HoneyMix: Toward SDN-based Intelligent Honeynet

Wonkyu Han, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn
Arizona State University
{whan7, zzhao30, doupe, gahn}@asu.edu

ABSTRACT

Honeynet is a collection of honeypots that are set up to attract as many attackers as possible to learn about their patterns, tactics, and behaviors. However, existing honeypots suffer from a variety of fingerprinting techniques, and the current honeynet architecture does not fully utilize features of residing honeypots due to its coarse-grained *data control* mechanisms. To address these challenges, we propose an SDN-based intelligent honeynet called HoneyMix. HoneyMix leverages the rich programmability of SDN to circumvent attackers' detection mechanisms and enables *fine-grained* data control for honeynet. To do this, HoneyMix simultaneously establishes multiple connections with a set of honeypots and selects the most desirable connection to inspire attackers to remain connected. In this paper, we present the HoneyMix architecture and a description of its core components.

Keywords

Software-defined Networking; Network Function Virtualization; Honeynet; Honeypot

1. INTRODUCTION

Honeypots [29], as a form of electronic baits, are built to intentionally expose vulnerable resources to attackers so as to encourage probing and exploiting. Because the key objective of honeypots is to learn about attackers' behaviors and capture new types of malware, honeypots do not have to implement all functionalities of a production system. Consequently, honeypots usually emulate or simulate certain systems and services to reduce the computational and maintenance cost.

However, emulator-based honeypots, including operating system and services, can be easily fingerprinted. For example, recent research efforts [1] revealed that an SSH honeypot called Kippo [7] always returns a hardcoded timestamp when attackers access its system information using the simple Linux command `uname -r`. Another honeypot called Kojoney [9], which also offers an SSH service, only needs a slightly different detection method, since it returns the timestamp when Kojoney was installed. Using such detection techniques, attackers can easily identify honeypots and behave differently afterwards. Therefore, techniques that prevent attackers from detecting the existence of emulated system and services are imperative in building effective honeypots.

Honeynet [23, 30], which is a network of honeypots, inevitably poses the same problem of honeypots themselves. Moreover, honeynets must control incoming and outgoing data in the network and aggregate captured data from different honeypots. For example, the third generation (Gen-III) honeynet [28, 13] employs a customized firewall called *honeywall* as the gateway of the network to realize better control on inbound/outbound traffic. Honeywall runs in layer-2 bridge mode to hide its existence and monitors all incoming and outgoing traffic. It is also used to contain and limit the large volume of outbound traffic generated by compromised honeypots (e.g., when used in a DDoS attack). However, Gen-III architecture cannot fully support today's heterogeneous services in honeynet due to its coarse-grained data control. For example, let us assume that honeypot A exposes XSS vulnerability while running a fake HTTP service, and we want to deploy a new honeypot B that emulates SQL injection vulnerability over HTTP. Since conventional architecture only allows one service to interact with attackers at any given time, opportunity for collecting SQL injection attack (or XSS attack) is inevitably restricted. We can also consider that the honeypot B emulates the HTTP service and XSS vulnerability in a different level of interaction. Current architecture is still failing in combining both honeypots to attract as many attacks as possible. Existing data control mechanisms in Gen-III architecture are not sufficient to accommodate such cases.

To defeat honeypot fingerprinting techniques and to provide fine-grained data control for honeynet, we propose to leverage the emerging software-defined networking (SDN) architecture and techniques [24]. SDN provides a flexible and programmable network environment along with enhanced control of the network by separating the control plane from the data plane. In SDN, a network administrator (or a program operating on their behalf) can centrally program data control logic via specific APIs (i.e., OpenFlow [12]). Because an SDN switch can dynamically control network traffic by applying various actions, data control in honeynet can centrally be managed with the help of SDN.

SDN-NFVSec'16, March 11, 2016, New Orleans, LA, USA.
© 2016 ACM. ISBN 978-1-4503-4078-6/16/03...$15.00
DOI: http://dx.doi.org/10.1145/2876019.2876022

```
1   class command_uname(HoneyPotCommand):
2   def call(self):
3   if len(self.args) and self.args[0].strip() in
        ('-a', '--all'):
4   self.writeln('Linux %s 2.6.26-2-686 #1 SMP
        Wed Nov 4 20:45:37 UTC 2009 i686 GNU/
        Linux' %self.honeypot.hostname)
5   else:
6   self.writeln('Linux')
7   commands['/bin/uname'] = command_uname
```

Figure 1: Hardcoded Linux Version in Kippo Honeypot Source Code.

```
def process_uname(self):
self.transport.write(FAKE_OS+'\r\n')
...
FQDN = "fqdn_placeholder" # fake domain name
    (i.e., www.example.com)
FAKE_KERNEL_VERSION = "2.6.9-5.ELsmp #1 SMP"
TIMESTAMP = datetime.now().strftime("%b %d %H
    :%M:%S %Z %Y")
FAKE_OS="Linux "+FQDN+" "+FAKE_KERNEL_VERSION
    +" "+TIMESTAMP+" i386 GNU/Linux"
```

Figure 2: Timestamp Generation in Kojoney Honeypot Source Code.

To take advantage of SDN in building such a honeynet, we propose an SDN-enabled intelligent honeynet called HoneyMix. HoneyMix keeps a map of all available services in the network and generates data control rules in a centralized manner. To maximize the use of every honeypot, HoneyMix adopts group communication methods (multicast) to distribute incoming traffic to a set of associated honeypots. Then, HoneyMix selects the most desirable connection that might induce attackers further behaviors, and it replies back to attackers while associated honeypots collecting malicious data. To do this, HoneyMix has five core components: (1) *Response Scrubber* module, (2) *Forwarding Decision Engine* (FDE), (3) *Connection Selection Engine* (CSE), (4) *Behavior Learner* module, and (5) *SDN switch (es)*.

This paper is organized as follows. We discuss the limitations of existing honeynet architecture in Section 2. To solve the problems, We present the design of HoneyMix along with its core components in Section 3. In Section 4, we discuss related work. Section 5 concludes this paper.

2. PROBLEM STATEMENT

In this section we overview two limitations of existing honeynets: (1) most honeypots can be easily fingerprinted and (2) Gen-III honeynets only provide coarse-grained data control.

2.1 Honeypot Detection Techniques

A wide range of fingerprinting techniques have been devised to detect the existence of honeypots. Some efforts [1, 22] focused on finding a group of invariants that indicates the operating system (or the service) is emulated. For example, Dean et al. [1] revealed that the Kippo honeypot [7], when emulating a fake SSH service, always returns the same string when attackers access its system information. As shown in Figure 1, the default system information emulated in Kippo is hard-coded so that it prints out in two ways: (i) `Linux` (line 6); or (ii) `Linux` ⟨*hostname*⟩ `2.6.26-2-686 #1 SMP Wed Nov 4 20:45:37 UTC 2009 i686 GNU/Linux` (line 4). Thus,

Table 1: Overlapping services of honeypots.

Honeypot	Level of Interaction	Provided Service		
		HTTP	SSH	FTP
Dionaea [3]	Low-Interaction	✓		✓
Valhala [11]	Low-Interaction	✓		✓
Glastopf [4]	Low-Interaction	✓		
HIHIT [25]	High-Interaction	✓		
Kojoney [9]	Low-Interaction		✓	
Kippo [7]	Low-Interaction		✓	
Honssh [5]	High-Interaction		✓	

this timestamp in (ii) is a strong indicator of Kippo. Another SSH honeypot named Kojoney [9] uses a timestamp generated at installation time as shown in Figure 2. Even though the timestamp is not hardcoded, attackers can easily compare the timestamp with the time of attack to detect Kojoney-based SSH honeypots.

In addition, Holz et al. [18] introduced other techniques that help attackers detect honeypots by checking for suspicious environments. Because the majority of honeypots run in virtualized environments, attackers can infer the presence of honeypots by checking their environmental variables. For example, user-mode Linux and physical device information (video card or network interface card) are used to identify virtual environments of honeypots.

2.2 Coarse-grained Data Control in Honeynet

A honeynet is a network of honeypots that is intended to attract as many attackers as possible to collect data and learn about the patterns, behaviors, and tactics of attackers [2, 28]. The third generation (Gen-III) honeynet [16, 23] adopts a customized firewall called `honeywall` to realize two important honeynet functionalities: *data control* and *data capture.*

- **Data control:** honeywall runs in layer-2 bridge mode to enable transparent monitoring of network traffic without revealing its presence. More importantly, it is used to contain the attacker's actions against external networks. Of particular worry is Denial of Service attacks, so honeywall limits the outbound connections that are generated from compromised honeypots.
- **Data capture:** to capture malicious payloads and behaviors, Gen-III honeynet integrates built-in logging tools and IDS utilities such as iptables [27], snort [10], and sebek [8].

However, the layer-2 bridge in Gen-III honeynet is not sufficient to dynamically convey data to honeypots where it can properly be handled. A set of honeypots offering the same fake services in the network (e.g., Kippo and Kojoney) need to receive the same copy of relevant packets to maximize the use of them while existing honeywall cannot support those architecture. As shown in Table 1, there exist various honeypots that emulate the same service in terms of SSH and HTTP (web) services. In particular, when we put a low-interaction honeypot and a high-interaction honeypot together, which is categorized by different level of interaction, in the network, determining the flow path of data for distribution becomes more complex. Because low-interaction honeypots are usually effective in only the early stage of attacks, in-depth data collection are mostly performed by high-interaction honeypots.

There are few efforts to address this issue such as Honeybrid [6]. To facilitate the use of both honeypots, Honeybrid forwards initial attack traffic to low-interaction honeypot and migrates the connection to a high-interaction honeypot if needed. However, this approach does not fully utilize both honeypots since it allows only one connection at a time, and it would not work if the pair of honeypots consists of low-interaction (or high-interaction) honeypots.

Moreover, Gen-III honeynet is only concerned about containing the *outbound traffic*, however an attack is also likely to be dangerous to internal network. If a compromised honeypot attempts to infect another honeypot in the same network, honeywall cannot provide protection because malicious traffic is not destined to the external network.

3. HoneyMix ARCHITECTURE

In this Section we illustrate the five core components of HoneyMix in detail. In particular, we focus on how HoneyMix achieves better *data control* than existing Gen-III honeynet using Software-defined Networking (SDN).

3.1 Overview of HoneyMix

HoneyMix is based on traditional Gen-III architecture that includes a honeywall for controlling network traffic and capturing malicious data. Behind the honeywall, we construct an SDN-enabled network to accomplish fine-grained *data control*. By doing this, we not only take advantage of Gen-III architecture but also enhance security of honeynet with the help of SDN.

HoneyMix architecture consists of five components:

- `Response Scrubber` takes known fingerprinting techniques into account to reduce the possibility of exposure by scrubbing the response. Recall the example in Section 2.1, network operators manually define existing detection mechanisms with their countermeasures to perform sanitization. Response Scrubber first inspects attackers' requests and selectively scrubs corresponding response that reveals the existence of honeypots.
- `Forwarding Decision Engine (FDE)` creates a "`service map`" that represents the services offered by the honeynet (heterogeneity) and overlapping services (redundancy) across honeypots. Based on the service map, FDE centrally determines where network traffic should be forwarded and pings each service running on every honeypot to ensure consistent and up-to-date honeynet status. To maximize the use of the honeypots, FDE leverages SDN switches to forward malicious requests to all associated honeypots. To contain the malicious traffic and deliver seamless service, FDE quarantines the compromised honeypot and instantiates an identical honeypot using Network Function Virtualization (NFV) technique.
- `Connection Selection Engine (CSE)` establishes an end-to-end connection between an attacker and a honeypot. HoneyMix maintains one connection between an *attacker* and *SDN switch* and a number of connections between *SDN switch (es)* and *honeypots*. CSE selects one of the connections from the latter area and pipes it to the connection of the former area. Because HoneyMix basically breaks end-to-end connection, rewriting several header fields such as SEQ/ACK and checksums is necessary. *Behavior Learner* is responsible for computing weights, which CSE uses to select the connection.
- `Behavior Learner` computes a *weight* for each connection between SDN switch(es) and honeypots. This weight acts as a score that indicates the activity of a specific attacker's connection. Based on the duration time (δt) of the active connection and the frequency of modification ($\#n$) counted by *Response Scrubber*, this module assigns a weight to each connection. The longer the connection continues and the fewer modifications are made, the higher the connection weight has.
- `SDN switch(es)` connects with HoneyMix-enabled controller to receive an instruction for steering data flow and modifying network traffic in flight. SDN switch can dynamically quarantine a compromised honeypot and establish a new data stream for the newly instantiated honeypot using Network Function Virtualization (NFV [17, 32]). The switch is mainly controlled by FDE.

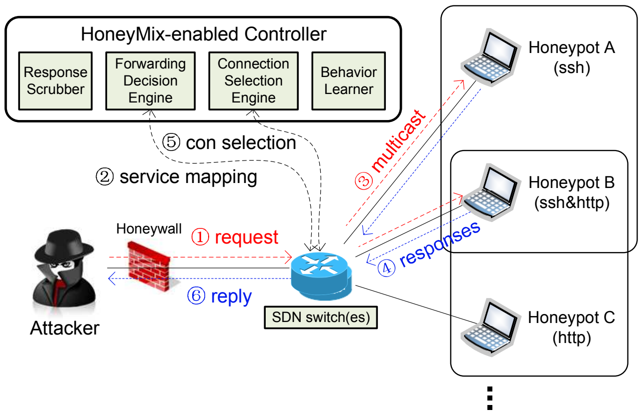

Figure 3: HoneyMix Architecture.

To illustrate how HoneyMix works, we walk through a use case as shown in Figure 3. When an attacker initiates a connection, HoneyMix inspects the IP addresses and port numbers to decide which services are associated with the connection. If the connection attempt is destined to an SSH service (default port: 22), FDE searches valid honeypots in the network using the `service map` and installs forwarding rules into corresponding SDN switches. At the same time, CSE establishes a connection with an attacker on behalf of honeypots. Upon a successful handshake, CSE creates multiple connections with relevant honeypots. HoneyMix performs selective traffic distribution using group communication (multicast). The conveyed request will trigger honeypots to generate multiple responses. *Behavior Learner* returns the *weight* of each connection so that CSE will choose one of them and pipeline it to the connection which is established by FDE. If *Response Scrubber* detects the attempt of fingerprinting, it sanitizes the response to make sure there exists no clear evidence that indicates the system (or the service) is emulated.

HoneyMix architecture has several strengths in *data control* when compared to the traditional Gen-III architecture. First, every honeypot that offers the same service can receive

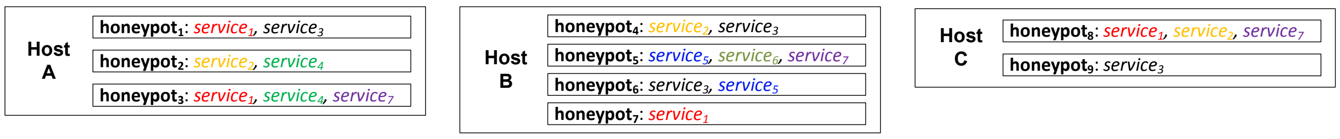
Host A
honeypot1: service1, service3
honeypot2: service2, service4
honeypot3: service1, service4, service7
Host B
honeypot4: service2, service3
honeypot5: service5, service6, service7
honeypot6: service3, service5
honeypot7: service1
Host C
honeypot8: service1, service2, service7
honeypot9: service3

Figure 4: Heterogeneous and Redundant Service Distribution in Honeynet.

malicious traffic by means of multicast while only one honeypot can be accessed at any given time in the traditional honeynet. This help us maximize the use of multiple honeypots in the network. Second, it dynamically selects the connection to send back with the most desirable response. This response, of course, is immune to known fingerprinting techniques when it is delivered to attackers (thanks to *Response Scrubber*). Third, SDN-enabled network can realize flexible incident response by isolating a compromised honeypot. With the combination of NFV and SDN, HoneyMix also activates a new honeypot and dynamically enable data communication to deliver seamless service. In addition to that, network reconfiguration techniques such as Moving Target Defense (MTD [20]) are also possible in HoneyMix architecture. Section 4.3 discusses this issue in detail.

3.2 Centralized Data Control

3.2.1 Network Rule Computation

Hosting honeypots in honeynet requires significant manual configuration (e.g., adding ACL and routing rules). In particular, honeypots co-existing on the same host may offer a set of redundant services. As shown in Figure 4, we consider a honeynet that consists of three hosts, running nine honeypots with seven vulnerable services[1] in total. Each honeypot may emulate multiple services (heterogeneity). In addition, several services are necessarily redundant (redundancy) because the number of services is less than the number of honeypots. For example, honeypot$_1$ provides two services ($service_1$, $service_3$) and $service_1$ is provided by two honeypots (honeypot$_1$, honeypot$_3$) on `Host A`.

Due to the heterogeneity and redundancy of provided services, generating network rules needs to consider the relations among *host*, *honeypot*, and *service*. We formalize this problem using aforementioned elements as follows:

- $HN = \{h_1, h_2, \cdots, h_l\}$ is a set of hosts in the honeynet;
- $HP = \{hp_1, hp_2, \cdots, hp_m\}$ is a set of honeypots in the honeynet;
- $SVC = \{svc_1, svc_2, \cdots, svc_n\}$ is a set of provided services in the honeynet;
- $\vec{H}_{h_l} = \langle rh_1, rh_2, \cdots, rh_m \rangle$ is an m-dimensional vector that corresponds to running honeypots on a specific host h_l. rh_m equals to '1' when hp_m is installed on host h_l, otherwise '0';
- $\vec{S}_{hp_m} = \langle as_1, as_2, \cdots, as_n \rangle$ is an n-dimensional vector that represents active services on a particular honeypot hp_m. as_n equals to '1' when svc_n is active on honeypot hp_m, otherwise '0';
- SM is the '`service map`' constructed in an $(m \times n)$ matrix form in which each row corresponds to $\vec{S}_{hp_1}$, $\vec{S}_{hp_2}$, $\cdots$, and $\vec{S}_{hp_m}$.

With $\vec{H}_{h_k}$ and SM, we first compute service distribution $\overrightarrow{SD}_{h_k}$ on a host h_k such that $1 \leq k \leq l$.

$$\overrightarrow{SD}_{h_k} = \vec{H}_{h_k} \cdot SM \tag{1}$$

$\overrightarrow{SD}_{h_k}$ is an n-dimensional vector that shows heterogeneity and redundancy of services on h_k. For example, the list of honeypots on Host A in Figure 4 is $\vec{H}_{h_A} = (1,1,1,0,0,0,0,0,0)$. SM would be as below:

$$SM = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Therefore, $\overrightarrow{SD}_{h_A} = \vec{H}_{h_A} \cdot SM = (2,1,1,2,0,0,1)$ refers to the distribution of redundant services on Host A. We next compute entire service distribution ($\overrightarrow{SDH}$) in honeynet via the addition of $\overrightarrow{SD}_{h_1}$, $\overrightarrow{SD}_{h_2}$, $\cdots$, and $\overrightarrow{SD}_{h_l}$.

$$\overrightarrow{SDH} = \sum_{k=1}^{l} \overrightarrow{SD}_{h_k} \tag{2}$$

Therefore, entire service distribution in honeynet (Figure 4) is $\overrightarrow{SDH} = (4,3,4,2,2,1,3)$. Note that $\overrightarrow{SDH}$ can also be computed by a sum of row vectors of SM.

Based on above observations, Forwarding Decision Engine (FDE) translates the entire service distribution ($\overrightarrow{SDH}$) into corresponding network rules. FDE generates network rules for each service and assigns different port numbers to differentiate redundant services on the same host if necessary.

3.2.2 Incident Response

For the best use of the centralized architecture of HoneyMix, we detect abnormalities in honeypots and reactively cope with incidents. HoneyMix takes advantage of Gen-III architecture that only limits a large volume of outbound traffic (rate limiting). In addition to this, we dynamically re-configure network rules to quarantine a compromised honeypot by leveraging the programmability of SDN. Based on the logs collected from honeywall, FDE in HoneyMix removes existing network rules associated with the compromised honeypot and installs a new rule to block outbound traffic from it. However, existing services provided by the compromised honeypot would remain damaged until we fin-

[1]Here, a *service* roughly means any type of program which is occupying a specific network port to provide a communication channel.

ish the investigation (e.g., forensic) and recover the honeypot.

To remedy this limitation, HoneyMix embraces network function virtualization (NFV) technique. Some efforts [26, 13] to build existing honeypots into a set of virtual instances would also help us realize this approach. HoneyMix periodically snapshots each honeypot, and dynamically activates it when an infected honeypot is detected. In such a way, HoneyMix ensures that every service in the network is always up and running.

3.3 Dynamic Connection Selection

To realize the architecture of HoneyMix, the importance of connection selection in *SDN switch(es)–honeypots* area cannot be stressed enough. However there exist many kinds of obstacles in enabling this. First, dynamically hopping one connection to another essentially breaks end-to-end connection between an attacker and a particular service. Modification of several packet headers, especially in the TCP protocol, must be considered (e.g., rewriting of SEQ/ACK numbers). Moreover, this may bring new chances for attackers to fingerprint the existence of NAT functions by checking RTT delays. Second, selecting the most desirable connection that could encourage attackers to launch subsequent attacks is not trivial. Because there exists no clue to judge about the suitability of connections, we need to develop a set of criteria to evaluate them in a reasonable way. Third, choosing the right time for connection selection is also challenging. For example, we do not want to transfer the connection in the middle of transmitting a big file.

To accommodate the aforementioned challenges, we devise Connection Selection Engine (CSE) as a core building block for connection selection. To enable seamless connection between an attacker and an emulated service in honeypot, CSE leverages existing SDN features that allows us to perform network address translation (NAT). For data transmission over TCP protocol, HoneyMix maintains the *state tracking table* in CSE that keeps track of sequence (SEQ) and acknowledgement (ACK) numbers of connections. In this table, CSE also inserts additional information for the higher layers in the OSI reference model to handle a lot of nonce which are dynamically generated by a specific service. For example, HTTP service independently keeps cookie, referer, and authorization information to maintain the state of users. Moreover, CSE records a set of key-pairs that are used for encrypting/decrypting messages for SSH/HTTPS service. Some additional header fields such as TCP checksum field are dynamically updated by CSE.

As discussed in Section 2.1, HoneyMix might circumvent existing fingerprinting techniques just by hopping one connection to another. To achieve this, selecting the most realistic and desirable connection from multiple connections is critical in building HoneyMix. Note that *Response Scrubber* module is also provisioning partial remedy for this in an ad-hoc manner by fixing apparent indicators, but we want to even avoid unknown fingerprinting techniques by dynamically hopping connections.

Behavior Learner module in HoneyMix determines the *weight* of each connection based on two criteria: (1) the duration time (δt) of a connection and (2) the number of modifications ($\#n$) made by *Response Scrubber*. To account for attackers' duration of session, the module measures continuing time of an active connection (δt_a). It next obtains the modification ratio which is the number of unsanitized response ($\#N - \#n$) divided by the number of successful responses ($\#N$), where $\#n$ is the number of modifications performed by *Response Scrubber*. Based on these two criteria, the *weight* of a connection a (W_a) is computed as below:

$$W_a = t_s \cdot \delta t_a \cdot \frac{\#N - \#n}{\#N} \tag{3}$$

We add a threshold (t_s) to prioritize a connection for the quality of service.

4. RELATED WORK AND DISCUSSION

4.1 Software-defined Networking (SDN)

SDN is an emerging network paradigm that separates the control plane from the data plane [24]. Legacy network devices embed complex control logics to process network traffic whereas SDN switches only perform simple "match-action" based processing. By simplifying the data plane, SDN abstracts the control plane and consolidates those control logic into a centralized controller. Because SDN enables logically centralized network environment, SDN supports significant programmability and flexibility that could help improve the security of honeynet.

As the prevalent and widely adopted SDN protocol, OpenFlow [12] realizes such an SDN paradigm. To dynamically program network traffic, the OpenFlow protocol supports "Set-Field" operation in the data plane that allows us to rewrite packet headers. Therefore, OpenFlow-enabled network implements network address translation (NAT) feature without employing additional network box. HoneyMix makes the best use of OpenFlow-enabled network for realizing connection selection, which helps build more flexible and robust honeynet.

4.2 Honeynet Architecture

The first generation (Gen-I) of honeynet, which was devised in 1999 [28], employs a firewall that mainly performs *data control* at OSI layer-3. Although Gen-I architecture successfully proved its ability in collecting attacks, it can be easily detected by attackers. It could not properly handle outgoing traffic either. The cornerstone of the second generation (Gen-II) and the third generation (Gen-III) honeynets is a layer-2 based firewall called honeywall. Honeywall has been devised to enable transparent network monitoring by provisioning layer-2 bridging, which is difficult for attackers to detect. Gen-II and Gen-III have the same architecture except several additional functionalities [13]. Having Gen-II components as the basis, Gen-III utilizes honeypot monitoring tools (e.g., sebek [8]) to check abnormalities and implements easier deployment of the honeywall. As cloud infrastructure is widely adopted in today's networks, deploying Gen-III honeynet in a virtual environment becomes more popular since it brings many benefits (e.g., maintenance) that deployment in a physical machine cannot provide [26].

4.3 Discussion

HoneyMix uses some ideas that have been presented in moving target defense (MTD) [15, 31]. To increase uncertainty, MTD adopts several randomization techniques to reconfigure a set of properties of operating systems or network interfaces [14]. Jafarian et al. [20] proposed to use the SDN controller to randomize host information (i.e., MAC and IP addresses) in order not to allow attackers to obtain the real host information. Panos et al. [21] also proposed other randomization mechanisms to hide service version and OSes by utilizing an SDN application to generate traffic that resembles a fake service or OS.

To prevent honeypot fingerprinting HoneyMix currently adopts a connection selection engine that transfers from one connection to another. This scheme helps increase anonymity of existing honeypots by changing active connections. Moreover, *Response Scrubber* is designed to sanitize specific payloads that reveal information of honeypots. HoneyMix could adopt aforementioned MTD techniques to further minimize the possibility of exposing network infrastructure (OS, service, and host). For example, random host mutation techniques introduced in MTD can also be considered in HoneyMix-enabled controller to hide honeypots. In addition, we may consider to generate virtual IP and MAC addresses to dynamically create corresponding DNS information to hide our network configurations.

5. CONCLUSION

In this paper, we introduced HoneyMix architecture along with its five components to build SDN-enabled intelligent honeynet. To defeat existing honeypot fingerprinting techniques, HoneyMix dynamically selects the most desirable connection among multiple connections and reactively sanitizes the response if it contains a known indicator of honeypots. To fix coarse-grained data control in traditional Gen-III honeynet architecture, HoneyMix centrally computes service distribution in the network to enable fine-grained data control and deploys corresponding rules via SDN switches. Moreover, HoneyMix conducts a security incident response including quarantine and recovery using SDN and NFV. We are currently implementing HoneyMix and planning to evaluate the architecture in real-world deployments.

Acknowledgments

This work was partially supported by the grants from Center for Cybersecurity and Digital Forensics at ASU.

6. REFERENCES

[1] Black Hat USA 2015 - Breaking Honeypots For Fun And Profit. https://www.youtube.com/watch?v=Pjvr25lMKSY.

[2] Blogs|The Honeynet Project. https://www.honeynet.org/.

[3] Dionaea - carnivore. https://github.com/rep/dionaea.

[4] Glastopf Honeypot Project Page. http://glastopf.org/.

[5] Honssh Honeypot. https://github.com/tnich/honssh.

[6] Hybrid Honeypot Framework. http://honeybrid.sourceforge.net/.

[7] Kippo SSH Honeypot. https://github.com/desaster/kippo.

[8] Know Your Enemy: Sebek (A kernel based data capture tool). http://old.honeynet.org/papers/sebek.pdf.

[9] Kojoney2 SSH Honeypot. https://github.com/madirish/kojoney2.

[10] Snort.Org. https://www.snort.org/.

[11] Valhalahoneypot Honeypot. http://sourceforge.net/projects/valhalahoneypot/.

[12] OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06), December, 2014. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf.

[13] F. H. Abbasi and R. Harris. Experiences with a generation iii virtual honeynet. In *Telecommunication Networks and Applications Conference (ATNAC), 2009 Australasian*, pages 1–6. IEEE, 2009.

[14] E. Al-Shaer. Toward network configuration randomization for moving target defense. In *Moving Target Defense*, pages 153–159. Springer, 2011.

[15] M. Carvalho and R. Ford. Moving-target defenses for computer networks. *IEEE Security & Privacy*, (2):73–76, 2014.

[16] M. Dornseif, F. C. Freiling, N. Gedicke, and T. Holz. Design and implementation of the honey-dvd. In *Information Assurance Workshop, 2006 IEEE*, pages 231–238. IEEE, 2006.

[17] R. Guerzoni et al. Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper. In *SDN and OpenFlow World Congress*, 2012.

[18] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, pages 29–36. IEEE, 2005.

[19] J. H. Jafarian, E. Al-Shaer, and Q. Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 127–132. ACM, 2012.

[20] J. H. Jafarian, E. Al-Shaer, and Q. Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12)*, pages 127–132. ACM, 2012.

[21] P. Kampanakis, H. Perros, and T. Beyene. Sdn-based solutions for moving target defense network protection. In *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[22] N. Krawetz. Anti-honeypot technology. *Security & Privacy, IEEE*, 2(1):76–79, 2004.

[23] J. Levine, R. LaBella, H. Owen, D. Contis, and B. Culver. The use of honeynets to detect exploited systems across large enterprise networks. In *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, pages 92–99. IEEE, 2003.

[24] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[25] M. Mueter, F. Freiling, T. Holz, and J. Matthews. A generic toolkit for converting web applications into high-interaction honeypots. *University of Mannheim*, 280, 2008.

[26] N. Provos et al. A virtual honeypot framework. In *USENIX Security Symposium*, volume 173, 2004.

[27] M. Rash. *Linux Firewalls: Attack Detection and Response with iptables, psad, and fwsnort*. No Starch Press, 2007.

[28] L. Spitzner. The honeynet project: Trapping the hackers. *IEEE Security & Privacy*, (2):15–23, 2003.

[29] L. Spitzner. Honeypots: Catching the insider threat. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 170–179. IEEE, 2003.

[30] D. Watson and J. Riden. The honeynet project: Data collection tools, infrastructure, archives and analysis. In *WOMBAT Workshop on Information Security Threats Data Collection and Sharing*, pages 24–30. IEEE, 2008.

[31] R. Zhuang, S. A. DeLoach, and X. Ou. Towards a theory of moving target defense. In *Proceedings of the First ACM Workshop on Moving Target Defense*, pages 31–40. ACM, 2014.

[32] M. Zimmerman, D. Allan, M. Cohn, N. Damouny, C. Kolias, J. Maguire, S. Manning, D. McDysan, E. Roch, and M. Shirazipour. Openflow-enabled sdn and network functions virtualization. *Solution Brief, ONF, Solution Brief sbsdn-nvf-solution. pdf*, 2014.