HoneyProxy: Design and Implementation of Next-Generation Honeynet via SDN

Sukwha Kyung, Wonkyu Han, Naveen Tiwari, Vaibhav Hemant Dixit, Lakshmi Srinivas,
Ziming Zhao, Adam Doupé, and Gail-Joon Ahn
Laboratory of Security Engineering for Future Computing (SEFCOM)
and Center for Cybersecurity and Digital Forensics (CDF)
Arizona State University, Tempe, AZ, USA
Email: {skyung1, whan7, nktiwar1, vdixit2, lsriniv2, zmzhao, doupe, gahn1}@asu.edu

***Abstract*—Honeynet is a network architecture that utilizes multiple honeypots to deceive attackers and analyze their malicious behaviors. However, existing honeynet has not evolved much since its latest architecture, Gen-III, which was proposed in 2004. Meanwhile, security threats and techniques used by adversaries have been continuously advanced. As a result, honeypot architecture is suffering from its limited functionalities of 'data control' and 'data capture'. Existing data control mechanism does not monitor internal propagation of malwares in the network and also does not support honeypot transition from one to another (e.g., a low-interaction honeypot to a high-interaction honeypot). The data capture capability of traditional honeynet is also insufficient as it is vulnerable to fingerprinting attacks.**

To address these challenges, we design and implement an innovative SDN-based honeynet named HoneyProxy as a next generation honeynet. To prevent internal propagation of malwares within honeynet, HoneyProxy globally monitors all internal traffic with the help of Software-defined Network (SDN) controller. HoneyProxy utilizes a novel connection management mechanism across different honeypots in the network to support honeypot transitions. To this end, a HoneyProxy-enabled SDN controller centrally programs the reverse proxy module that operates in three specific modes. In addition, HoneyProxy improves the data capture capability in the existing honeynet by circumventing fingerprinting attacks through multicasting malicious traffic to relevant honeypots and selecting the response which does not contain fingerprinting indicator(s). Experimental results show that HoneyProxy can support almost line rate throughput (8.23 Gbps) on 10 Gbps link with a negligible latency overhead ($0.5 - 1.2$ milliseconds).

I. Introduction

A honeypot is a system that is designed to intentionally let attackers probe, scrutinize and ultimately exploit the system by exposing a set of vulnerable services [25], [12], [23]. The primary purpose of a honeypot is to closely monitor the emulated system to learn behaviors of attackers and collect malicious data during and after the exploitation of the honeypot. To achieve this goal, honeypots are intended to be under active attack by real adversaries and they are often isolated from the real operating system, services, or network. The activities of adversaries collected from honeypots can provide early warnings of new attacks and exploitation, enabling administrators to protect the real systems and networks.

Honeypots are generally categorized into two types: low-interaction honeypot (LIH) and high-interaction honeypot (HIH). The main difference between the two types lies in their complexity and the level of interaction they provide to the attacker. LIHs emulate operating systems and other services, and therefore do not provide attackers with much control over the given system. The main advantage of LIHs stems from their simplicity (i.e., easy deployment and maintenance) and the low risk factor, because they are merely simulated systems. However, LIHs can be easily fingerprinted [7]. HIHs are typically actual systems and elicit more interactive information from attackers than LIHs. However, maintenance and deployment cost of HIH is much higher. HIHs also have higher risk factor than that of LIHs because, unlike LIHs, they are real systems and therefore could cause more severe damages when compromised.

A *honeynet* is a network of honeypots created to enhance the interaction with attackers. However, honeynet poses the same weakness as that of honeypots. In addition, the first honeynet architecture (Gen-I [24]) has first been proposed in 2002, and the latest architecture, Gen-III [13], was built in 2004. Due to the outdated honeynet architecture, existing honeynet suffers from insufficient *data control* mechanisms and *data capture* capability. For example, inbound/outbound traffic control mechanisms in Gen-III architecture cannot prevent internal propagation of malware within a honeynet because access control rules are mainly enforced by a custom gateway called *honeywall* [16], [21]. It is also incapable of supporting the transition between a LIH and a HIH. LIHs are effective for collecting high level information about attackers (e.g., username and password pair), whereas HIHs focus on collecting low level details [5]. However, existing honeynet architecture does not provide a practical way to fully utilize the advantages of both LIH and HIH.

In order to solve aforementioned problems, we argue that the architecture of current honeynet should be redesigned to provide more flexibility in terms of its network access management. We observe that such flexibility and network access controls can be satisfied by taking advantage of Software-defined Networking (SDN [9]). SDN basically provides a centralized network management platform by decoupling the control plane (e.g., exchanging network rules) from the data plane (e.g., network switches). Routing policies of connected devices in SDN are centrally configured via the SDN controller, and

978-1-5386-0683-4/17/$31.00 ©2017 IEEE

the controller can provide a global view of the network to SDN applications to help network administrators easily build network-wide business logic. These strengths of SDN have high potentials to address the limitations of existing honeypots and honeynet architecture.

In this paper, we propose a novel honeynet architecture to overcome the limitations of existing honeypots and honeynet architecture by leveraging the SDN technology. HoneyProxy consists of a proxy module and a corresponding SDN application. It takes the form of a reverse proxy to provide improved control over incoming and outgoing traffic while obtaining network configuration via the SDN controller. Malicious traffic from attackers is redistributed to all associated honeypots, and HoneyProxy selects one response from the response queue that does not contain fingerprinting indicator(s). To prevent internal malware propagation, HoneyProxy cooperates with the SDN controller to detect any anomalies within the network. Supporting dynamic transition between a LIH and a HIH is realized by enabling three types of operating modes (Section IV-A).

The contributions of this paper are summarized as follows:

- We propose an SDN-based honeynet architecture called HoneyProxy that consists of a reverse proxy module and corresponding SDN application. HoneyProxy addresses important problems in existing honeypots and honeynet architecture: (1) fingerprinting attacks targeting honeypots, (2) internal malware propagation in honeynet, and (3) lack of honeypot transition.
- We propose a connection management engine that supports three operating modes: (1) Transparent Mode, (2) Multicast Mode, and (3) Relay Mode. Based on the decision of HoneyProxy controller, malicious traffic is processed differently so as to meet our design goals (Section III).
- We implement a prototype of HoneyProxy, and our experimental results show that TCP throughput of HoneyProxy achieves the line rate throughput (8.23 Gbps). The latency incurred by HoneyProxy is in the range of $0.5 - 1.2$ milliseconds on average. Connections per second (CPS) handled by HoneyProxy ranges from 640 to 1473 depending on the type of connection.

II. Problem Statement

Existing honeypots suffer from fingerprinting attacks, and current honeynet architecture suffers from internal malware propagation and a lack of honeypot transition mechanisms.

Vulnerable to fingerprinting attacks. A fundamental drawback of existing honeypots is that they can be easily fingerprinted by attackers. The essential objective of honeypots is to collect as much information of malicious behavior as possible to learn attacker's techniques and to discover new types of attacks and malware to provide early warnings to network administrators. However, lack of functionalities and insufficient interactions with honeypots (especially LIHs) discourages attackers from probing and exploiting the system. For example, existing ssh honeypots such as kippo [7] can be easily fingerprinted by using Linux commands such as `uname -a` because kippo simply simulates the functionality of `uname` command by printing out the hard-coded timestamp "Wed Nov 4 20:45:37 UTC 2009" (see Table I). In this way, attackers can instantly identify the presence of honeypots, which reduces the effectiveness in collecting attackers' behavior. In other words, neither honeypots nor honeynet architecture is capable of preventing fingerprinting attempts due to the lack of *data control* mechanism.

Request		Response	
type	payload	type	payload
exact match	uname -a	exact match	Wed Nov 4 20:45:37 UTC 2009
pattern	.{7,}\n	exact match	bad packet length
exact match	vi	exact match	E558: Terminal entry not found in terminfo
exact match	ifconfig	exact match	HWaddr 00:4c:a8:ab:32:f4

TABLE I: Example of known fingerprinting indicators for the ssh honeypot kippo.

Internal propagation of malware. Current honeynet architecture cannot monitor internal traffic because access control mechanisms are enforced at the custom gateway called the *honeywall*. Honeywall monitors incoming and outgoing traffic at a fixed location and acts as a traditional network firewall. Due to the fixed placement of a honeywall, monitoring and preventing internal propagation of malware in the honeynet is difficult. In general, honeypots are not to be trusted because attackers are encouraged to actively exploit those honeypots. Therefore, if a honeypot is compromised, it can easily infect other honeypots coexisting in the same network. To prevent these incidents, administrators may want to add host-based protection mechanisms within a machine (e.g., anti-virus, iptables, or sandbox). However, host-based solutions are not feasible because the attacker, who is taking control of the honeypot, can circumvent these countermeasures. This is why existing honeynet architecture should be redesigned to provide better network-level protection.

Dynamic transition between LIH and HIH. LIHs emulate a set of real functionalities and expose (fake) vulnerable and exploitable services to attackers. In particular, LIHs are widely used in the early stage of attacks to collect information on scanning attacks and login attempts (i.e., username/password pairs). HIHs implement the majority of real service (e.g., ssh and http). While HIHs provide deeper and realistic interactions to attackers, they require sophisticated configurations, high maintenance cost, and high possibility of compromise. Consequently, Current honeynet mechanisms totally rely on the capability of each honeypot, resulting in the loss of potential opportunities for maximizing the advantages of both LIHs and HIHs. For example, we could use both types of honeypots by activating LIHs for scanning attacks or the login phase of an attack, while enabling HIHs to provide more interactive attacker actions after a successful login event. Honeybrid [20], [5] strives to facilitate the use of both honeypots by supporting transition mechanisms between a LIH and a HIH. However, this approach does not provide

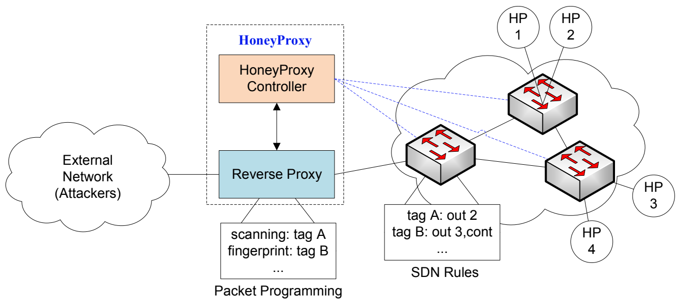
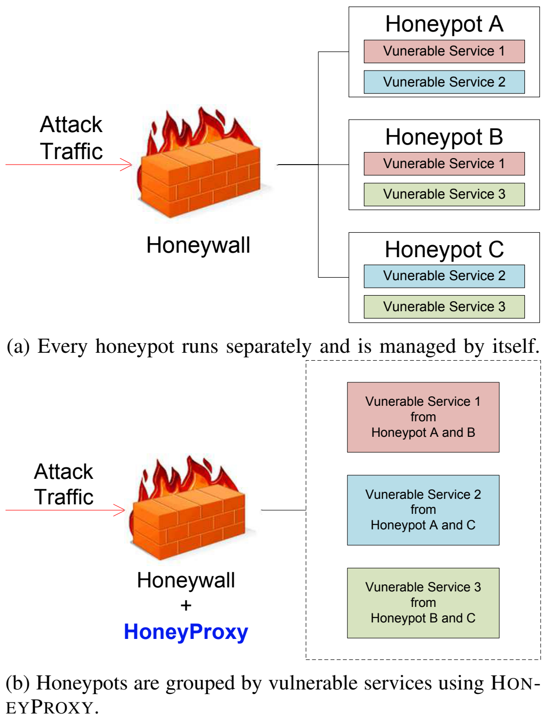


Fig. 1: Overview of HONEYPROXY.



(a) Every honeypot runs separately and is managed by itself.

(b) Honeypots are grouped by vulnerable services using HONEYPROXY.

Fig. 2: HONEYPROXY reshapes the landscape of honeynet architecture toward one '*BIG*' honeypot.

a flexible way to configure when and how to migrate the establish connection from the one to another.

III. HONEYPROXY: DESIGN AND ARCHITECTURE

We propose HONEYPROXY as a next-generation honeynet architecture, which leverages Software-Defined Networking (SDN) to overcome the limitation of existing honeypots. In this section, we describe the key design goals of our approach, and we illustrate the architecture of HONEYPROXY along with the detailed building blocks.

A. Design Goals

We define the following design goals that any next-generation honeynet architecture should support:

- **Universality.** The approach should be able to monitor all internal traffic to prevent compromised honeypots from propagating malware within the network. Universality also means centralized network monitoring and network-wide (i.e., *universal*) policy enforcement, which is achieved by leveraging SDN.
- **Flexibility.** The honeynet architecture must support a seamless transition from a LIH to a HIH and vice versa. This transaction should also be flexible and configurable.
- **Stealthiness.** The approach must be covert — it has to hide the existence of itself and minimize the exposure of residing honeypots as much as possible. Therefore, the approach should not incur noticeable delay in conducting the given tasks, as the delay can result in the detection of the honeynet.
- **Generalization.** The approach should be applicable regardless of the type of residing honeypots or running services. The key question here is related to how the approach can address and coordinate the redundant services offered by different honeypots.

B. HONEYPROXY *Overview*

At high level, HONEYPROXY consists of a proxy module and a SDN controller with corresponding application (*HoneyProxy controller*) that enforces security rules and necessary network rules (see Figure 1). Multiple honeypots are connected to different switches, and they are centrally managed by the HONEYPROXY controller. The requests sent by the attackers pass through a series of modules in the proxy and are transmitted to a set of relevant honeypots.

As shown in Figure 1, the proxy pushes a specific type of tagging information inside the packet headers. HONEYPROXY controller then creates SDN rules that check the tagging information in SDN switches to enforce network policies efficiently. The proxy module has three operational modes. Based on the decision made by the HONEYPROXY controller, the operating mode of the proxy would be reconfigured when necessary (Section IV). To prevent fingerprinting attack, the proxy module inspects the payloads of response to see if it includes any fingerprinting indicators that may expose the presence of honeypots and/or honeynet. Upon discovering such an indicator, the proxy module signals the HONEYPROXY controller to take appropriate action, such as changing the proxy mode or updating network configurations. The proxy module is also responsible for handling encrypted communication (for e.g., ssh connection). Section III-C provides detailed architecture and building blocks of HONEYPROXY.

Figure 2 illustrates how HONEYPROXY changes the landscape of honeynet architecture. Traditional honeynet architecture runs multiple honeypots behind the custom firewall (honeywall). However, the traditional architecture may rise redundancy of the same emulated services because of the lack of interaction between honeypots, as shown in Figure 2a. This is the main cause of inefficient data control, and as a result, only one honeypot is accessible to an attacker at any given time. Moreover, each honeypot requires a lot of manual configurations to simulate all possible services due to lack of transition between honeypots, which is the main cause of redundant services in the honeynet.

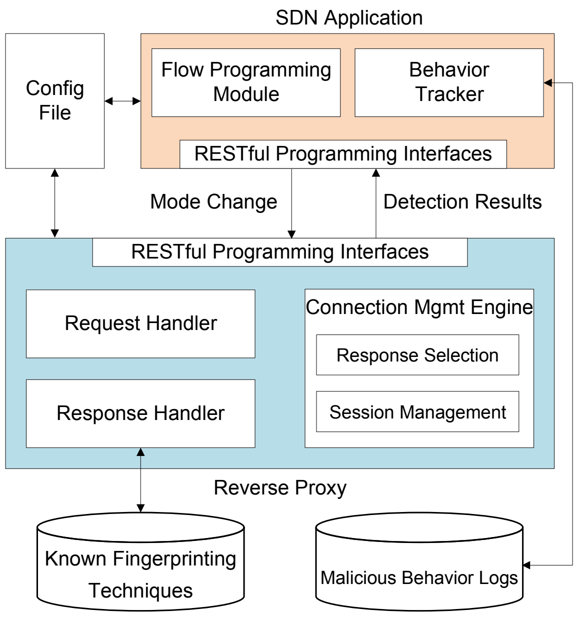


Fig. 3: HONEYPROXY Architecture.

HONEYPROXY, shown in Figure 2b, allows us to operate the honeynet as one large honeypot running many vulnerable services, which are the union of individual honeypots. It does not require the extra burden of running redundant services across diverse honeypots, because the proxy module distributes requests and selects the best response. In other words, the attacker is not able to determine one honeypot from another and sees our honeynet as only one honeypot. To effectively generate multicast messages, the proxy module of HONEYPROXY internally performs network address translation (NAT [26]) and deep packet inspection (DPI [14]) to interconnect the same services across honeypots.

Our approach has several strengths compared to the existing honeynet architecture: (1) Our approach allows attackers to easily access various vulnerable services inducing more interaction with honeypots, which enables network administrators to collect more data about malicious behaviors; (2) Fingerprinting attacks can be mitigated by dynamically selecting the most appropriate response; (3) SDN controller monitors the entire network of honepots to detect abnormal behaviors (for e.g., connection attempts between honeypots) so that internal propagation of malware can be easily prevented beforehand.

C. Architecture and Building Blocks

The HONEYPROXY architecture is illustrated in Figure 3. HONEYPROXY consists of a *reverse proxy* module and an SDN application (*HoneyProxy controller*). This design divides network programming and packet processing into two distinct logical layers. The reverse proxy module processes incoming and outgoing traffic using three sub-components: *Request Handler*, *Connection Management Engine*, and *Response Scrubber*. The SDN application manages network configurations and enforces SDN rules, while monitoring suspicious packets within the network. Detailes of HONEYPROXY modules are as follows:

Request Handler is responsible for handling the incoming traffic. When a packet is received by Request Handler, the payload is checked to decide if the traffic contains any known fingerprinting attacks, which can reveals existence of the honeypot (see Table [7]). If the payload contains scanning attacks, which require to use L3 or below layer protocols, Request Handler adds the *scanning* tag to the packets and directly forwards to honeypots that are running intrusion detection systems (IDS). Then, based on the result of checking payload, the Request Handler signals the Connection Management Engine to perform NAT and DPI to manage the sessions. Therefore, the main function of Request Handler is to monitor incoming traffic for suspicious packets and send the result to Connection Management Engine.

Connection Management Engine is the core of reverse proxy module that orchestrates Request and Response Handler. The main goal of the engine is to select a response among multiple responses received from honeypots and maintain the sessions to support three operating modes of HONEYPROXY (Section IV). Connection Management Engine also adds tagging information to packet headers of incoming traffic, allowing SDN switches to forward them to matching destination.

Response Handler is responsible for detecting fingerprinting indicators that may exist in the responses received from honeypots. Responses including such indicators (Table I) trigger this module to notify HONEYPROXY controller. First, responses from associated honeypots are recorded in the R_Queue, waiting for the arrival of remaining responses until the size of the queue is equal to the number of associated honeypots. If the queue size and number of honeypots match (or timeout event is triggered), then Connection Management Engine selects the most appropriate response from the R_Queue.

Flow Programming Module runs as a part of the SDN applications of HONEYPROXY controller. This module is responsible for notifying the controller to add SDN rules (i.e., a flow entry) that correspond to particular traffic processed by the reverse proxy. Packets marked as *scanning* will be forwarded to appropriate honeypots. i.e., the ones that are running IDS (e.g., snort [11]), which is specifically designed to detect scanning attacks.

Mode Decision Module determines operation mode of the proxy. Based on several criteria (Section IV-C), this modules sends request to the proxy to change the operating mode.

To achieve the first design goal (universality), HONEYPROXY leverages SDN to make a decision on operating modes of HONEYPROXY and enforce network and security rules via SDN controller. HONEYPROXY monitors all flows in the network via the SDN controller so that any connection attempts generated by (potentially) compromised honeypots can be logged, monitored, and prevented. To support dynamic transitions between honeypots seamlessly (the second design goal), *Connection Management Engine* in the proxy selects

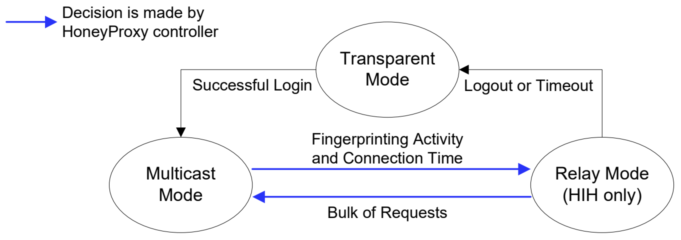


Fig. 4: Transition Conditions among Three Operating Modes.

the most appropriate response from the receiving queue and tracks the state changes of all active connections. In this way, HONEYPROXY can also transparently migrate the connection from one honeypot to another. To achieve the third design goal (stealthiness), HONEYPROXY attempts to minimize the performance gaps between different operating modes of HONEYPROXY using multi-processing techniques [22]. As elaborated in Section VII, latency gaps between different modes are less than a millisecond (< 1 ms), which is hardly distinguishable when attackers connect over the internet. To meet the last design goal, generalization, HONEYPROXY establishes multiple sockets with the associated honeypots to support L4 or higher in OSI layer. Since vulnerable services are mostly utilizing application layer protocol (L7) except for scanning attacks, HONEYPROXY can accommodate to most of protocols. For scanning attacks utilizing L3 or below, SDN application of HONEYPROXY redirects those packets to one of honeypots that runs an intrusion detection systems, which are specifically designed to detect scanning attacks.

IV. Operating Modes and Connection Management Mechanism

The *Connection Management Engine* supports three operating modes: transparent mode (*T-Mode*), multicast mode (*M-Mode*), and relay mode (*R-Mode*). The purpose of these modes is to efficiently and effectively deliver malicious traffic to relevant honeypots and select the most appropriate reply among multiple responses from the honeypots.

A. Operating Modes

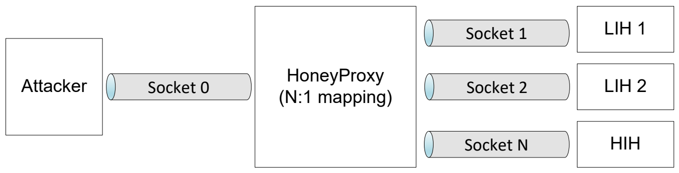
Figure 4 illustrates the operation of three modes that HONEYPROXY supports. Each mode is intended to provide the following features:

- **Transparent Mode (*T-Mode*):** T-Mode accounts for the initial stage of attacks such as login trials. Because these attempts are normally launched in an automated manner (e.g., bots or scripts), low-interaction honeypots can effectively handle such attacks. For scalability, HONEYPROXY only performs network address translation without conducting deep packet inspection in this mode.
- **Multicast Mode (*M-Mode*):** Upon the successful login of attacker, HONEYPROXY transfer the mode from T-Mode to M-Mode to counteract fingerprinting attacks. In this mode, every incoming payload is delivered to all associated honeypots. However, merely sending multicast messages would not work because each session has unique session variables such as cookie or shared session key, which are created and managed by the end honeypot. To address this issue, HONEYPROXY builds multiple sockets to maintain a set of connections between the honeypots and HONEYPROXY and records session data. Section IV elaborates on how HONEYPROXY maintains multiple session data and determines the best reply to send to the attacker among all received responses.
- **Relay Mode (*R-Mode*):** On the receipt of mode change commands issued by HONEYPROXY controller, HONEYPROXY transfers the operating mode from M-Mode to R-Mode or vice versa. R-Mode essentially allows only one connection, which is established by a HIH, to interact with the attacker while other sessions are temporarily suspended. Keeping advanced and motivated attackers connected only with LIHs is impractical and not feasible. In such case, HONEYPROXY does no longer take the burden caused by M-Mode (i.e., sending multicast messages to associated honeypots). Therefore, R-Mode enhances performance by configuring rest of the sessions to a standby state. If necessary (e.g., bulk requests that exceed a specified threshold), the controller can issue a mode transition back to M-Mode to let LIHs interact with attackers again.

B. Response Selection and Session Management

HONEYPROXY maintains a database of known fingerprinting indicators that expose the presence of honeypots or honeynet architecture. For example, Table I describes several known fingerprints from an ssh honeypot named kippo [7], [1]. The proxy module of HONEYPROXY prevents sending known fingerprinting responses to the attacker, therefore it selects another response that does not contain the fingerprint(s). Because this task is performed during the deep inspection of packets in flight, the selection decision would not be made by the SDN application but by the proxy module directly. However, fingerprinting traces are reported back to the SDN application of HONEYPROXY for later usage by the Mode Decision Module. Note that finding fingerprints of honeypots or honeynet architecture shown in Table I is out of our research scope.

HONEYPROXY manages multiple sessions in a structured fashion. It establishes a session with an attacker and internally creates a number of sessions (n) with the associated honeypots. Figure 5 illustrates a snapshot of active sockets running in M-Mode that maintains $1 : N$ sessions. Socket 0 corresponds to a connection made by the attacker, while the rest correspond to each connection with a vulnerable service of N honeypots. Each session is centrally managed by the proxy module of HONEYPROXY (Connection Management Engine). The proxy module also manages attacker's identity (e.g. a pair of username and password). Because the amount of each session data may vary, the data is stored in the table so that HONEYPROXY can rewrite the attacker's socket, allowing the vulnerable service to properly accept payloads. Examples of session information include cookies of a HTTP service and a shared session key of an ssh service.



{SOCKET 0: {"USER": "AB", "PASSWD": "pwd-ab", "HOST": "[proxy_ip]:80", "SESSION": "x"}}
{SOCKET 1: {"USER": "AB", "PASSWD": "pwd-ab", "HOST": "10.0.0.1:80", "SESSION": "x"}}
{SOCKET 2: {"USER": "AB", "PASSWD": "pwd-ab", "HOST": "10.0.0.2:80", "SESSION": "xx"}}
{SOCKET N: {"USER": "AB", "PASSWD": "pwd-ab", "HOST": "[ip]:[port]", "SESSION": "xxx"}}

Fig. 5: An illustration of connection selection and session management.

C. Transition Criteria of HONEYPROXY Controller

To make a reasonable decision of whether or not the attacker needs to be served by a high-interaction honeypot (R-Mode), we develop several criteria, consisting of connection duration (δt), fingerprinting attack counts ($\#c$), and previous record of an attacker ($< R_t, R_c >$). The same IP address that has the same identity (username and password pair) is used to locate the previous records. For attackers who previously accessed our honeynet, we keep records of them in *Malicious Behavior Logs* repository. HONEYPROXY looks up the past records from the repository and utilize them to better serve the attackers. The mode of operation function (f_m) for the session (s) is determined by the following equation.

$$f_m(s) = \begin{cases} \text{R-Mode}, & \text{if } \frac{R_t}{R_t+\delta t} \cdot R_c + \frac{\delta t}{R_t+\delta t} \cdot \#c \geq \theta \\ \text{M-Mode}, & \text{otherwise} \end{cases}$$

The threshold value (θ) is configured to balance the workloads of low-interaction honeypots and high-interaction honeypots.

V. FLOW PROGRAMMING MECHANISM

HONEYPROXY takes advantage of the high programmability of SDN. The Connection Management Engine classifies the type of incoming packets and adds tagging information to the packets. Using those tags, the SDN controller enforces appropriate actions to process the packets.

A. Flow Programming

Inspired by tagging techniques [15], [17], we leverage the MPLS field to classify incoming traffic and statically reroute the packets based on the marked tag. The Request Handler first determines whether the incoming traffic is "scanning attacks" (L3 or below), and then the Connection Management Engine further categorizes the remaining packets. The Connection Management Engine classifies the packet into four types ($T =< S, F, T, M, R >$) such that S indicates scanning attacks, F represents fingerprinting attempts and T, M, R are elements of T-Mode, M-Mode, and R-Mode, respectively. However, this syntax cannot account for each vulnerable services across diverse honeypots. We thus specify the destined service information ($S =< s, h, d, f \cdots >$) for more accurate analysis of malicious traffic where s, h, d, f stand for the ssh, http, database, and ftp services, respectively. In summary, the total number of SDN rules to process incoming traffic is therefore computed by $|T| \times |S|$. This information is recorded and used by the Mode Decision Module to take appropriate actions for attackers.

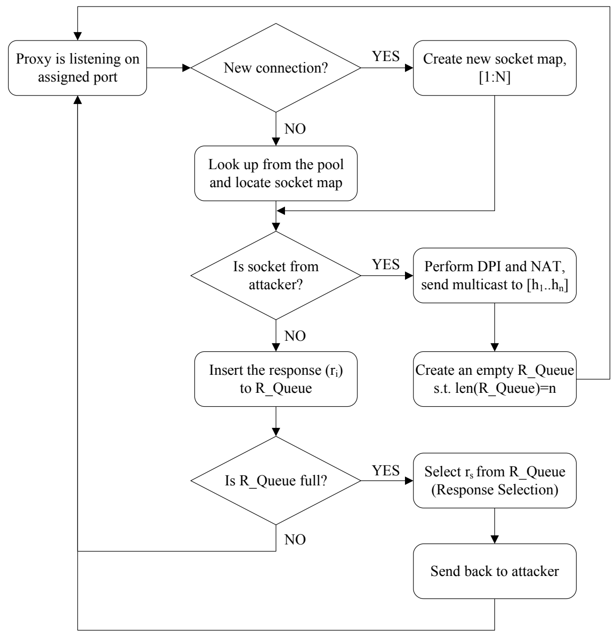
B. Blocking Malware Propagation

To block internal propagation of malware within the honeynet, traditional honeynet inserts host-based access control rules (e.g., iptables) in each honeypot machine to prevent potential malicious traffic from being generated. However, once a honeypot is compromised, the attacker can circumvent the host-based access control rules. To address this issue, HONEYPROXY uses a network-wide monitoring scheme and enforces access control rules via SDN controller instead of enabling host level protection. SDN rules are installed in the network to forward outgoing traffic to the specific honeypot that runs the intrusion detection system (IDS), such as snort. In this way, internal traffic between honeypots is also be monitored by IDS, consequently helping network administrators detect internal malware propagation. Note that the routing path of incoming traffic is not identical to that of outgoing traffic because the incoming packets would pass through IDS. Also, incoming and outgoing flows are physically separated by SDN rules as all incoming traffic is tagged by the proxy module. This mechanism is extremely useful for network administrators to manage the network and investigate security breaches.

VI. IMPLEMENTATION

We implement HONEYPROXY with a commonly used SDN controller, POX [10], along with KVM virtualization infrastructure to run a number of virtual honeypots. To maximize the performance of HONEYPROXY, we choose C language to build the proxy module. We also use Python for SDN applications as the language is supported by POX. As explained in Section III, the reverse proxy module has three subcomponents, and it runs a separate RESTful server to communicate with the SDN application over HTTP. Because the proxy module runs at TCP layer (L4), any services built on top of TCP also work, including http, ftp, and database services. It additionally supports transport layer security (TLS) to address https and ssh services. After configuring honeypots, the SDN application can instantiate a number of proxies by sending `/api/runproxy/[service]` request via RESTful API. Each proxy is bound to the specified port and serves one vulnerable service per proxy. If a proxy receives fingerprinting indicators, it notifies the controller with relevant data, along with associated connection identifier.

Next, we elaborate the packet processing logic of M-Mode in HONEYPROXY. We only explain M-Mode as T- and R-Modes are relatively straightforward to understand. First, a proxy instance listens on an assigned port. Upon the receipt of a new payload via a specific socket, the proxy checks affiliation of the socket. If the socket is not found from the existing socket pool, it means that a new attack has arrived at our honeynet, causing the proxy to create a new socket map (see Figure 5). Otherwise, HONEYPROXY locates the matching socket map from the pool. In case the socket is originated from the attacker, HONEYPROXY performs deep packet inspection



Proxy is listening on assigned port
New connection?
YES
Create new socket map, [1:N]
NO
Look up from the pool and locate socket map
Is socket from attacker?
YES
Perform DPI and NAT, send multicast to [h1..hn]
NO
Insert the response (ri) to R_Queue
Create an empty R_Queue s.t. len(R_Queue)=n
Is R_Queue full?
YES
Select rs from R_Queue (Response Selection)
NO
Send back to attacker

Fig. 6: Packet processing logic of M-Mode of HONEYPROXY.

(DPI) to search for any known fingerprinting attempts. It then makes a copy of the payloads and performs network address translation (NAT) to send multicast messages to all associated honeypots. Consequently, HONEYPROXY creates an empty receiving queue (*R_Queue*) for this socket map where size of the queue is set to the number of associated honeypots (N). Returning responses from honeypots are inserted into the R_Queue until it becomes full (i.e., all requests are returned). When all responses are collected (or a timeout is reached), HONEYPROXY chooses an appropriate response from the R_Queue and sends it to the attacker. Section IV-B discusses details of how HONEYPROXY selects an appropriate response.

VII. EVALUATION

Monitoring and analyzing payloads of incoming and outgoing packets requires a considerable amount of resources. In particular, when the data arrives at the proxy module of HONEYPROXY, it must conduct a pair-wise comparison with known fingerprinting attacks, thereby it could be a bottleneck for processing the requests. The fundamental question, hence, is to quantify the overhead of HONEYPROXY and how the overhead affects the behaviors of attackers. To answer the question, we consider three test metrics while conducting the experiments: (1) throughput (Gbits per second), (2) latency (milliseconds), and (3) CPS (connections per second). We first introduce our testbed followed by the detailed experimental results.

A. Test Environment

Figure 7 illustrates the testbed setup for the evaluation. Our testbed consists of two physical machines, each of which has Intel Xeon CPUs (E5-2658v3 @ 2.20GHz, 24-cores) and 128GB RAM. One machine runs HONEYPROXY (the proxy module and HONEYPROXY controller) on Ubuntu 16.04 LTS

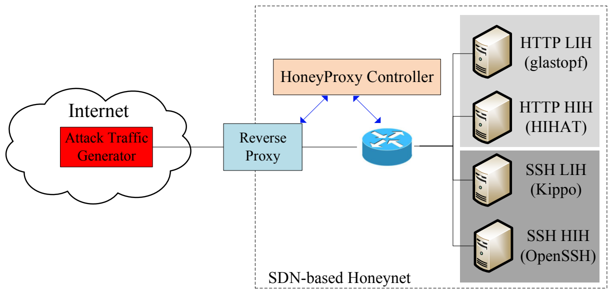


Fig. 7: Testbed network configuration.

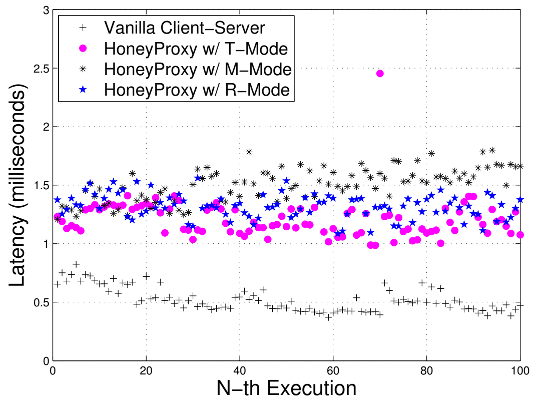


Fig. 8: File transmission Latency incurred by HONEYPROXY

(Linux kernel v.4.4.0) and the other runs KVM virtualization infrastructure on the same OS to emulate a set of honeypots. To create a network of honeypots, we used a software switch (OpenvSwitch v.2.5.0 [8]) that can act as an SDN switch. All incoming traffic destined to our SDN-based honeynet is considered malicious and therefore, passes through HONEYPROXY. As shown in Figure 7, HONEYPROXY interconnects the external network (Internet) and honeynet by relaying the packets from the Internet to honeynet or vice versa.

To effectively run the experiments, our testbed was configured to run two representative services: http and ssh. For the http service, we chose a widely deployed LIH, glostopf [3], and HIHAT [2] for a HIH. To run the ssh service, kippo [7] was selected as a LIH, and OpenSSH was used to mimic a ssh HIH. Each honeypot is configured to have 4 vCPUs along with 4GB RAM, and the link speed for every honeypot was set to maximum 10 Gbps.

B. Performance of HONEYPROXY

Figure 9 shows TCP and SSL *throughput* results of HONEYPROXY using iperf [6]. HONEYPROXY achieved 8.23 Gbits per second (Gbps) in T-Mode, while M-Mode and R-Mode showed 4.5 and 7.79 Gbps, respectively. It is worthwhile to note that M-Mode creates a copy of malicious payloads and sends multicast messages, and therefore it requires a considerable amount of resources compared to the others. Additionally, the throughput is decreased by 57 percent when using SSL protocol as it obviously handles encryption and decryption of data.

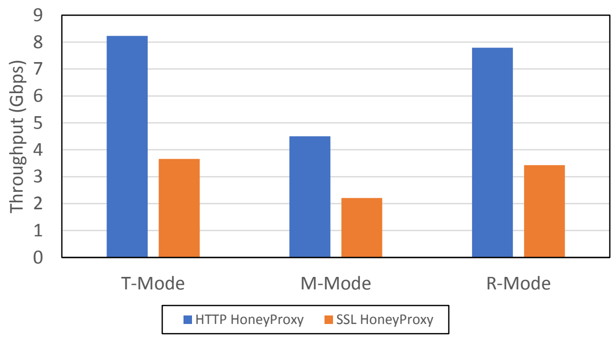


Fig. 9: Throughput of HONEYPROXY with respect to three different running modes.

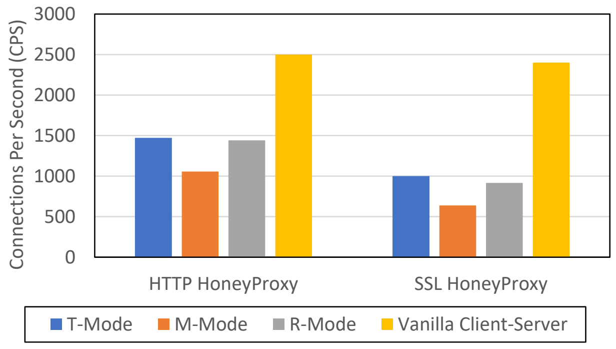


Fig. 10: CPS handled by HONEYPROXY with respect to three different running modes.

We also take *latency* as one of the test metrics to measure the performance, because latency can show end-to-end responsiveness between attackers and honeypots while *throughput* metric measures the performance with respect to a massive data transaction. To conduct these experiments, we implement a socket server and client to measure the latency. The client sends a hello message to the server, and the server responds back with an echo message. We measure the round trip time (*RTT*) for the client to send and receive the message. To obtain ground truth, we first run the custom server on one of the honeypots and measured the latency without running HONEYPROXY. As illustrated in Figure 8, RTT took 0.5 milliseconds on average. When HONEYPROXY is enabled, the average RTT is observed in the range of 1 and 1.7 milliseconds. Each mode shows 1.24, 1.50, and 1.32 on average (T-Mode, M-Mode, and R-Mode respectively). From these results, we can expect $0.5 - 1.2$ milliseconds delay incurred by HONEYPROXY. We then conduct the identical experiments over internet to measure how much latency can be attributed to different geolocational network access. Interestingly, RTTs are not distinguishable even if the client (attackers) access from internet. In this case, consistent duration time is measured at approximately 166 milliseconds regardless of the mode used.

Finally, connections per second (*CPS*) is also incorporated to evaluate the performance of HONEYPROXY. CPS is measured by allowing a flood of requests reaching honeypots via HONEYPROXY. Both incoming and outgoing connections along with the nanosecond time precision are continuously recorded during the period of 60 seconds. Figure 10 illustrates the experimental results of CPS for each mode, along with the results for vanilla setup (i.e., the setup without HONEYPROXY). Without HONEYPROXY, the CPS handled by honeypots are 2500 and 2410 for HTTP and SSL respectively. With HONEYPROXY, CPS of HTTP HONEYPROXY are 1473, 1058, and 1443, while that of SSL HONEYPROXY are 1000, 640, and 917 for T-, M-, and R-Mode respectively. The results are consistent with that of throughput — since M-Mode multicasts the incoming messages, it shows slightly decreased number of CPS for both HTTP and SSL HONEYPROXY. Compared to the vanilla results, degradation of CPS by using HONEYPROXY could range from 41 to 55 percent. Our experimental results demonstrate that it would be difficult for attackers to identify the operating mode of HONEYPROXY remotely.

VIII. RELATED WORK

SDN provides a global view and centralized control mechanisms to SDN applications. In addition, SDN can help provide flexibility in monitoring and controlling untrusted traffic within honeynet. We leverage the SDN paradigm in our design and implementation to centrally monitor and route packets to honeypots, thereby supporting internal traffic monitoring and mitigate the risk of internal malware propagation.

Honeypot farm [4] is an approach which involves deployment of many virtual honeypots in a network. Any malicious traffic directed to the real network will be sent to the dedicated group of honeypots in the network without knowledge of the attacker. However, this approach only redirects the malicious traffic to the honeypot farm and does not provide any data control mechanisms. In addition, it is also vulnerable to internal propagation of malware.

Honeybrid [5] is an architecture which is closely related to our approach which uses connection migration between low-interaction and high-interaction honeypots to take advantage of the functionalities provided by both types of honeypots. However, honeybrid has a few design flaws. In their mechanism, only the first scanning attacks are handled by low-interaction honeypot, and the rest of connection are relayed to a high-interaction honeypot. Therefore,only the high-interaction honeypots are active during majority of the connection time. On the other hand, HONEYPROXY routes traffic dynamically based on its analysis of the incoming payloads. In this way, the routed traffic is handled by either low-interaction or high-interaction honeypot at any given time.

Collapsar [19] enables a VM-based honeyfarm architecture, which consists of a group of virtual honeypots. It aims at providing centralized administration, efficient data classification, and distributed view of honeypots. Though this architecture succeeds in providing centralized monitoring of the honeypots, it does not support connection migration between low-interaction and high-interaction honeypots, which becomes important when dealing with a large scale of various attacks to the system.

HONEYPROXY is greatly influenced by HoneyMix [18], which presents a native SDN-based honeynet architecture. HoneyMix involves deployment of various custom modules in the SDN controller for dynamic connection selection and prevention of fingerprinting attack. However, HoneyMix lacks mitigation mechanism for internal malware propagation and more importantly, does not support transition between honeypots, which is one of the core functionalities of next-generation honeynet for encouraging more interaction between attackers and honeypots.

IX. DISCUSSION

HONEYPROXY does not fix the defects and vulnerabilities in individual honeypots [19]. Our assumption is that by having honeypots that is foolproof for fingerprinting attack, we could take advantage of the uniqueness of each honeypot by connecting them to an entity, which we could centrally monitor and respond to the attacker behind a honeywall. Thus, reducing the chances of revelation of honeynet enables network administrators to collect sufficient information that helps detect, prevent and secure the network infrastructure. In addition, HONEYPROXY relies on intrusion detection systems (IDS) for detecting internal malware propagation by configuring way point of internal traffic. It means that an advanced malware might not be filtered by our approach, depending on the capability of IDS.

X. CONCLUSIONS

In this paper, we articulate the limitations of existing honeypots and honeynet architecture: (1) vulnerable to fingerprinting attacks, (2) internal malware propagation, and (3) lack of honeypots transition. To overcome the shortcomings, we present an SDN-based honeynet architecture called HONEYPROXY as a next generation honeynet. HONEYPROXY consists of a reverse proxy module and a corresponding SDN application. The core of the reverse proxy is a novel Connection Management Engine that selects the response and enables dynamic transitions between low-interaction and high-interaction honeypots. To address internal malware propagation, HONEYPROXY uses a flow programming scheme that is implemented in the SDN application. With our prototype implementation, experimental results demonstrate that HONEYPROXY is able to support the near line rate throughput (8.23 Gbps) with neglibible latency ($0.5 - 1.2$ milliseconds) and handle enough number of connections (1473 CPS).

ACKNOWLEDGEMENT

This work was partially supported by grants from National Science Foundation (NSF-ACI-1642031).

REFERENCES

[1] Black Hat USA 2015 - Breaking Honeypots For Fun And Profit. https://www.youtube.com/watch?v=Pjvr25lMKSY.
[2] GHihat-Honeypot. http://hihat.sourceforge.net/.
[3] Glastopf Honeypot Project Page. http://glastopf.org/.
[4] Honeypot Farms. http://www.symantec.com/connect/articles/honeypot-farms.
[5] Hybrid Honeypot Framework. http://honeybrid.sourceforge.net/.
[6] Iperf. https://iperf.fr/.
[7] Kippo SSH Honeypot. https://github.com/desaster/kippo.
[8] Open vSwitch. htpp://openvswitch.org/.
[9] openflow. https://www.opennetworking.org/sdn-resources/openflow.
[10] POX Wiki-OPen Networking Lab-Confluence. https://openflow.stanford.edu/display/ONL/POX+Wiki.
[11] Snort.Org. https://www.snort.org/.
[12] 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.
[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] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral. Deep packet inspection as a service. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 271–282. ACM, 2014.
[15] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: a retrospective on evolving sdn. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 85–90. ACM, 2012.
[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] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 19–24. ACM, 2013.
[18] W. Han, Z. Zhao, A. Doupé, and G.-J. Ahn. Honeymix: Toward sdn-based intelligent honeynet. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 1–6. ACM, 2016.
[19] X. Jiang, D. Xu, and Y.-M. Wang. Collapsar: A vm-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention. *Journal of Parallel and Distributed Computing*, 66(9):1165–1180, 2006.
[20] T. K. Lengyel, J. Neumann, S. Maresca, and A. Kiayias. Towards hybrid honeynets via virtual machine introspection and cloning. In *Network and System Security*, pages 164–177. Springer, 2013.
[21] 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.
[22] J. Palach. *Parallel Programming with Python*. Packt Publishing Ltd, 2014.
[23] C. Seifert, I. Welch, P. Komisarczuk, et al. Honeyc-the low-interaction client honeypot. *Proceedings of the 2007 NZCSRCS, Waikato University, Hamilton, New Zealand*, 2007.
[24] L. Spitzner. The honeynet project: Trapping the hackers. *IEEE Security & Privacy*, (2):15–23, 2003.
[25] L. Spitzner. Honeypots: Catching the insider threat. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 170–179. IEEE, 2003.
[26] G. Tsirtsis and P. Srisuresh. Network address translation-protocol translation (nat-pt). Technical report, 2000.