllers which isolate attackers whenever an alert has been issued.
- creates a random behavior for the attacker while being predictable for the administrator (so that the latter can comprehend the whole intrusion process) by dynamically changing the topology and functioning parameters at runtime

2.3 Underlying technologies

The crucial requirement in any honeypot system is the ability to isolate itself either from the production infrastructure or the hosting environment, while also posing as a harmless and genuine production branch for a company and institution in order to deceive experimented attackers.

2.3.1 Docker containers

Due to the demands of simulating an entire institutional system such as a datacenter, we oriented towards Docker containers as a lightweight method of isolation and manipulation of vulnerable services exposed to malicious users.

Instead of virtualizing the underlying hardware as virtual machines do, containers benefit from a modern feature found in nowadays kernels called namespaces. These namespaces represent different resources a user would take advantage of during its interaction with a machine. By carefully “splitting” certain namespaces based on their resource nature we can achieve a certain grade of isolation from the host environment. For example, by putting our vulnerable service in a different PID namespace, we decouple the intended application from sharing the same PID interval as the rest of processes found on the host machine.

[2] https://github.com/vulhub/vulhub

2.3.2 Software-defined networks (SDN)

Instead of using classical networking protocols in a distributed manner, in which nearby routing equipment cooperate in finding the shortest itinerary, SDN use a concept of centralized decision making in regard of how every networking equipment should act for a certain packet.

By letting the administrator make their own controller in a programmatical way, we bring more flexibility and dynamicity to the networking layer of the honeypot. Perhaps, in limited situations, we are not interested in providing the best latency, parameter which classical routing algorithms strive for, but maybe in modifying certain packets in order to increase the honeypots deceiving and defensive capabilities.

While classical networking is made possible using rigid protocols (for example matching the longest destination IP prefix for forwarding or destination MAC address for switching), SDN controllers provide flexibility in the process of networking by defining a more relaxed set of rules, by combining and matching independent fields found inside packet headers, based on the logic described by the developer. These types of rules are called flows.

The contrast between classical networking pipeline and the flow matching logic can be seen inside the Linux kernel source code (routing tables have a fixed data structured assigned to them called rtable [8] while flows use flexible hash tables along with a tuple space search classifier algorithm [9]).

Tuple space search classifier enables us to build multiple matching structures which contain the indexed criterion (for example a hashed combination of the source IP address, message length and network interface card vendor's id found in the MAC address) and its assigned actions (drop or modify the packet, flood the network, forward on a certain interface). Every matching structure, called a flow table, can only index the same permutation of such fields resulting in multiple flow tables for complete flow recognition. Although it has drawback regarding intensive memory usage, the time complexity is linear depending on the number of flow tables used [9].

In figure 2 it is shown the process of traffic and network control. Whenever a routing equipment receives a packet which does not match to any registered flow, the controller intervenes by receiving the specified packet and returning a new flow rule to the managed virtual switch.

Therefore, the networking process is divided in two major components: control plane (the central logic which decides what to do with transiting packets) and data plane (the logic local to every commanded equipment which implements the action using the available hardware components).

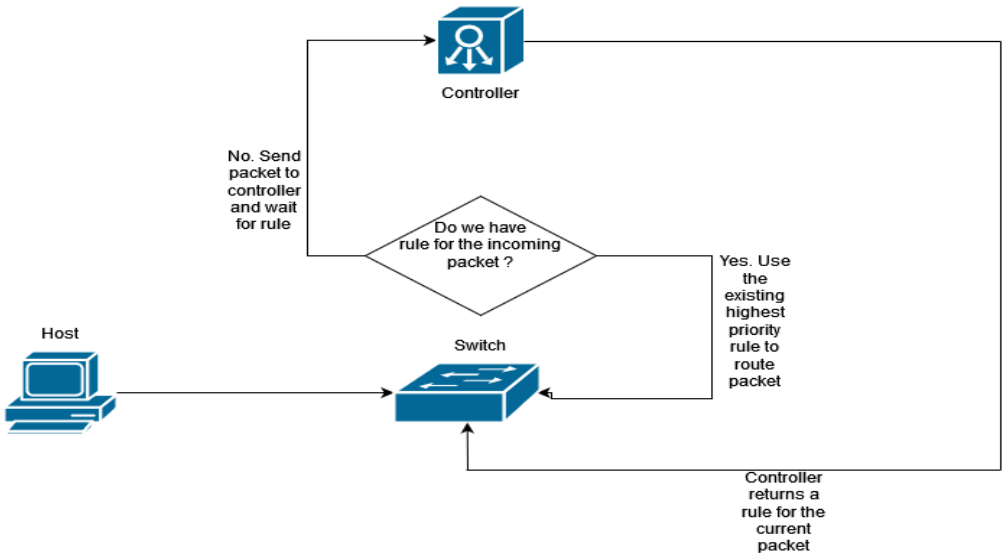

Fig. 2: interaction between SDN controller and a switch. A controller may manage multiple switches.

2.3.3 Recursively defined topologies (RDT)

It is necessary but not sufficient to integrate the honeypot along the production branch in the manner of a DMZ (de-militarized zone) in order to increase its authenticity in the eye of a skilled attacker. What is obvious in case of large companies or institutions is that the production system needs to have a massive hardware support, which means that the honeypot must simulate the underlying physical layer as well, more precisely the exact server topology.

The increasing requirements regarding scalability, availability, consistency in addition to the cost of auxiliary equipment, forced engineers to change the networking paradigm in data center environments. Instead of the classical networking approach of using complex and expensive networking equipment (128-port 10GigE switch which costs approximately 20.000$) as the backbone of routing process, the responsibility of packet switching shifted towards the servers themselves. In addition, servers would be organized into predictable and symmetric topologies in order to simplify routing and fault tolerance algorithms.

Therefore, the recursively defined topologies become a potential solution due to their properties:

- networking specialized devices are only used in local switching between servers located in the same LAN
- routing between LANs is made by servers not by routers
- a basic structure within a LAN is represented by a fixed number of servers linked by a single switch
- networking equipment can only link hosts/servers and must not link other routing devices like themselves.

Initially, we consider the basic example of a fixed number of hosts interconnected to a central switch. This type of structure is considered to be the building block for more complex RDTs and are assigned, in this field's articles, as having a grade 0 [13,14], while hosts, by our notation have a grade of -1.

By combining same structures of grade 0 in a completely connected graph we define a structure of grade 1 [6]. Repeating this process, we can build a structure of grade k by linking structures of grade k-1.

A crucial aspect of the topology is the linking process. Depending on how many available network interface cards (NIC) each host possesses, we can define the maximum number of links these hosts can maintain, thus the maximum grade of the topology. The maximum number of hosts per basic structure of grade 0 is given by the number of ports provided by the switch.

By abstracting structures of grade higher than -1 to be considered standalone servers, we define a link of grade k to be between 2 servers of grade k. For example, a link of grade -1 is a real link between 2 physical hosts intermediated by a physical switch, while a higher-grade link (grade higher than -1) is a form of a logical abstract link between higher-grade structures. The number of higher-grade links that every node is responsible for and the property of completely connected graph inside a structure [6] gives us a hint about how many structures of every grade exists and how are they organized.

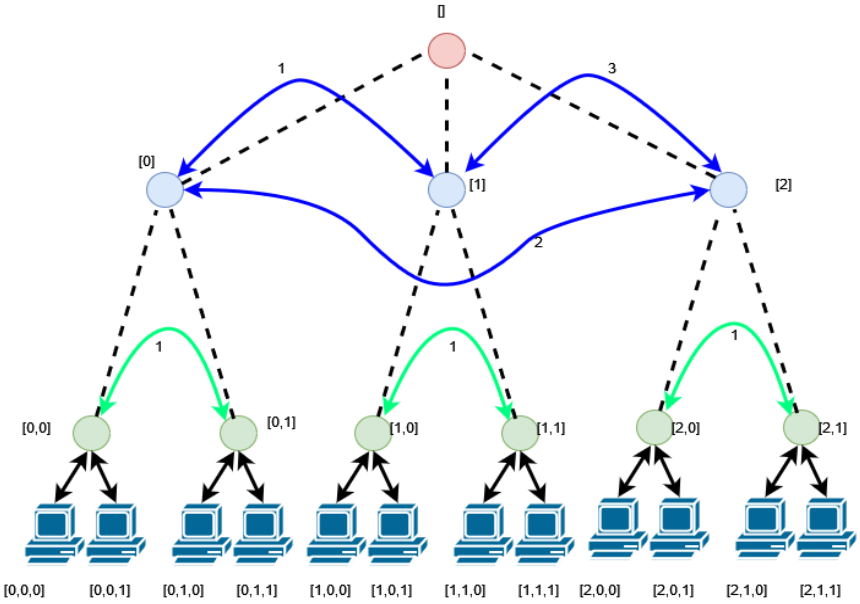

Fig. 3: A custom made recursively defined topology.

Figure 3 shows a RDT of maximum grade of 2. As observed, higher-grade links do not interconnect physical hosts directly but the abstracted nodes (both endpoints stop to the logical bound of a higher-grade structure). Links of this type are reduced using our developed algorithm in section 3.4.2, from a grade k to a grade 0, the honeypot having the further responsibility to distribute this links to physical hosts based on chosen heuristics.

There is only 1 structure which encapsulates the whole topology and represents the maximum grade achieved.

According to Li et al. [6], a recursively defined topology of type FiConn reduces the cost of switches by 80% (result concluded by the reduced number of networking equipment portrayed in (1) and (2) equations compared to hierarchical topologies such as Fat-Tree.

$$S_{FiConn} = \frac{N}{n} \quad (1)$$

$$S_{Fat-Tree} = \frac{5N}{n} \quad (2)$$

$N = \text{total number of servers}$
$S_x = \text{total number of switches in topology X}$
$n = \text{number of ports available on a switch}$

Moreover, Li et al. showed that the maximum number of servers accommodated in a Fat-Tree topology increases cubically depending on the number of hosts placed in a basic structure ($\frac{n^3}{4}$) while the results presented by Gue et al [7] proved that recursive topologies of type D-Cell provide a double exponential growth (n^{2^k}) based on the number of hosts in a basic structure (n) and the maximum grade (k).

3 Design and architecture

The solution can be decomposed in 6 independent components, which are discussed in the next sections:

- intrusion detection system (IDS)
- heuristic configuration generator (HCG)
- tracing module (TM)
- abstract recursively defined topology (ARDT)
- active virtual topology (AVT)
- central command module (CCM) which acts as both a SDN controller and the entity which decides how the honeypot should react to malicious actions.

3.1 Central command module (CCM)

We consider the central command module, the component which manages any action regarding container life cycle in addition to abstracting and changing the recursively defined topology chosen by the administrator. In Figure 4, it is described the general architecture of the honeypot.

Initially, we expose a genuine interface used by both genuine users and attackers. In the first step, the monitoring system, for example a strace[3] instance hooked to the stated service, collects system calls issued during the interaction with the user.

An intrusion detection system (IDS), which is explained in depth in section 3.2, classifies based on the fed syscalls given by the monitoring system what type of user it may be.

[3] https://strace.io/

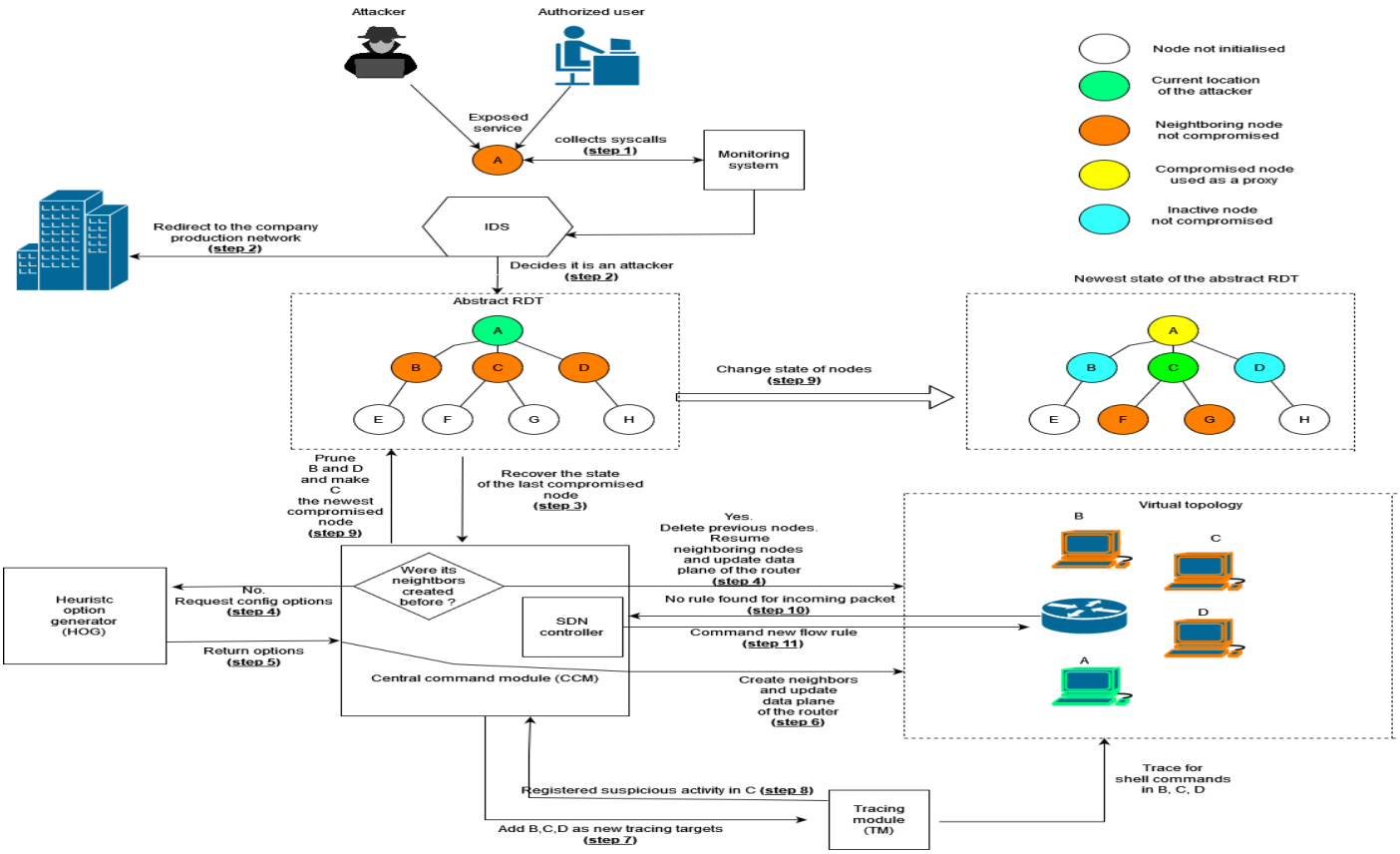

Fig. 4: Main pipeline of events

In our interest, we study the process of detecting and deceiving attackers by initiating a honeypot (Figure 4) whenever they have obtained root privileges over the container hosting the initial interface.

We memorize the state of the recursively defined topology and extract the next neighbors (if created) which are at risk of being compromised (step 3).

Afterwards, in the context of step 4, we verify if the neighboring nodes were previously generated. If not, we employ the use of the heuristic option generator to change as much as possible the pattern of generation of new challenges.

During step 4 and 5, we inform the SDN controller to change its control plane and push the new data plane to the singleton instance of an OVSwitch. In addition, instruct Containernet framework to link the neighboring nodes to the virtual switch.

The moment a new container has started we instruct the tracing module (TM) (step 7) to notify whenever a new shell command appeared in the previously generated nodes at the global level (the set obtained by the union of inactive nodes but not compromised, compromised nodes used as a proxy, neighboring nodes not compromised and the current location of the attacker). If the target is different than the current location of the attacker, we change both the state of the abstract topology representation and the virtual topology. The current location of the attacker becomes the target discovered by the tracing module while the previous location is considered as a proxy container part of a compromised chain of nodes. The neighbors of the previous node which are not adjacent to the current location of the attacker are pruned from the virtual topology and marked as inactive in the abstract counterpart (step 9). Occasionally, new flow rules are requested by the virtual switch as stated in section 2.3.2 (steps 10 and 11).

3.2 Intrusion detection system (IDS)

The first module which encounters either a threatening user or an authorized entity is represented by the IDS. This system collects the current trace of system calls issued by an entry point service (for example a web server which communicates with the possible attacker) and decides based, on a certain algorithm, if this trace differentiates from registered normal behavior traces.

To define a minimal architecture of an IDS, we consider a containerized entry point (for example a web server inside a Docker container) and the IDS command center located on the host system, which listens for incoming system calls issued by the entry point during its interaction with a user. The newly caught system call is added to the current trace of system calls and the command center decides based on the chosen detection algorithm if the behavior distances itself from a normal one. From a networking point of view, there are 2 network addresses: 1 for the honeypot and 1 for the production branch.

In order to satisfy all utilization cases in which the attacker never accesses the production branch even though it was detected, the entry point is, located, by default, inside the honeypot network. Considering the worst-case scenario, in which the attacker takes control of the system before a detection alert has been issued, would not represent a threat to our production branch. If it is a malicious user, no other actions will take place, however if it is a legitimate user, the SDN controller will make sure that the entry-point container is linked to the production branch network.

There are certain detection algorithms which represent potential candidates for the designed IDS, however there are also critical requirements that must be satisfied:

1. vulnerable element exposed to malicious users is the system owner's newly designed application, meaning that we do not know any vulnerabilities. Detection methods that use training sets must only use normal behavior traces.

2. vulnerable element is communicating remotely with a malicious user in real time, meaning that no complete behavior trace can be provided in an online environment. Detection methods must be able to detect abnormalities with the actual state of the trace during tests.

3. any new syscall caught by the aggregation system triggers a new action in discovering whether an intrusion occurred. Specifically, algorithms need to test using registered syscalls one by one or using a sliding window across an incomplete trace. Methods which utilize rigid scopes of fixed length that are mutually exclusive throughout a syscall trace are not compatible with our system. An explanation for avoiding this type of implementation can be portrayed by the situation in which the number of last syscalls from a completed user-system interaction does not make a complete set prepared for analysis. This means that the aggregating component will wait indefinitely for the remainder syscalls.

During the research phase of the project, we have identified 4 potential candidates: lookahead pairs (I), frequency-based algorithms (II), Hidden Markov Models (III) and K-Means Clustering (IV).

I. Forrest et al [3] presented in their work the concept of lookahead pair algorithm. The method is simple yet effective in high true positive rates and low false positives and it is done by traversing the current syscall trace and building a database of sequences that may appear after every distinct system call. This phase can be associated with the training stage occurring in machine learning algorithms. Unfortunately, the size of the database may represent a disadvantage since there is no parameter which decides when the database contains the complete normal behavior or not, which may result in either underfitting or overfitting problem.

A brief example of building the normal behavior database:
- let s be the current trace of syscalls: $open, read, read, lseek, read, write, exec, write\ close$
- let *p = 2* be the maximum lookahead positions considered
We can now create the normal behavior database (Table 1) querying every distinct system call.

Table 1

Database used in lookahead pair algorithm

System call	Position 1	Position 2
open	read	read
read	read lseek write	lseek read exec
lseek	read	write
write	exec close	write
exec	write	close
close	-	-

After we finished building the database, any testing syscall trace can be verified by observing its lookahead pairs and comparing it with the database. Any mismatch can increment a parameter. Whenever this parameter exceeds a certain threshold, an intrusion alert is issued.

II. The detection system designed by Abed et al [5] represents a second candidate. Firstly, they build at every training epoch a normal behavior database and compare it with the database resulted during the previous epoch. By applying a cosine similarity test to see how much the current database improved comparing to the past instances represent an objective method measuring the learning rate. The moment the similarity rises over an established limit, it is considered the moment the training ends.

Future sequences from a testing trace are verified to exist in the normal behavior database. However, if a sequence is not found, a counter is incremented and eventually the whole trace is mapped as malicious when the counter surpasses a certain threshold.

III. Warrender et al [1] propose a Hidden Markov Model (HMM) with a hidden number of states equal to the number of distinct syscalls that can appear in normal traces. Regarding observable states, their number is the same as of the hidden ones and permit us to compare the output of the machine with the current trace that we are checking. All hidden states build a complete graph in which edges have assigned a certain probability. In addition, all hidden states are linked to all observable states.

Regarding the training stage in which we calibrate the probabilities for every transition, Warrender et al. propose the usage of the Baum-Welch algorithm.

Whenever the detection system receives a syscall from the vulnerable container, it is added to the trace of observable features and calculates based on the Forward algorithm [2] the sequence of hidden states which maximize the probability of having the observed trace. If the probability is lower than a threshold, we can say that either we encountered a very rare event or an anomalous state.

IV. Usage of unsupervised learning may be favorable in situations such as the current one, when we do not have a priori knowledge on malicious events in order to label all the training data. Tunde-Onadele et al [4] propose a method of clustering frequency vectors, obtained by a fixed-length sliding window over the current trace. In their work, the K-Means Clustering offer the second highest detection rate among other algorithms, the most accurate one being self-organizing map (SOM), which unfortunately does not satisfy condition (1) stated in paragraph 3.1.1, due to the existence of anomaly neurons which are trained using malicious sets.

A brief example of clustering usage is portrayed below:

Let:

S be the distinct number of sycalls which may appear in a normal behavior trace
K be the number of centroids which define a cluster
L be the fixed-length of a sliding window
T be the current trace's length of syscalls such as $A_1, A_2, \dots A_T$

Using a sliding window of size L, we build a frequency vector of size S which contains the number of occurrences of syscalls found inside the sliding window in the context of all S distinct syscalls.

The frequency vectors may be sparse if $L \ll S$.

Afterwards, the frequency vectors are used as coordinates in a S-dimensional space in order to build points which are clustered based on their Euclidean distance from the K randomly picked centroids.

Tunde-Onadele et al. stated that whenever an anomaly occurs, the frequency vectors will create numerous clusters containing a modest number of members. By carefully choosing a threshold of minimum number of members for normal behavior clusters we can identify malicious or out of the ordinary actions.

3.3 Tracing module (TM)

The concept of dynamicity, in the context of our honeypot, is referring to the ability of building new “challenges” for the attacker based on previous data collected (preferences for a particular type of application, duration of the attack etc.). In order to expand the frontier of new vulnerable nodes we need to permanently know the last node, which was compromised by the attacker, so that we build appropriate neighbors based on the above specified metrics.

We consider the worst-case scenario in which, the attacker took control of the entire container, by exploiting the deployed service and acquiring root privileges. The common thing which reunites all these types of attacks of privilege escalation is the final result – the ability to interact with a shell. Instead of monitoring and manually calibrating intrusion systems to recognize 0-day exploits, we can monitor the command history of the shells opened inside the container.

Our solution consists in forcing every existing and new shell session from the affected container to append the command issued to them in a global logging file. In other words, every shell has a shared history. Configuring this type of behavior can be done by changing the default “.bashrc” file which contains all the rules that are applied whenever a new session is spawned. Figure 5 contains an example of a bash configuration file which satisfies the above requirements.

```
export HISTCONTROL=ignoredups:erasedups  # no duplicate entries
export HISTSIZE=100000                   # big big history
export HISTFILESIZE=100000               # big big history
shopt -s histappend                      # append to history, don't overwrite it

# Save and reload the history after each command finishes
export PROMPT_COMMAND="history -a; history -c; history -r; $PROMPT_COMMAND"
```

Fig. 5: bashrc configuration file example

Due to containers not sharing the “mount” namespace with the host operating system, we need to map the logging file within the container with an existing file residing in the host environment. This is made possible with Docker volumes, a method which automatically manages to save chosen container data inside the host without being limited by the type of filesystem used.

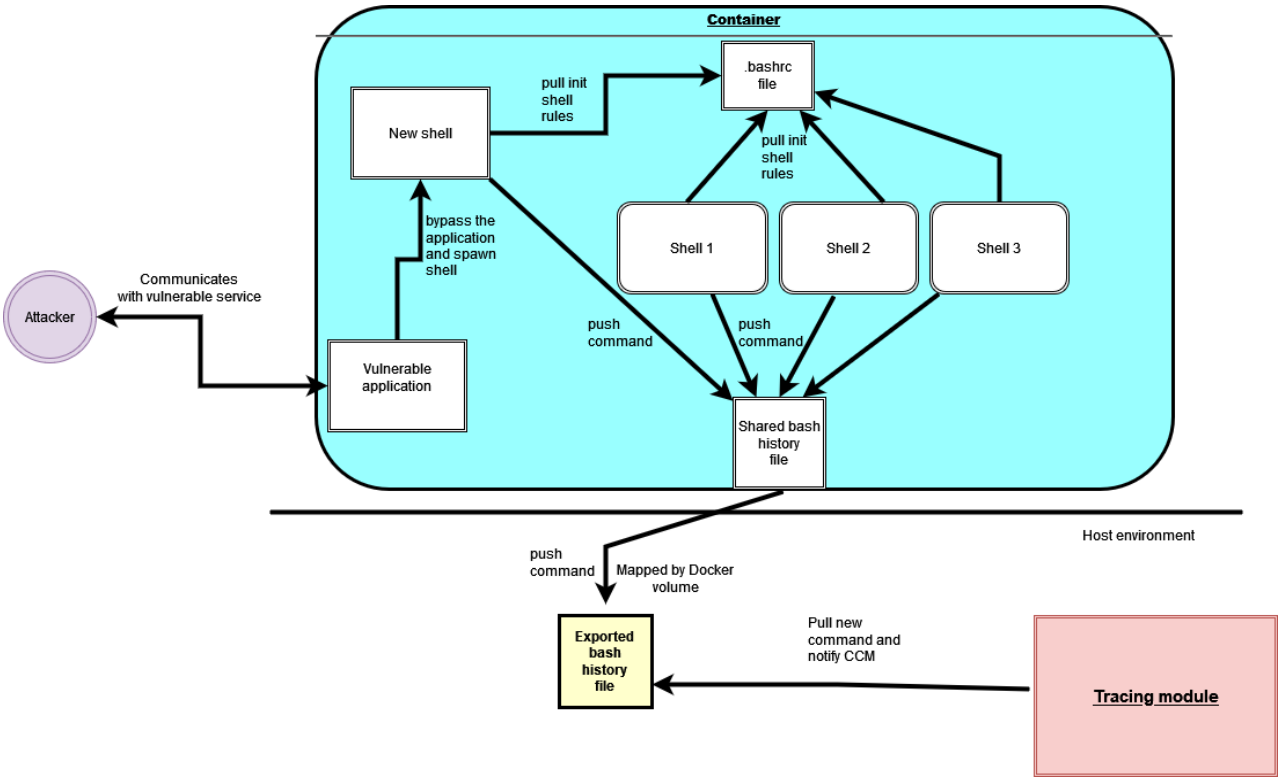

Fig. 6: Process of tracing a single container by our module.

This method of tracing is fool proof even when the attacker hides their activity by disabling the shared history file. Deleting it requires a single shell command which is recorded by the tracing module. The only issue is that it decreases the quantity and quality of observed data in order to predict attacker's behavior.

3.4 Abstract recursively defined topology (ARDT)

This module manages the initialization of the logical representation of the topology and informs the central command module (CCM), during the lifetime of the honeypot, about the neighbors of the attacker's current location in terms of the 5 states, found in Figure 4. This representation has only management purposes and enables the CCM to view what resources are not yet initialized (containers that were not compromised, high-order links not distributed to physical hosts etc.) in order to request additional resource parameters from the heuristic configuration generator (HCG).

3.4.1 Representing recursively defined topologies

In order to rigorously define a RDT, we have identified similarities between various implementations such as DCell[7] and FiConn[6]. In case of Dcell of maximum grade k, every physical node (grade -1) needs to have 1 link of every grade from 0 up to k-1.

Meanwhile, FiConn does not impose a strict number of links on every physical node, but rather on the total links a basic structure of grade 0 manages. For example, for a FiConn with $n = 4$ hosts per basic structure and maximum grade $k = 2$ we have 4 links of grade -1, 2 links of grade 0 and 1 link of grade 1 going out of a basic structure. The distribution of those links between the hosts contained in the basic structure is the honeypot's responsibility, but with the promise of preserving the same number and nature of links among all basic structures of grade 0.

We can define a RDT as a tuple (N, K, c, f_c) where N represents the number of hosts found in a basic structure of grade 0, K is the maximum grade of the RDT, c is the maximum grade which imposes the same number and nature of the links for every node of grade c and f_c being a function which states how many links of every grade a node of grade c must have (equation (3)).

$$\mathrm{f_c}: \mathbb{N} \rightarrow \mathbb{N}, \mathrm{f_c(x)} = \begin{cases} > 0, x < K \\ 0, x \geq \mathrm{K} \end{cases} \quad (3)$$

The grade of every node suggests us the hierarchical aspects of an RDT. For our implementation we used a balanced tree.

First of all, before building the abstract RDT tree we need to know how many nodes of every grade compose the next higher-grade node (equation (4)).

$$\mathrm{H}: \mathbb{N} \rightarrow \mathbb{N}, \mathrm{H(x)} = \text{how many hosts of grade x} - 1 \text{ make a node of grade x} \quad (4)$$

Given the property of completely connected graphs having $x + 1$ vertices when each vertex has an order of x edges, we can conclude the following:

$$H(x) = L_{x-1} + 1 \quad (5)$$

where L_{x-1} represents the total number of links of grade $x - 1$ imposed by all nodes of grade c residing inside a node of grade $x - 1$.

Concluding, the $H(x)$ can be calculated by the following:

$$H(x) = f_c(x) \prod_{x-1}^{i=c} H(i-1) + 1 \quad (6)$$

Figure 7 shows a Python implementation example for calculating $H(x)$ in a bottom-up dynamic programming approach only for recursively defined topologies where $c \in \{-1,0\}$, as these types of networks have practical purposes, at the time of writing this work.

```
def coreAlgo(maxHostsPerLevel : List[int], maxGrade : int, constraints : List[int], constraintGrade : int):

    previousProduct : List[int] = [1 for _ in range(len(maxHostsPerLevel))]
    if constraintGrade == -1:
        for i in range(len(constraints)):
            constraints[i] = constraints[i] * maxHostsPerLevel[0]

    maxHostsPerLevel[1] = constraints[0] + 1
    previousProduct[1] = maxHostsPerLevel[1]
    for grade in range(2, maxGrade + 1):
        maxHostsPerLevel[grade] = constraints[grade - 1] * previousProduct[grade - 1] + 1
        previousProduct[grade] = maxHostsPerLevel[grade] * previousProduct[grade - 1]

    # one massive structure of grade maxGrade + 1 which incorporates the whole topology
    maxHostsPerLevel.append(1)

    return maxHostsPerLevel
```

Fig. 7: implementation of the algorithm used to find the number of nodes needed to form the next grade node

3.4.2 Tree representation of the RDT

Following the calculation of H we build the balanced tree representing the topology in 4 distinct phases:

Initialization: Initialize all nodes in top-down manner and link same grade nodes inside each parent node in a completely connected graph

Propagation: In a top-down manner, equally distribute for every node with its grade higher than 0, the previously created links to their respective children

Collapse: prune the abstract tree by only preserving the root node and nodes with grade 0 and -1

Inheritance: Whenever we need to initialize a new physical node contained in a different grade 0 node than the attacker's current location, we request the HCG to announce the physical node what links it needs to "inherit" from its grade 0 parent (equal distribution of links between grade 0 and grade -1 happens if only if constraint grade c is -1).

During the initialization phase, we attribute to every node of grade x a unique sequence of numbers,

$$\mathrm{S} = \{\mathrm{a_{K-1}a_{K-2}} \dots \mathrm{a_x}\} \tag{7}$$

where every element of sequence S represents its position regarding every ancestor.

$$\mathrm{a_y} \in \{0, \mathrm{H(y)} - 1\}, \exists\ \mathrm{y}\ \in \{-1,\ \mathrm{K} - 1\} \tag{8}$$

Afterwards, we build the links for every grade x, except K, in order to achieve completely connected graphs.

Let:

$\mathrm{S} = \{\mathrm{a_{K-1}a_{K-2}} \dots \mathrm{a_{x+1}a_x}\}$,
$\mathrm{S'} = \{\mathrm{a_{K-1}a_{K-2}} \dots \mathrm{a_{x+1}b_x}\}$,

2 distinct nodes of same grade. Then :

$$(\mathrm{S}, \mathrm{S'}) \text{ is a link } \Leftrightarrow 0 \leq \mathrm{a_x} < \mathrm{b_x} \leq \mathrm{H(x)} - 1, \exists\ \mathrm{x}, -1 \leq\ \mathrm{x} < \mathrm{K} \tag{9}$$

We continue with the propagation phase which is also the most complex out of all 4 stages. After the first step presented above, every node should have a collection of links with its siblings.

In order to present the algorithm of propagating the links towards every child (Fig. 8) we need to define several terms:

$\emptyset$ as the root of the ARDT (10)

L_S^x as set of ordered links of grade x of a node identified by the sequence S (11)

$T_S = \{L_S^x |\ \mathrm{K} > \mathrm{x}\ \geq \mathrm{K} - |\mathrm{S}|\}$ as the set of all links inherited or owned by S (12)

$G_S = K - |S|\ as\ grade\ of\ the\ node\ S$ (13)

$\mathrm{C_S} = \{\mathrm{S}\ \cup\ \{\mathrm{x}\} \mid 0\ \leq \mathrm{x} < \mathrm{H(G_S)}\}$ as the children of node identified by S (14)

$P_S = \left\{p_y^x \;\middle|\; 0 \leq y < H(G_S),\ x\ \geq K - |S|,\ \left|p_y^x\right| = \frac{|L_S^x|}{|C_S|}\right\}$ as set of partitions from $\mathrm{L_S^x}$ equally distributed between children of S (15)

$p_y^x\ as\ partion\ of\ links\ of\ grade\ x\ inherited\ by\ S \cup \{x\}\ (the\ x-th\ child\ )$ (16)

```
append ∅ to queue
while queue not empty
{
        S ⟵ pop from queue
        if G_S ≤ 0
            goto next
        if S = ∅
            goto next
        for every L_S^x in T_S
            for every p_y^x in P_S
                for every link (S, S') in p_y^x
                    (S, S') ⟵ (S ∪ {y}, S')
                    T_{S ∪ {y}} ⟵ T_{S ∪ {y}} ∪ (S ∪ {y}, S')
next:
        for every child in C_S
            append child to queue
}
```

Fig. 8: Pseudo-code implementation of the propagation phase

Continuing with the collapse phase, we update the children of the root to be all nodes of grade 0, resulting in the elimination of the nodes with grades between 1 and K - 1 which were used as helper vertices in the process of propagation. In this moment, all links are between nodes of grade 0 and are ready to be distributed to physical hosts using the heuristic provided by the HCG.

Regarding the inheritance phase, additional information is provided in section 3.6 due to its implementation being associated with other subsystems such as the AVT.

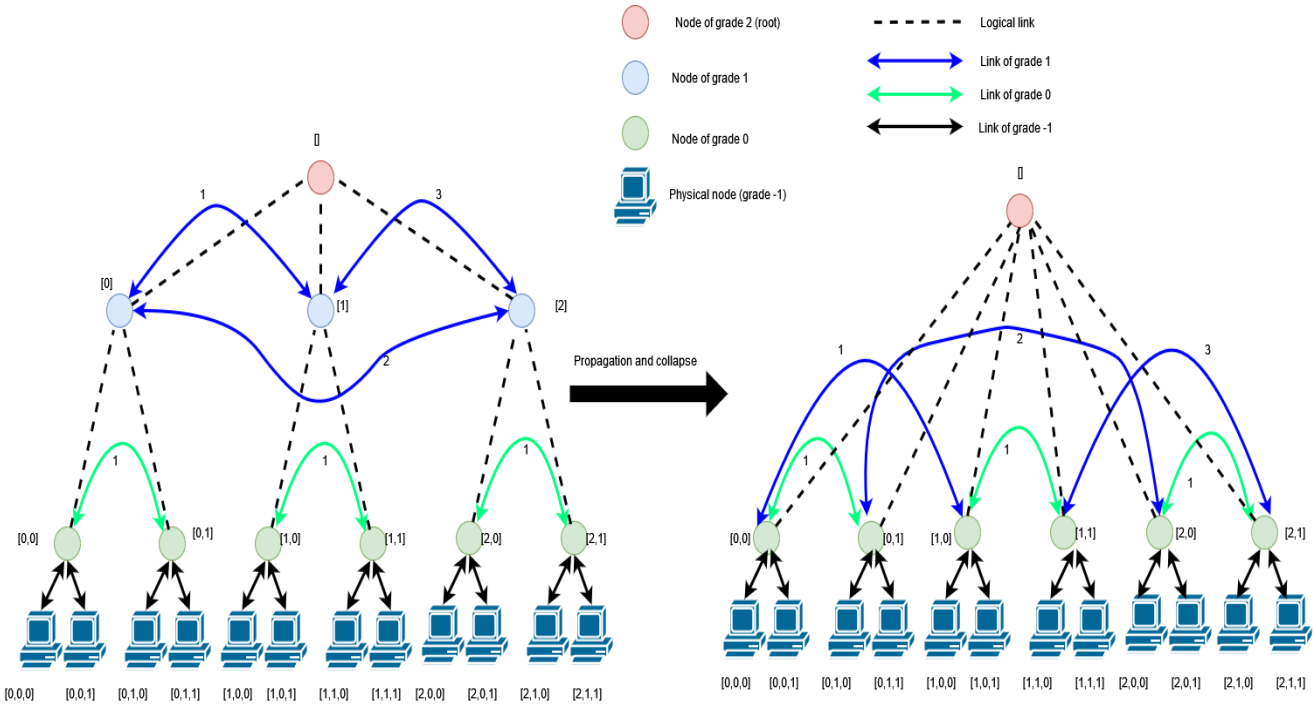

Fig. 9: ARDT transformation after the propagation and collapse phases

Figure 9 portrays an example of the propagation and collapse phases on a recursively defined topology where $(N = 2, K = 2, c = 0, f_c(0) = 1, f_c(1) = 1)$. As one can observe, every node is assigned with a unique sequence S, while the links can be put into 2 categories: logical links which are only used in managing the hierarchical levels, ensuring that we are correctly traversing the ARDT and graded links which represent future networking bonds between vertices.

Initially, we define the sets L_S^x of all nodes that are prone to the propagation phase such as: $L_{[0]}^1 = \{([0],[1]),([0],[2])\}$, $L_{[1]}^1 = \{([0],[1]),([1],[2])\}$, $L_{[2]}^1 = \{([0],[2]),([1],[2])\}$. Due to every node of grade 1 having 2 links of grade 1 and 2 children, results in every child of grade 0 inheriting only 1 link, where the order in the set is equal to the order of the child. Before the inheritance phase, graded links are not physically supported due to the abstraction of higher-grade nodes as physical hosts. We can consider the high-grade links propagated until grade 0 nodes to be physically supported as networking links the moment the CCM requests from the HCG to compute a distribution of said links towards physical nodes (inheritance phase). As a particular case, equal distributions similar to the ones during the propagation phase, can only occur when $c = -1$. A special status is reserved for links of grade -1, as they are not representing direct network links between hosts. These links are intermediated by a switch in order to respect one of the requirements of recursively defined topologies.

3.5 Heuristic configurations generator (HCG)

Before discussing the process of building the virtual topology using the RDT, we need to understand the usage of the HCG which provides a wide variety of configuration parameters regarding containers and the overall topology. Its purpose is to bring a dynamical behavior either for obfuscating recognition methods used to detect honeypots [12] or to adapt to the intruder's behavior.

Regarding the design of this module, we consider using as a starting base the strategy design pattern to decouple the central management from the generation algorithm. Moreover, we also provide flexibility in changing different implementations throughout the runtime of our honeypot.

In order to provide reusability and easier modification for further features, we have identified 2 use cases regarding generation: independent heuristics for options of different components (containers, network links, virtual switch) and dependent heuristics between components.

The first use case can be easily solved by building distinct default strategy classes for every identified component and extending their functionality through inheritance. For example, we can define default parameters for every managed entity (for links we can choose bandwidth, delay; for containers we require the Docker image, starting IP address, upper-grade links inheritance explained in section 3.4.2) and then create subclasses specialized in providing at least the required default options.

However, there are some scenarios in which configuration parameters of a component are strongly dependent on their counterparts from a different entity (name of the container may need to include the name of its neighboring switch). In addition, the lazy evaluation forces us to compute those options in the moment of their utmost need, forbidding us to initially generate all of them and be stored for future use. In order to solve this issue, honeypot dependent strategies designed for different components are contained inside same strategy class and are generated in different phases throughout the runtime cycle. As an example, we always need to create physical nodes before requesting options for links.

```
class CombinedOptionGenerator(OptionGenerator):

    NAME_IDX = 1
    SWI_IDX = 0

    def __init__(self, seed : int = 1, ip: int = 1, globalIp : str = '0.0.0.0'):
        self.globalIp = globalIp
        self.mask = mask
        super().__init__()

    @OptionGenerator.registerOption('link', 'upperLinks')
    def genLinks(self, *args, **kwargs) -> List[List[HoneyLink]]:
        pass


    @OptionGenerator.registerOption('node','name')
    def genNameContainer(self, *args, **kwargs) -> str:
        pass


    @OptionGenerator.registerOption('node', 'ip')
    def genIpForLevel(self, *args, **kwargs) -> str:
        pass

    @OptionGenerator.registerOption('node', 'dimage')
    def genImageDocker(self, *args, **kwargs) -> str:
        pass

    @OptionGenerator.registerOption('switch', 'name')
    def genSwitchName(self, *args, **kwargs) -> str:
        pass
```

Fig. 10: Python skeleton implementation of a class with dependent strategies between nodes, switch and links

Figure 10 is an example of such class which incorporates behaviors of multiple honeypot submodules. Considering that the complete configuration of a component requires multiple named options, we designed an elegant way of attributing each named option a method. Each such method is decorated in order to instruct the Python interpreter to only call named option methods assigned with the requested phase sent by the CCM. The first parameter of the decorator represents the generation phase, while the second represents the name of the option.

```
def requestOptions(self, phase : str, *args, **kwargs):
'''
Method used to partially build a new HoneyTree leaf node which contains a container host

For changing the default behavior it is advised to overwrite it

Default behavior: calls all registerd option generators and returns the dictionary of options
where the key is the option name received as a parameter in "registerOption" decorator method
and value being the result returned by the registered method
'''
        opts = {}

        if phase not in self._aprovedOptions:
            logging.error(f'Phase {phase} not found')
            exit(1)


        for optionName, optionGen in self._aprovedOptions[phase].items():
            opts[optionName] = optionGen(self,*args, **kwargs)

        return opts
```

Fig. 11: method used to obtain the complete behavior of a component based on a certain phase

The result of such configuration behavior (Figure 11) is represented by a Python dictionary where the key is the option name provided as the second parameter of the decorator while the value is the result returned by the corresponding method. This solution can only be possible due to the Python's interpreter having access to each class metadata, more specifically its methods, through the process of reflection. Helper methods which do not provide a certain option should not be annotated with the previously mentioned decorator.

If the honeypot is mainly for research purposes, it is not recommended to use nondeterministic heuristics, due to making impossible remaking the entire attack sequence at latter times by the researcher. In order to provide deterministic behavior for the administrator and unpredictability for the attacker, it is recommended to use pseudo-random algorithms with a known private seed.

3.6 Active virtual topology (AVT)

The whole purpose of the ARDT is to provide to the CCM the global state of the topology, optimizing the resource usage by having an in-memory representation rather than building and emulating in real time all the networking components.

Due to our intention of simulating entire datacenters, having at our disposal only 1 physical host, we decided on a “lazy evaluation” paradigm of creating and maintaining only the current attacker’s location together with neighboring hosts either from the same basic structure or from inherited high-order grades. In addition to these types of nodes we are forced to also maintain previously compromised containers which are used as proxies by the intruder according to the use cases mentioned in section 4. If any proxy is pruned, the existing networking tunnel towards the attacker’s current location will cease to exist.

The virtual topology consists of an OpenFlow Virtual Switch, a SDN controller and the previously mentioned type of nodes.

As an example, we consider the topology portrayed after the propagation and collapse phases from Figure 9. For testing purposes, the heuristic which distributes higher-grade links towards physical nodes is of type “first come first served” meaning that all of such links in the node of grade 0 are given to the first initialized node contained in the respective basic structure. Supposedly [0,0,0] and [0,1,0] were previously compromised in this order. Current location of the intruder is [0,1,0] while the neighboring nodes are [0,1,1] (link of grade -1), [0,0,0] (link of grade 0), [2,0,0] (link of grade 1). Figure 13a shows the correlation between ARDT and the AVT.

Let us assume that the attacker successfully compromises the node [2,0,0]. We can now prune the nodes [0,1,1] as this node was not previously compromised and does not have any direct link towards [2,0,0]. We further initialize node [2,0,1] and node [2,1,0] which inherits all links from [2,1]. Now the state of the AVT can be seen in Fig. 13b. Regarding the pruning method, we use the feature of pausing [10] the container using the cgroup freezer kernel mechanism [11].

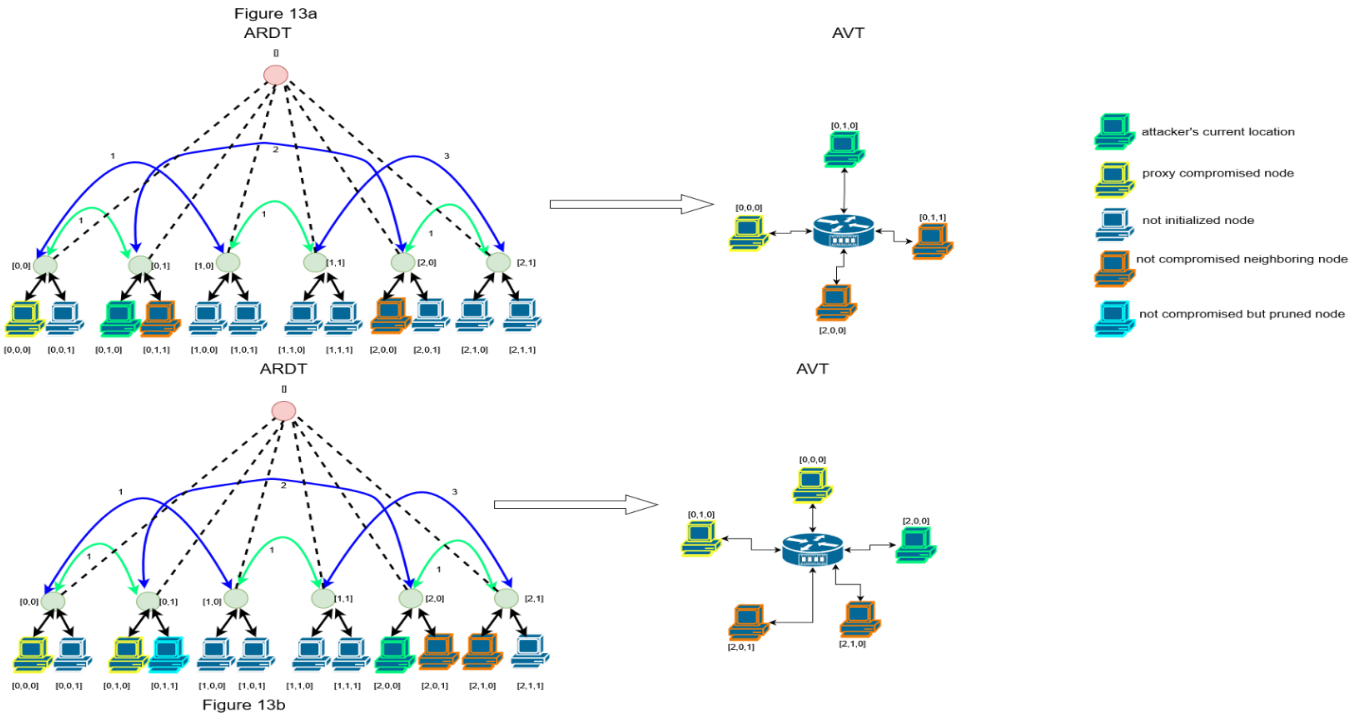

Fig.13: Relation between ARDT and AVT

4 Testing and evaluation

In order to verify the implementation, we designed a test case situation in which all containers are based on a vulnerable web server image that can allow a reverse shell exploit. The attacker accesses a vulnerable URL and successfully runs the command given as a parameter. Thus, by running the command, which issues a connection to the attacker's current location, we are able to spawn a new shell in the targeted container. The targeted container now becomes the newest location of the attacker while the previous one is used as a proxy. The attacker replicates the same attack and results in a sequential chain of compromised containers.

Figure 14 shows the CPU usage of a container in idle state. Fluctuations in core usage can be explained by either container's runtime environment manager or the webserver itself.

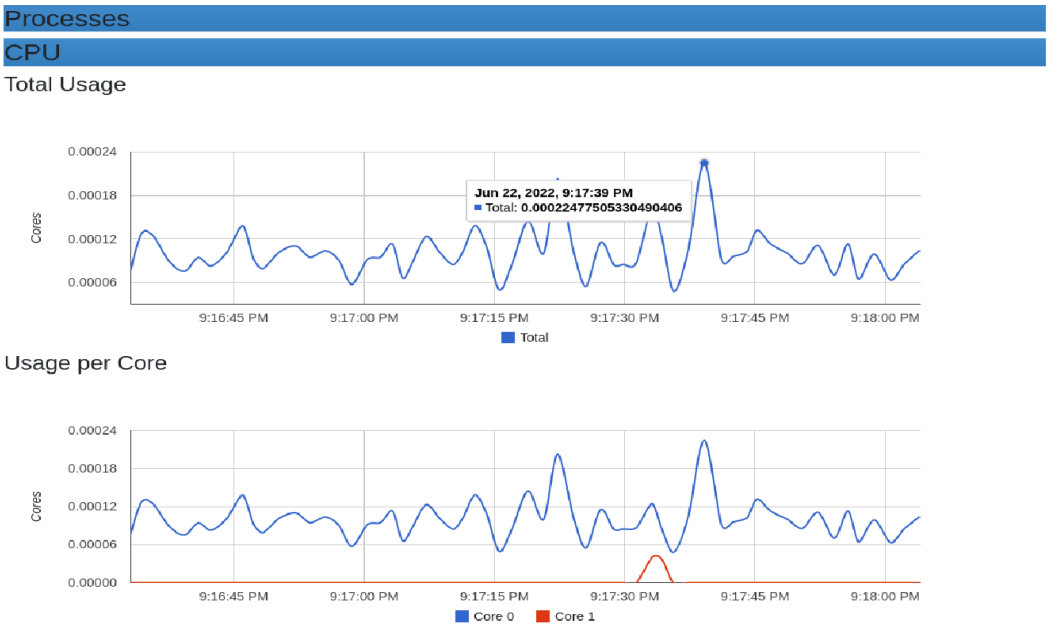

Fig. 14: Container CPU usage in idle state

The first compromise action issued in a new container can also be identified by CPU usage as portrayed in Figure 15:

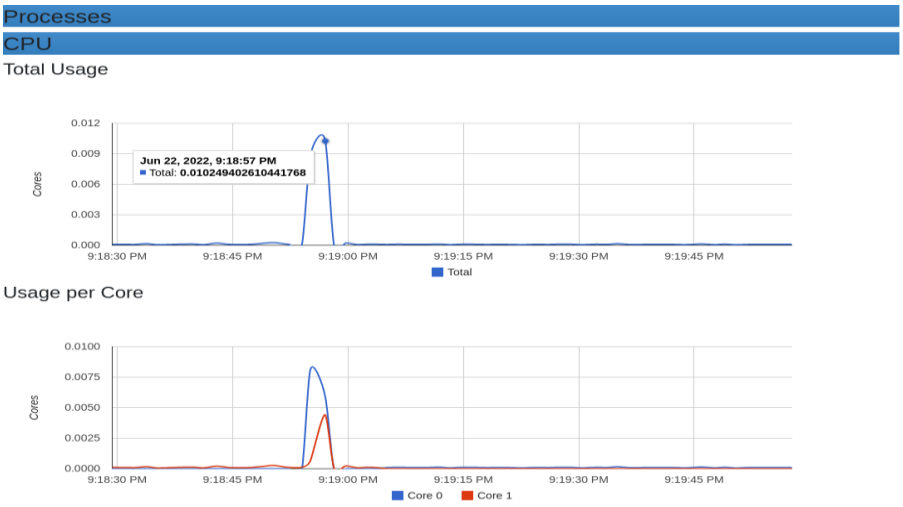

Fig. 15: Container CPU usage during an attack

As we can observe, the CPU usage peak in the case of an attack is 50 times higher than in idle state.

The attacker now, wants to compromise a neighboring container and issues a reverse shell exploit.

Figure 16 shows the network traffic from the first compromised container towards the potential target while Figure 17 shows the incoming traffic from the target's point of view.

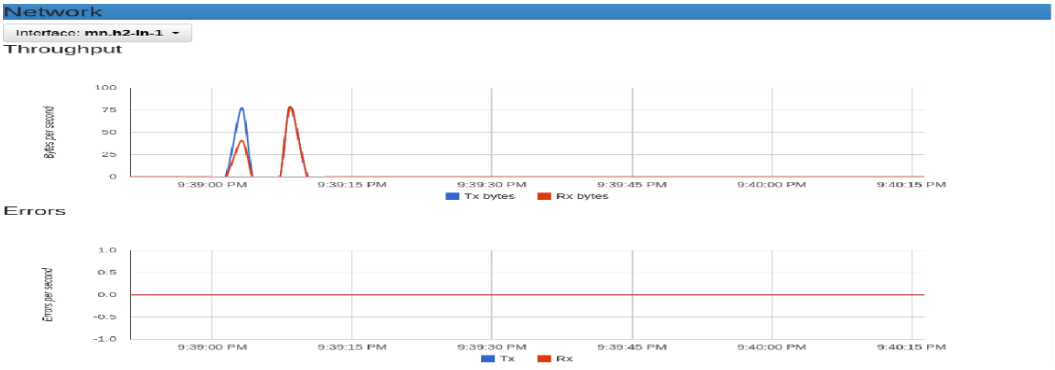

Fig. 16: Network traffic from proxy to target

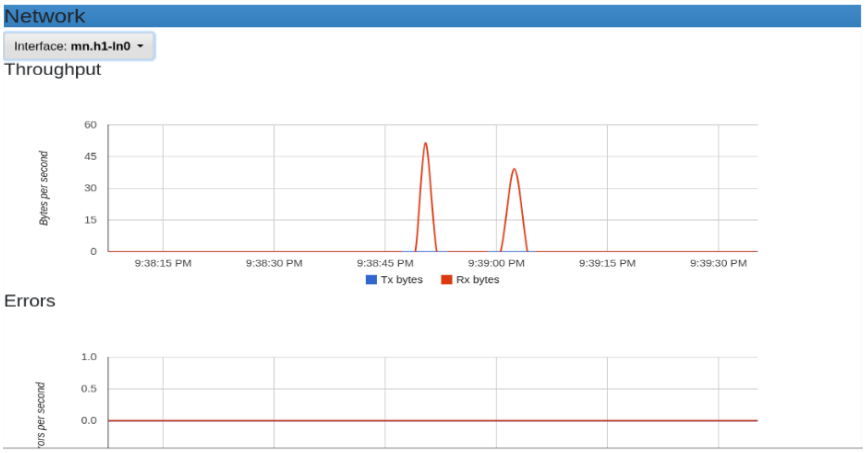

Fig. 17: Network traffic from target to proxy

As we can see no networking errors have appeared and proves that both containers communicate through the virtual switch commanded by the SDN controller.

In addition, we tested our minimal pruning subsystem which pauses the non-neighboring containers (Figure 18 proves that after the attacker scanned the target and decided not to attack it, the CPU quota assigned to the respective container is put to zero).

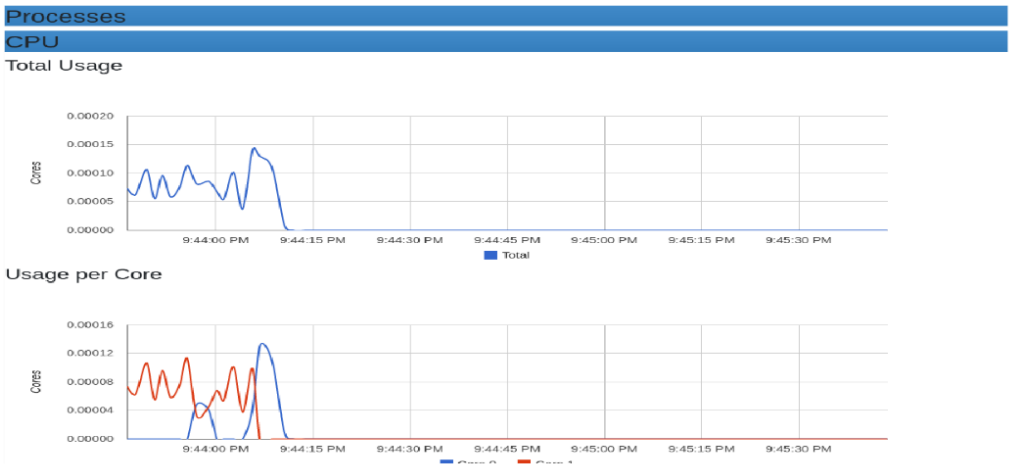

Fig. 18: Container CPU usage during pruning

Our bench-marking experiments regarding the ARDT construction have highlighted the maximum size of the representation. Regarding the hosting environment, we are using Ubuntu 20.04 virtual machine with Intel I5 as CPU and 6020MB RAM memory. The RDT studied minimizes the number of physical hosts while preserving all links with grades ranging from -1 up to K - 1 and is represented by N=2, c=0 and $f_c(x) = 1, \exists\, x < K$, where K is variable. Table 2 shows the time duration and memory allocated in order to build the ARDT.

The number of hosts is calculated using the below formula:

$$\text{Hosts} = \prod_{K}^{i=0} H(i) \tag{17}$$

Table 2

Time and space metrics in relation to maximum grade

K	Hosts	Memory (KB)	Time (s)
2	12	21076	0.66
3	84	21236	0.96
4	3612	27916	1.11
5	6526884	-	*≥300

As we can observe, the maximum grade that we can achieve is $K = 4$ and is explained by the exponential growth of the total number of potential physical hosts.

Our implementation achieves significant results in effortless configurability regarding networking topologies. Current solutions such as HoneyD[4] provide difficult configuration requirements for optimal use. In Figure 19a it is presented the minimal requirements for simulating the maximum supported ARDT presented in Table 2 while on the right side we have a configuration file used by HoneyD to achieve a much smaller topology. Difference in both quantity and syntax can be easily observed.

```
{
    "constraints" :
    {
        "grade" : 0,
        "links" :
        {
            "0" : 1,
            "1" : 1,
            "2" : 1,
            "3" : 1,
            "4" : 1
        },

        "numHosts" : 2
    },

    "debug":
    {
        "linkHistory" : true
    },

    "optimizations":
    {
        "enableCollapse" : true,
        "enablePrunePause" : true
    }
}
```

Fig. 19a: Proposed JSON configuration file format

```
route entry 10.0.0.1 network 10.0.0.0/8
route 10.0.0.1 link 10.0.0.0/24
route 10.0.0.1 add net 10.4.0.0/14 tunnel "thishost" "honeyd-b"
route 10.0.0.1 add net 10.1.0.0/16 10.1.0.1 latency 55ms loss 0.1
route 10.0.0.1 add net 10.2.0.0/16 10.2.0.1 latency 20ms loss 0.1
route 10.0.0.1 add net 10.3.0.0/16 10.2.0.1 latency 20ms loss 0.1
route 10.1.0.1 link 10.1.0.0/24
route 10.2.0.1 link 10.2.0.0/24
[...]
route 10.2.0.1 add net 10.3.0.0/16 10.3.0.1 latency 10ms loss 0.1
route 10.3.0.1 link 10.3.0.0/24
route 10.3.0.1 add net 10.3.1.1/24 10.3.1.1 latency 10ms
route 10.3.0.1 add net 10.3.240.0/20 10.3.240.1 latency 5ms
route 10.3.1.1 link 10.3.1.1/24
route 10.3.240.1 link 10.3.240.0/20
route 10.3.240.1 add net 0.0.0.0/0 10.3.0.1 latency 40ms loss 0.5
[...]
bind 10.2.0.243 to fxp0
bind 10.3.1.15 to fxp0
```

Fig. 19b: HoneyD configuration file

[4] http://www.honeyd.org/general.php

5 Further work

As one can observe the pruning subsystem lacks the power to avoid IO bottlenecks, memory usage and PID consumption. At this moment, we have only achieved optimization of the CPU usage by pausing the containers. We are working for a method to stop and delete them when they are not necessary. Our proposal would be to store locally the changes appeared in a container's private filesystem. Along with these changes (deltas) and the base starting image, we can compute anytime its actual state of filesystem. In the eyes of the attacker, he would not know that a previous target of theirs has been deleted while for the hosting environment we only archive changes made by the attacker. In addition to memory saving, we also avoid PID consumption.

As an additional objective, we need to improve our SDN controller to respond to unexpected events such as scenarios in which the honeypot is further used in malicious DDOS attacks. If such measures are not taken, the owner of the honeypot is prone to paying reparations in court to potential further victims.

Furthermore, we are also working in building the ARDT in a lazy evaluation paradigm similar to option generation and container initialization presented in previous chapters. Thus, we will be able to simulate larger recursively defined topologies without massive and sudden memory consumption.

Lastly, we are experimenting with advanced deceptive techniques in order to increase the honeypot's authenticity in the eyes of experimented attackers.

6 Conclusions

Our implementation accomplishes moderate achievements regarding hardware resource optimization and deceptive techniques. However, the most important feature is being able to easily integrate recursively defined topologies and SDN infrastructure, two key concepts which at the moment of redacting this paper are still being studied.

In addition, we have designed an efficient way to build and represent, recursively defined topologies, from a mathematical point of view, using the 4 phases described in section 3.4.2.

Due to its effortless initialization and flexible architecture, our implementation may thus represent a promising framework for developing advanced honeypots used in studying malicious behavior from the perspective of data science or game theory.

B I B L I O G R A P H Y

[1]: Warrender, C., Forrest, S. and Pearlmutter, B., 1999, May. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)* (pp. 133-145). IEEE.

[2]: https://people.csail.mit.edu/rameshvs/content/hmms.pdf

[3] : Forrest, S., Hofmeyr, S.A., Somayaji, A. and Longstaff, T.A., 1996, May. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy* (pp. 120-128). IEEE.

[4]: Tunde-Onadele, O., He, J., Dai, T. and Gu, X., 2019, June. A study on container vulnerability exploit detection. In *2019 IEEE International Conference on Cloud Engineering (IC2E)* (pp. 121-127). IEEE.

[5]: Abed, A.S., Clancy, T.C. and Levy, D.S., 2015, December. Applying bag of system calls for anomalous behavior detection of applications in linux containers. In *2015 IEEE Globecom Workshops (GC Wkshps)* (pp. 1-5). IEEE.

[6]: Li, D., Guo, C., Wu, H., Tan, K., Zhang, Y. and Lu, S., 2009, April. FiConn: Using backup port for server interconnection in data centers. In *IEEE INFOCOM 2009* (pp. 2276-2285). IEEE.

[7]: Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y. and Lu, S., 2008, August. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication* (pp. 75-86).

[8]: https://cs.dartmouth.edu/~sergey/me/netreads/path-of-packet/netLec.pdf

[9]: Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P. and Amidon, K., 2015. The Design and Implementation of Open {vSwitch}. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)* (pp. 117-130).

[10]: https://docs.docker.com/engine/reference/commandline/pause/

[11]: https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt

[12]: https://honeyscore.shodan.io/