DESIR: Decoy-Enhanced Seamless IP Randomization

Jianhua Sun, Kun Sun
Department of Computer Science,
College of William and Mary
Email: {jianhua, ksun}@cs.wm.edu

***Abstract*—Sophisticated adversaries usually initiate their attacks with a reconnaissance phase to discover exploitable vulnerabilities on the targeted networks and systems. To mitigate the effectiveness of persistent reconnaissance attacks, we develop a defensive mechanism that dynamically mutates network topology with a large number of decoys to invalidate the attacker's knowledge from network scanning. We combine the IP randomization technique with decoy techniques and solve two challenges, namely, service availability to legitimate users and service security against unauthorized users. First, our solution can minimize the probability of the real servers being identified and compromised by unauthorized users through deploying a large number of decoy nodes, which change their IP addresses along with the real servers to prolong the scanning time of the attackers. Second, our solution can ensure seamless connection migration so that all existing communication connections between the legitimate users and the servers are always kept alive even after the servers migrate to different IP addresses multiple times. We implement a virtual machine based system prototype and evaluate it using state-of-the-art scanning techniques. Both theoretical analysis and experimental results show that our solution can effectively mitigate network reconnaissance attacks without sacrificing service availability.**

I. Introduction

In advanced persistent threat (APT), well-resourced and trained adversaries typically initiate the attacks with thorough reconnaissance to gather intelligence about the targeted networks and systems. Once a vulnerability is identified, the adversary can proceed to mount customized exploits to compromise the system. This attacking strategy has been working well due to the static nature of the current network configurations.

In recent years, researchers have proposed to mitigate reconnaissance attacks by dynamically shifting the network attack surface [1] including IP and MAC addresses, open ports, and network topology [2], [3], [4]. In general, by proactively changing the host IP addresses and the network topology, the entire network can be made unpredictable so that the vulnerabilities discovered by an attacker in an early stage become obsolete and useless. However, all those IP randomization based solutions face two challenges. First, though the size of available IP address pool is large, due to the small number of alive IP addresses at one time, the attackers may still complete scanning the entire targeted network quickly and compromise the targeted system before the next round of IP randomization. For instance, ZMap is capable of surveying the entire IPv4 address space within 45 minutes from a single machine [5]. Second, when the servers change their IP addresses, existing active connections may be disrupted, since high-layer protocols such as TCP or UDP depend on a stationary IP address [6]. Therefore, it is a challenge to seamlessly migrate all existing network connections to the new IP addresses with a minimal migration time.

In this paper, we develop a decoy-enhanced seamless network address randomization mechanism called *DESIR* to defeat network reconnaissance attacks and ensure service availability. First, we fortify the IP randomization technique with a large number of decoys to protect the servers against reconnaissance attacks. Besides the real servers, we deploy a number of decoy nodes [7] that will change their IP addresses along with the real servers. Decoys have been widely used to distract attacker's attention from the real system; however, APT attacks may eventually identify the decoy nodes based on their response time and fingerprint analysis after interacting with the decoys [8], [9]. In our solution, in addition to deploying a large number of decoys, we randomly shuffle the IP address space of the target network including both the real servers and the decoys. Therefore, though the attacker may create a blacklist of decoy IP addresses through reconnaissance, this blacklist becomes invalid after the next round of IP randomization, and the attacker has to start over the reconnaissance process. In other words, we combine both IP randomization technique and decoy technique to effectively defeat persistent reconnaissance attacks, though neither of them can achieve this goal by itself only.

Second, we develop a seamless network connection migration mechanism to keep alive the existing connections between legitimate users and the servers even after the servers change their IP addresses multiple times. The basic idea is to separate the connection's transport identification from its network identification so that the dynamic changes of network addresses are transparent to the transport layer and the application layer. We introduce a pair of internal addresses to identify the transport endpoints and another pair of external addresses to identify the network endpoints. The internal address remains consistent during the life of the connect session and the external address is changed as the server migrates. Moreover, we guarantee that the legitimate users can always locate the servers and initiate service requests by using a trusted authentication server. Whenever a server changes its IP address, it will notify the updated IP address to the authentication server. When a

client wants to connect to the real server, it first authenticates itself to the authentication server, which then sends the server's current IP address to the client.

We evaluate the effectiveness of our decoy-enhanced IP randomization mechanism through both theoretical analysis and real prototype implementation. Our theoretical analysis shows that decoy-enhanced IP randomization can effectively prolong the attacker's scanning time. Suppose one real server is protected by n IP addresses, where $n-1$ IP addresses are occupied by decoy nodes. When the attacker is not aware of our defense mechanism, it may only scan the entire IP address space once either sequentially or randomly. In this scenario, our IP randomization technique can increase the average number of probes from $0.5n$ to $0.63n$. When the attacker knows that the real system is protected by our defense system, it may scan the entire IP address space multiple times, and it will increase the average number of probes from 0.5 to n. More importantly, the attacker has to spend tremendously more time to distinguish the decoys from the real system, and there is high probability that the attacker will be trapped into one decoy instead of the real server.

We implement a virtual machine (VM)-based prototype that integrates decoy-enhanced IP address randomization with seamless connection migration. The experimental results show that the overheads for both decoy deployment and IP randomization are reasonably low and can defeat the practical scanning attacks using tools such as Nmap [10] or ZMap [5].

In summary, we make the following contributions:

- We propose a decoy-enhanced seamless network address randomization framework for constructing dynamically mutable networks to thwart persistent reconnaissance attacks against targeted servers.
- Our solution supports seamless connection migration with network address randomization. It has good scalability to seamlessly migrate a large number of network connections after the servers change their IP addresses multiple times.
- We implement a VM-based prototype. The experimental results show that the system overhead is small and our system can effectively defeat persistent reconnaissance attacks.

II. Threat Model and Assumptions

An adversary may undertake various scanning strategies with abundant resources. When the adversary believes that the IP address space layout is fixed, it may scan the IP address space of the targeted network either linearly or randomly and probe each IP address only once. After the first round of quick probing, the adversary may collect further information and exploit vulnerabilities against those alive IP addresses. Alternatively, when the adversary realizes that IP randomization mechanism has been deployed to protect the real servers, it may randomly probe an IP address and immediately exploit potential vulnerabilities against it if it believes the IP address belongs to an alive server. Note in this scenario the adversary may probe the same IP address more than one time.

This paper focuses on defeating persistent reconnaissance attacks, so we do not consider insiders that deliberately disclose the current server IP address to attackers. Also, we assume the adversaries are not in the same subnet with legitimate users, so that they cannot obtain the server IP addresses through packet eavesdropping. We assume secret keys are shared between the legitimate users and the servers. We assume the protected network consists of a large number of IP addresses to accommodate decoy nodes. This can be satisfied in private IPv4 networks and IPv6 networks. The main purpose of deploying decoys is to prolong the attacker's scanning time, and we have no interests on attracting and profiling attacks. The adversary may compromise a decoy node and misuse it to attack the real servers or the other decoy nodes.

III. System Architecture

Figure 1 shows the system architecture of DESIR, which is composed of four major components, namely, an authentication server, a protected server pool, a decoy bed, and a randomization controller. The authentication server grants service access to the client after successfully authenticating the client's credentials. The server pool contains all the servers that need to be protected. The decoy bed controls a large number of decoys. A centralized randomization controller is responsible for controlling the entire network address randomization process among both the real servers and the decoys.

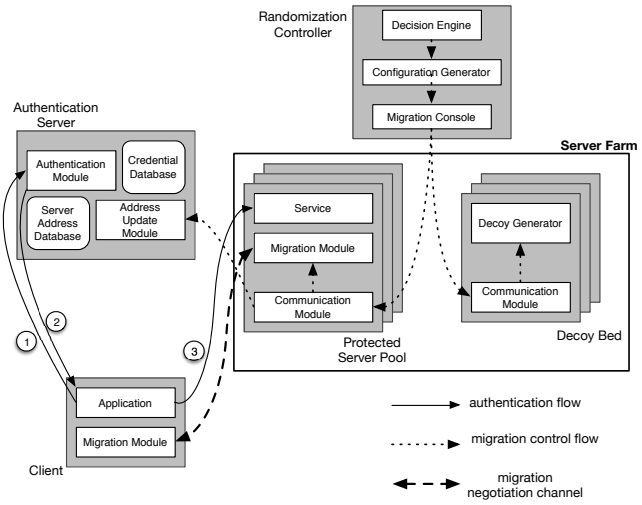

Fig. 1: System Architecture

A. Authentication Server

The authentication server consists of an *authentication module* and an *address update module*. As shown in Figure 1, the client needs to first provide its security credential to the authentication server before accessing the server. The authentication module verifies the client's identity after checking the credential database. Once the client is authenticated, the authentication server sends the server's current IP address to

the client. Otherwise, the access request is denied. Next, the client proceeds to connect to the server using the obtained server IP address. Whenever the server migrates to a new IP address due to network address randomization, it immediately informs the authentication server, whose address update module will update the server address database that stores all the real servers' most updated IP addresses.

B. Randomization Controller

The randomization controller is responsible for coordinating the dynamic mutation of the target network, such as determining the size of the server farm, assigning IP addresses to the server farm, and setting the frequency of shuffling the IP address space, etc. It consists of three modules: a *decision engine*, a *configuration generator*, and a *migration console*. The decision engine determines the frequency to randomize the network addresses and chooses the algorithm for generating new network configurations. The configuration generator focuses on generating the new network configurations for both the real servers and the decoys. It has two main functions. First, it controls the overall topology of the network, such as the deployment of virtual routers and switches. Second, it guarantees that there is no interference in the IP address assignment. The migration console is responsible for distributing the new configurations to the servers and the decoy subsystem. After receiving the new configurations, the real servers update their network addresses and notify the connecting clients about the network migration.

C. Decoy Bed

The decoy bed generates a number of decoy servers, which share the same IP address space with the protected real servers. It contains a *communication module* and a *decoy generator*. The communication module receives new configuration settings from the randomization controller, which determines the overall architecture of the decoy network, including the decoys' IP addresses and MAC addresses as well as the installed or emulated operating systems and applications. According to the new configuration, the decoy generator regenerates the decoy network. Our system is flexible to deploy both high-interaction and low-interaction honeypots as decoys depending on the system's resources and the configurations sent from the controller. Originally, honeypots are deployed to attract attackers and learn their attacking strategies; while the honeypots in our system are used mainly to confuse the attackers and prolong their scanning time.

D. Protected Server

The real server includes one *migration module* that communicates with the counterparts on the authenticated users to achieve a seamless connection migration when the server moves to a new IP address. To minimize service disruption caused by network address randomization, we develop a seamless connection migration mechanism that leverages the virtual network address translation concept [11] to maintain all existing connections alive after the server migrates to different addresses multiple times. Therefore, the normal users do not need to re-initiate new service requests as the IP randomization is performed.

Since transport layer protocols depend on stationary IP address and port number, an end-to-end transport socket connection will be broken if one or both connection endpoints change their network addresses. Accordingly, we separate the transport identification of a connection from its network identification to enable transparent connection mobility. We introduce a pair of internal addresses to identify the transport endpoints and another pair of external physical addresses to identify the network endpoints. The internal address is set to be the address when the connection is initially established and remains consistent during the life of the connection. The external address can change as the server migrates. By separating the transport endpoint identity from the network endpoint identity, the transport layer protocols get the illusion that the network endpoints never move. Our seamless migration scheme consists of three major components:

- *connection interception*: It creates an initial internal-external address mapping and replaces the external physical address provided by the application with the internal address. As a result, the transport protocol stacks on both the client and the server perceive a connection identified by the internal addresses. Since the identity of the transport endpoint is detached from the network endpoint, the movement of the physical host is transparent to the transport layer or higher layer protocols.
- *connection translation:* A connection is only ready to be migrated after being intercepted. However, the actual traffic is routed through the network using the external address. To allow network packets flowing through internal connections, the connection translation component intercepts packets in the network layer and translates the internal addresses in the packet headers to or from the external addresses for outgoing packets and incoming packets, respectively.
- *connection migration:* It coordinates the moving of an endpoint associated with active connections to another place. The process involves first suspending an active connection at one location and later resuming it at another. To suspend a connection, the migration module on the moving endpoint saves the current state including the internal-external address mapping and notifies the endpoint on the other endpoint about this event. When a connection is resumed, the migration module updates its address mapping and notifies the other endpoint the new external address. Meanwhile, the migration module inserts appropriate translation rules into the network stack.

IV. Implementation

We implement a DESIR prototype using virtual machines on one host computer. Figure 2 shows the detailed implementation architecture. The entire system is integrated on a single host machine running Ubuntu 14.04 with support for

Kernel-based Virtual Machine (KVM) enabled. The computer features eight core Intel(R) Core(TM) i7-4712HQ CPU and 16 GB memory. Five virtual machines are created on top of the KVM hypervisor, serving as decoy bed, a real server, an authentication server, a client, and an attacker, respectively. Each VM is allocated one host CPU and 2 GB memory. The decoy bed VM is running Ubuntu 12.04; while the other VMs are installed Fedora 15 with Linux kernel version 2.6.38.

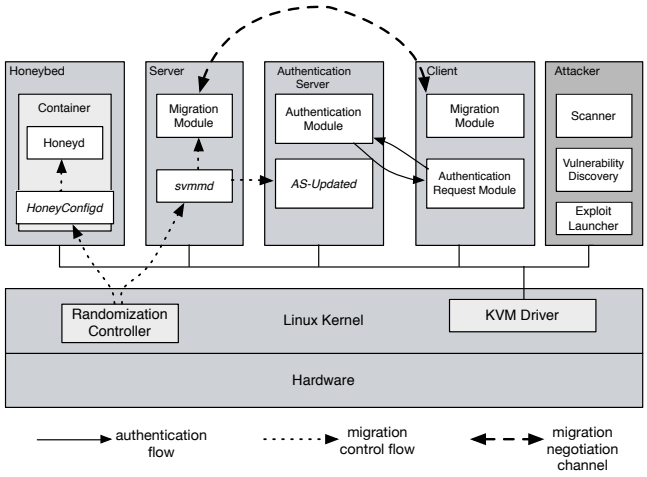

Fig. 2: Prototype Implementation

A. Authentication Server

The authentication server maintains a credential database for all legitimate users and a server address database. In our prototype, the authentication module uses password-based authentication and sends its decision back to the client in an encrypted message. The server address database is updated by a daemon named *AS-Updated* without any manual intervention. *AS-Updated* listens on a randomly generated port for address update messages from a daemon named *svmmd* on the migrating server. Once receiving the message, *AS-updated* randomly generates a new port number and notify it to *svmmd*. Then it starts to listen on the newly generated port for future updates. All the exchanged messages are encrypted using a shared secure key.

B. Three-layered Decoy Bed

We implement a specially designed decoy bed that can integrate a large number of low-interaction and high-interaction honeypots on a single host. Specifically, three levels of decoys are incorporated into the decoy bed: *virtual machine level*, *operating system level* and *process level*. The VM-level decoys are virtual machines with fully functional operating systems and applications. The OS-level decoys run in Linux Containers (LXC) [12]. Each container in the same VM shares the same OS kernel but has its own user level space. We employ OS container for the following reasons. First, it is lightweight, resource-friendly, and thus highly scalable. Multiple OS instances or applications can run simultaneously on a single host without incurring excessive CPU and memory overhead. Second, it supports resource and security isolation. A compromised container does not affect the remaining containers. In the prototype, we use containers to hold both high-interaction honeypots and low-interaction honeypots (e.g., honeyd [13]) that run as a process in the container.

Honeybed VM serves as the base layer on which virtual decoys are deployed. We use Honeyd for setting up and emulating virtual hosts. However, we do not deploy Honeyd directly on Honeybed. Instead we create multiple Linux containers (LXC) in Honeybed and deploy Honeyd in the containers. By default, a bridge `lxcbr0` is created after booting up the host. A `dnsmasq` instance is run listening on `lxcbr0` and provides DNS and DHCP services to the containers. Since we want to have a full control of the entire network configuration, we do not use `lxcbr0` and `dnsmasq`. Instead we create a network bridge `br0` and attach it to the `eth0` interface of Honeybed. Furthermore, we change the IP address assignment scheme of the Honeyd container to be static by modifying the LXC configuration file. The configuration of Honeyd in each Linux container is managed by a centralized randomization controller on the KVM host.

C. Seamless Connection Migration

We discuss how a transport connection can be seamlessly migrated and elaborate on the implementation details of the three key components.

1) Connection Interception: For the stateful TCP-based protocols, we intercept the system calls for connection setup from the application layer to the transport layer. Those system calls include `socket`, `accept`, `connect`, `close`, `getsockname` and `getpeername`. The address mapping together with other states about the internal connection are saved for suspending and resuming the connection later. When the connection is closed, its associated address mapping states are cleaned up. We intercept those system calls at INET sock layer instead of BSD socket layer to avoid dealing with userspace objects such as socket descriptors, which requires routines that are not exported by the Linux kernel. We replace the corresponding kernel functions with our customized versions, which can make the movement of network endpoints imperceptible to the transport protocols by separating the transport identification of a connection from its network identification.

To migrate UDP-based connections, we intercept `getsockname` and `getpeername` system calls to hide the changes of the external physical addresses from the applications. However, this solely cannot support the migration of servers providing UDP-based services. Due to the lack of the connection establishment process, the initial internal-external address mapping cannot be created. So it is impossible to insert *Netfilter* translation rules and maintains the "connections" alive. Since most UDP-based applications involve extensive data exchange, besides the command channel, we can intercept potential separate data channels to retrieve necessary information for constructing

the address mapping. Specifically, we intercept the related socket system calls including `send`/`sendto`/`sendmsg` and `recv`/`recvfrom`/`recvmsg` by instrumenting the underlying INET sock layer functions `inet_sendmsg` and `inet_recvmsg`. By examining the binding sockets and the sending/receiving message headers, we can identify the addresses of both the local host and the remote peer and then create the initial address mapping. For UDP sockets binding to `INADDR_ANY`, we perform route lookup to find the packet source address based on the destination address.

2) Connection Translation: Netfilter enables packet filtering, network address translation (NAT) and other packet mangling. The most popular implementation of *Netfilter* in Linux is *iptables*, which matches each packet against a chain of rules, each consisting of a matching criteria and a target that specifies what to do with the packet once it matches the criteria. *iptables* organizes rules into four tables: the `nat` table, the `mangle` table, the `filter` table, and the `raw` table.

The actual traffic is routed through the intermediate network using the external physical address of the migrating endpoint. To ensure smooth packet flow, the internal connection needs to be translated to and from the external connection for outgoing and incoming traffic, respectively. We use the `nat` table for network address translation. For outgoing packets, we perform destination address translation (DNAT) on the OUTPUT chain and source address translation (SNAT) on the POSTROUTING chain. For incoming packets, we perform DNAT on the PREROUTING chain and SNAT on the INPUT chain. We also use the `mangle` table to block any connection attempt to the server's internal address. As a result, the server cannot be accessed through its old addresses after migrating to a new address. When the connection is being resumed after the migration, we instrument the server side PREROUTING chain to discard all packets destined to the old server address simply by adding a rule on the PREROUTING chain with the DROP target. Since the `mangle` table rules are applied ahead of the `nat` table rules, the address translation is not affected so that the migrated connection can still be maintained.

3) Connection Migration: Two daemon threads running in the kernel space of both endpoints negotiate with each other about the migration based on a set of predefined control protocol. The communication ports are synchronously randomized between the daemons. To avoid information leakage, the protocol messages exchanged between the two daemons are encrypted using a shared secret key. The migration involves suspending the connection at one location and resuming it later at another one. To suspend a connection, the daemon within the migrating endpoint needs to clean up the internal-external address mapping and destroy the virtual interface. To resume a connection, the daemons need to update the address mappings and recreate a new virtual interface on the moving endpoint.

In case that multiple clients connect to a server at different times, we create a virtual network interface on the server side for each connection. Each virtual interface's IP address is set to be the server IP when the connection is initially established and is kept consistent as the server migrates. Since the granularity of creating the virtual interface is per connection, if two applications initiate a connection separately when the server is at different locations, these two connections will be associated with different virtual interfaces. Therefore, we can ensure that multiple connections created by different applications at different times can be kept alive after a number rounds of migrations.

4) Removing Migration Residues: Besides IP address and port number, adversaries may perform a correlation between two hosts using other network layer information such as MAC address or application layer information such as SSL certificates. Thus, we should be careful to remove all the potential migration residues that may be exploited by the adversaries. For instance, in addition to the IP address, it is not difficult to update the MAC address that has been loaded from NIC device to RAM memory.

When an attacker uses Nmap to scan two IP addresses used by the same host, it can discover that an SSL certificate remains the same before and after the address change, so it can use this correlation information to defeat the IP address randomization. It happens because sshd does not automatically change its key pairs when the server moves to a new IP address. We solve this problem by refreshing the sshd key pairs without breaking the existing connections. After regenerating new rsa1, rsa and dsa keys using "ssh-keygen" command and restarting sshd service, a set of different keys will be set for the new IP address, and the existing ssh connections won't be impacted.

D. Randomization Controller

The controller is implemented on the hypervisor. The decision engine proactively shuffles the IP address space, and the configuration generator generates the new Honeyd configurations according to the shuffled address space. Then, the migration console distributes the new addresses and configurations to the protected servers and the decoy bed. In our implementation, the messages sent by the controller are encrypted using a shared secret key between the controller and the server/decoy bed.

A daemon named *HoneyConfigd* is created in each Linux container running Honeyd. *HoneyConfigd* listens on a randomly generated port for the new Honeyd configuration sent from the migration console. Upon receiving the new configuration file, *HoneyConfigd* invokes a script to terminate the existing Honeyd process and restart a new one with the new configuration. A daemon named *svmmd* is created on each server. *Svmmd* listens on a randomly generated port for the server's new IP address from the randomization controller's migration console. Once *svmmd* receives the new address, it informs the authentication server using the *AS-Updated* daemon and invokes the migration module to migrate the server.

V. Security Analysis

We first perform a theoretic analysis on the effectiveness of our decoy-enhanced IP randomization against persistent

reconnaissance attacks, and then we discuss the potential attacks towards our system and the countermeasures.

We analyze the expected number of probes for adversaries to identify the real server with and without the knowledge of the deployment of our defense system, respectively. For both cases, we suppose the IP address space is n and there is only one real server.

Scanning without knowing IP Randomization defense. After scanning an IP address in static networks, the adversary typically won't scan it again. Therefore, it can be considered as a sampling without replacement problem [14]. When the IP address space is static, the probability to identify the real server after m probes is

$$\underbrace{\frac{n-1}{n}\cdot\frac{n-2}{n-1}\cdots\frac{n-(m-1)}{n-m}}_{Pr[first\ m-1\ probes\ fail]}\cdot\frac{1}{n-(m-1)}=\frac{1}{n}$$

where $m \leq n$. Therefore, the expected number of probes required is

$$\sum_{m=1}^{n} m\cdot\frac{1}{n}=\frac{1}{n}\cdot\sum_{m=1}^{n} m=\frac{n+1}{2}$$

Now Let us see the impacts of IP randomization on the adversary's success probability. In this case we assume the adversary is not aware of the deployment of our defense mechanism. The best we can do is to randomize the IP address space after each probe of one IP address, so we can maximize the probability that adversary fails each single probe equals to $\frac{n-1}{n}$. In this case, the probability that the target server is identified after exactly m probes is

$$(\frac{n-1}{n})^{m-1}\cdot\frac{1}{n}$$

Therefore, the expected number of probes is

$$\sum_{m=1}^{n} m\cdot(\frac{n-1}{n})^{m-1}\cdot\frac{1}{n}+n\cdot(\frac{n-1}{n})^{n}$$

which approaches $(1-\frac{1}{e})n \approx 0.63n$ even for a small IP address space of 64. We see that IP randomization can at most prolong 26% more scanning time than a static setting.

Scanning with the knowledge of IP Randomization defense. Since the adversary knows that the real server may be protected by IP randomization technique, it may randomly probes an IP address from the IP address pool. It can be considered as a sampling with replacement problem [14]. In other words, a single IP address may be scanned twice for one round of random scanning. In this case, the number of probes performed by the the adversary is a geometric random variable with probability $p = 1/n$. Therefore, the expected number of probes is $1/p = n$. Similar result has been presented in [15] when analyzing the effectiveness of address space layout randomization (ASLR) against buffer overflow attacks.

In summary, IP randomization can reduce the effectiveness of scanning attacks and forces the attacker to spend 26% more efforts when it is not aware of the IP randomization protection and 100% more efforts when it knows the deployment of IP randomization protection. The explanation of this counter-intuitive results is that the adversary only scans each IP address at most once for a static network and may scan some IP addresses twice or each more times for a dynamic network. On the other side, to make the IP randomization scheme useful against reconnaissance attacks, we should have a large IP address space available to protect the real server. We can also randomize the m port numbers on n IP addresses and thus increase the scanning space from n to $m * n$. Note this is only the expected number of probes to be able to scan the real server. Since we have deployed a number of decoys, the attacker has to spend more time to distinguish a decoy from the real server, and this time delay depends on the fidelity of the decoys simulating the real servers. In other words, high-interaction decoys may trap the attacker for a long time, but it may demand more system resources.

Our framework contains several communication channels for client authentication and migration control. The external adversaries cannot obtain the server IP addresses through eavesdropping attacks, since all exchanged messages are protected with a secret key shared by two endpoints. Our current system cannot prevent an attacker locating in the same network as a normal client from eavesdropping the messages between the client and servers and steal the server's most updated IP addresses. One potential solution is to add a layer of proxy nodes between the client and the server, so that the attacker can only know the IP address of the proxy node, whose IP address may also be changed periodically. We consider this extension as a future work.

VI. Performance Evaluation

We measure the performance overhead incurred by network address migration and its breakdown for both TCP and UDP based connections, and they are shown in Table I. We separate the applications by if they support constant data transfer. For applications such as $udpchat$ and ssh, they provide live sessions to execute commands remotely, and they don't need to support constant data transfer. In contrast, *ftps* is an extension to the traditional *ftp* file transfer service with support for Transport Layer Security (TLS) and Secure Socket Layer (SSL). *sftp* is a secure file transfer service based on Secure Shell protocol (SSH) version 2.0. Both services are TCP-based. Alternatively, *tftp* is a simple, lock-step, file transfer service based on UDP.

TABLE I: Microbenchmarks

Application	Name	Protocol	Const. data transfer
Secure Shell	*ssh*	TCP	No
FTP over SSL	*ftps*	TCP	Yes
Secure File Transfer	*sftp*	TCP	Yes
UDP-based chat	*udpchat*	UDP	No
Trivial File Transfer	*tftp*	UDP	Yes

A. System Overhead

We use *sftp*, *ftps* and *tftp* to transfer a 1 GB file and measure the extra transfer delay when migrating the connections with different migration frequency. Figure 3 shows the relative performance with respect to the number of migrations.

The performance degradation increases linearly as the number of migrations increases. For *sftp* and *ftps*, migration incurs 1% to 9% overhead. For *tftp*, the overhead is much larger ranging from 5% to 15%. This is due to the difference in implementation for UDP-based applications. To construct the internal-external address mapping for connection migration, the UDP message sending and receiving process is intercepted. For TCP-based file transfers, the mapping is only created during the connection setup process, and the data traffic is not intercepted.

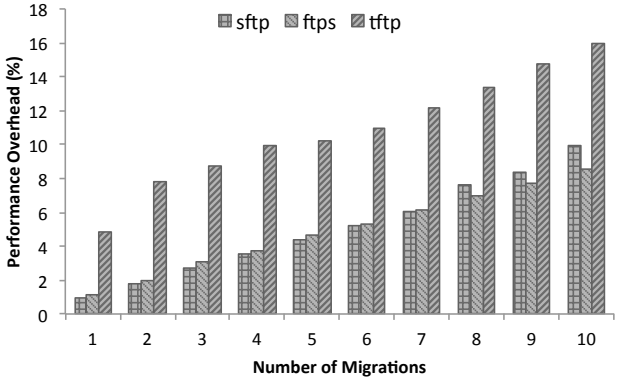

Fig. 3: Delay overhead of various migration frequency

Table II shows a breakdown of the time overhead for each migration, which involves suspending and then resuming a connection. Connection suspension includes cleaning up the internal-external address mapping and destroying the virtual interface. To resume a connection, we need to update the mapping and recreate a new virtual interface. As we can see, the average connection restoration time is 30 ms for almost all tested applications except *ftp*. For *ftp*, the connection restoration time is 57 ms. This is mainly because an *ftp* session maintains both a control connection and a data connection. Virtual interface-related operations and the transfer control messages incur 12% and 18% overhead, respectively. The local processing related to updating and cleaning up the address mapping incurs 70% of the overhead. Because it requires invoking the userspace *iptables* program from the kernel space, which involves two expensive context switches.

We use *Netperf* [16] to measure our system's impacts on network latency and throughput and compare the performance under three different system configurations: *Vanilla*, *Vanilla+Virt* and *Migration*. *Vanilla* represents a stock Linux with *Netfilter* firewall rules loaded on boot. *Vanilla+Virt* represents the system with both *Netfilter* and migration module loaded, where the connections are not migrated but the socket system calls are intercepted. *Migration* represents the system with both *Netfilter* and migration module loaded and all connections are migrated.

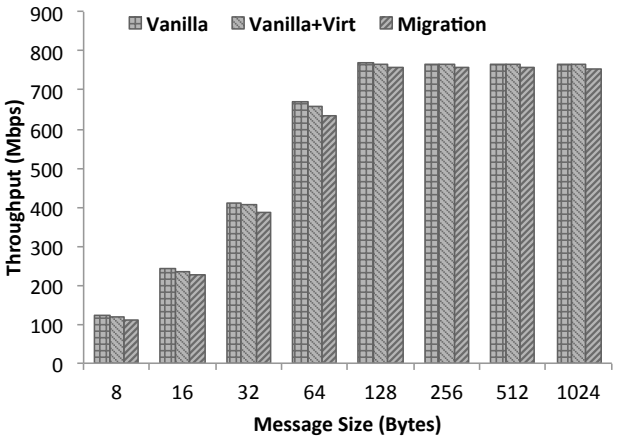

Fig. 4: Throughput overhead

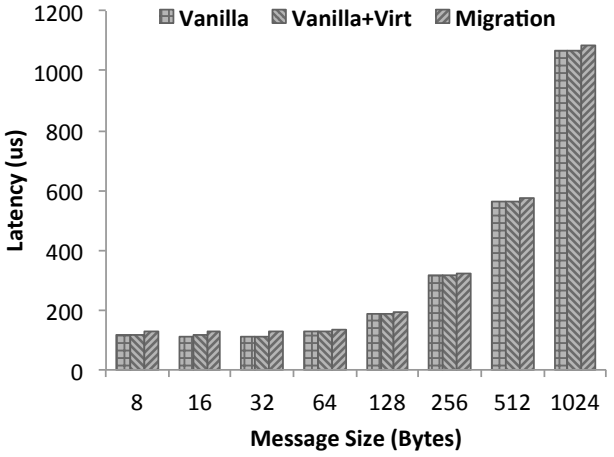

Fig. 5: Latency overhead

We run the *Netperf* client on one client VM and the *Netperf* server on the server VM. The throughput experiment uses the bulk data transfer mode TCP_STREAM and measures the achieved throughput when sending data as fast as possible from the server to the client. The latency experiment uses the request/response mode TCP_RR and measures the transaction rate in unit time. The message sizes range from 8 bytes to 1024 bytes and each measurement takes 60 seconds. Figure 4 and Figure 5 show the network throughput and latency for the three system configurations. Compared to the *Vanilla* system, there is almost no performance overhead for the *Vanilla+Virt* system; while the *Migration* system incurs 2% to 7% overhead for a single migration. Therefore, we can see that the socket system call interception incurs negligible performance overhead and the majority of the overhead is caused by the connection translation process.

TABLE II: Migration time breakdown

Application	Connection suspension (ms)			Connection restoration (ms)		
	Delete mapping	Destroy vif	Message communication	Update mapping	Recreate vif	Message communication
ssh	4.64	1.99	1.93	20.74	3.46	4.58
sftp	4.87	2.26	2.15	20.96	3.52	4.82
ftps	10.55	4.59	4.72	40.05	7.23	9.28
udpchat	3.91	2.28	1.82	19.65	3.39	3.86
tftp	4.38	1.59	2.02	20.91	3.66	4.07

B. System Scalability

We also study the scalability of DESIR when maintaining and migrating a large number of TCP connections simultaneously. Table III shows the connection handling time and the memory consumption. The average connection restoration time is 35 ms and remains constant. The memory overhead includes two parts: the virtual interface and the internal-external address mapping, both of which are proportional to the number of migrated connections. Furthermore, virtual interface accounts for over 90% of the memory consumption, each incurring 1.06 KB. As we can see, the memory consumption is acceptable (i.e. 5.2 MB) even when migrating 5000 connections.

C. Migration Frequency

We perform study on the maximal migration frequency for applications listed in Table I. For a connection with no constant data transfer, the highest migration frequency we can achieve is 30 ms per round. When there is ongoing data transfer, we can migrate 2 s per round. Because, to migrate a connection, we need to suspend the process owning the connection and reset the network interface IP which will temporarily suspend the transfer. The kernel network stack also needs a warm-up time before resuming the original data transfer. Moreover, the applications may perceive that the data transfer gets stalled too long time and reset the connection if we migrate too fast.

Our address randomization can be finished in 2 s, which is short enough to defeat most scanning attacks. For a class C network consisting of a real host and 253 Honeyd decoys, we use Nmap [10] to scan the network with the `insane` timing template, which by default is the fastest provided by Nmap. In this setting, Nmap waits 5 ms between scans and stops querying a node when the response time exceeds 250 ms. When performing ping sweep and port scan only, it takes on average 4.5 s to scan a host. If the service version detection and OS version detection are turned on, the average scanning time per host is 10.1 s. We also use ZMap [5] to perform a TCP SYN scan of the network. When scanning the lowest 1024 ports as Nmap does, Zmap can finish in 0.5 s. However, it only checks the aliveness of raw IP addresses without probing other information such as service version and OS version which is essential for the attacker to identify vulnerabilities.

VII. Related Work

Network address randomization aims at creating dynamic networks that change network properties including network protocols and addresses. Network address space randomization (NASR) implements IP address hopping based on DHCP updates to defend against hitlist worms [17]. DyNAT [18], [19] offers a protocol obfuscation approach that randomizes parts of network packet headers to mitigate scanning attacks. DynaBone [20] and Revere [21] introduce dynamic routing by creating a resilient overlay network on top of the Internet or multiple inner virtual overlay networks inside of a larger outer virtual overlay network, respectively. Corbett et al. introduces diversity in network protocol stack to counter jamming attacks [22]. Recently, software-defined networking (SDN) is being adopted to introduce dynamics into the network. Jafarian et al. use an OpenFlow random host mutation to protect a network from scanning attacks [4]. However, such NASR related techniques cause service disruption to active connections and thus hinder service availability to legitimate users.

A honeypot [23] can be viewed as a network decoy that is usually confined in a managed environment. Honeypots can be classified into two categories by their level of interaction with the adversaries: *low-interaction* and *high-interaction*. Low-interaction honeypots only emulate portions of a real host and offer minimal interaction with the attacker (e.g., honeyd [13], HoneyBOT [24], Dionaea [25]). High-interaction honeypots deploy real operating systems and applications with which the attacker interacts extensively (e.g., Honeynet [26], Sebek [27], Argos [28]). The recent deployment of honeypot framework mostly takes a hybrid approach involving both low-interaction and high-interaction honeypots [29], [30]. All these honeypot deployments aim to attract the attackers so as to learn about their attacking strategies; while we mainly use honeypots to entrap the attackers and prolong their scanning time.

Live VM migration refers to the process of migrating a running virtual machine (VM) or application from a physical machine to another without disconnecting the client. Many state-of-the-art VM managers have included support for live virtual machine migration, including Xen [31], VMware ESX [32] and KVM [33]. Typically they use a pre-copy approach to iteratively copy memory pages from source host to destination host. Moreover, network state transfer requires the source and destination hosts to be in the same subnet and the migration should be fast to prevent network connection timeout. In contrast, our migration does not require transmitting the disk and memory states. The source host and the destination host can also lie in different subnets. Furthermore, our solution allows much longer connection timeout during migration.

TABLE III: Connection migration overhead

Total number of connections	Total suspension time (s)	Suspension time per connection (ms)	Total restoration time (s)	Restoration time per connection(ms)	Memory consumption (KB)	
					Virtual interface	Address mapping
10	0.14	13.77	0.35	35.31	10.63	0.86
50	0.69	13.81	1.83	36.65	53.13	4.30
100	1.45	14.51	3.74	37.43	106.25	8.59
500	7.33	14.67	17.37	34.74	531.25	42.97
5000	73.5	14.7	174.1	34.82	5315	429.8

VIII. Conclusion

We propose a defense framework for constructing a dynamically mutable network with a number of decoys to protect the real servers against scanning attacks. Our solution can ensure seamless connection migration with IP address randomization. Moreover, with the deployment of a decoy bed, we can guarantee both service availability and service security of the real servers. We implement a VM-based prototype, which shows that our system has good scalability and flexibility, where the protected servers can be migrated with high frequency and multiple connections can be migrated simultaneously with acceptable network and system performance overhead.

IX. Acknowledgment

This work is partially supported by ONR grants N00014-15-1-2396, N00014-15-1-2012, ARO grant W911NF-12-1-0448, and a Cisco award.

References

[1] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Trans. Software Eng.*, vol. 37, no. 3, pp. 371–386, 2011.

[2] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, Eds., *Moving Target Defense - Creating Asymmetric Uncertainty for Cyber Threats*, ser. Advances in Information Security. Springer, 2011, vol. 54.

[3] M. Albanese, A. De Benedictis, S. Jajodia, and K. Sun, "A moving target defense mechanism for manets based on identity virtualization," in *Communications and Network Security (CNS), 2013 IEEE Conference on*, Oct 2013, pp. 278–286.

[4] 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*, ser. HotSDN '12, 2012, pp. 127–132.

[5] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast Internet-wide scanning and its security applications," in *Proceedings of the 22nd USENIX Security Symposium*, Aug. 2013.

[6] M. Abu Rajab, F. Monrose, and A. Terzis, "On the impact of dynamic addressing on malware propagation," in *Proceedings of the 4th ACM Workshop on Recurring Malcode*, ser. WORM '06, 2006, pp. 51–56.

[7] N. Provos and T. Holz, *Virtual Honeypots - From Botnet Tracking to Intrusion Detection*. Addison-Wesley, 2008.

[8] X. Fu, W. Yu, D. Cheng, X. Tan, K. Streff, and S. Graham, "On recognizing virtual honeypots and countermeasures," in *Dependable, Autonomic and Secure Computing, 2nd IEEE International Symposium on*, Sept 2006, pp. 211–218.

[9] 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*, June 2005, pp. 29–36.

[10] G. F. Lyon, *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, 2009.

[11] G. Su and J. Nieh, "Mobile communication with virtual network address translation," in *Technial Report CUCS-00302*. Department of Computer Science, Columbia University, February 2002.

[12] Linux Containers, https://linuxcontainers.org.

[13] N. Provos, "A virtual honeypot framework," in *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, 2004, pp. 1–14.

[14] S. K. Thompson, *Sampling, Third Edition*. Wiley, 2012.

[15] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 298–307.

[16] R. Jones *et al.*, "Netperf: a network performance benchmark," *Information Networks Division, Hewlett-Packard Company*, 1996.

[17] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis, "Defending against hitlist worms using network address space randomization," *Computer Networks*, vol. 51, no. 12, pp. 3471–3490, 2007.

[18] D. Kewley, R. Fink, J. Lowry, and M. Dean, "Dynamic approaches to thwart adversary intelligence gathering," in *DARPA Information Survivability Conference amp; Exposition II, 2001. DISCEX '01. Proceedings*, vol. 1, 2001, pp. 176–185 vol.1.

[19] J. Michalski, C. Price, E. Stanton, and E. Lee, "Network security mechanisms utilizing dynamic network address translation," 2002.

[20] J. D. Touch, G. G. Finn, Y.-S. Wang, and L. Eggert, "Dynabone: dynamic defense using multi-layer internet overlays," in *DARPA Information Survivability Conference and Exposition,*, vol. 2. IEEE Computer Society, 2003, pp. 271–271.

[21] J. Li, P. L. Reiher, and G. J. Popek, "Resilient self-organizing overlay networks for security update delivery," *Selected Areas in Communications, IEEE Journal on*, vol. 22, no. 1, pp. 189–202, 2004.

[22] C. Corbett, J. Uher, J. Cook, and A. Dalton, "Countering intelligent jamming with full protocol stack agility," *Security & Privacy, IEEE*, vol. 12, no. 2, pp. 44–50, 2014.

[23] L. Spitzner, "Honeypots: catching the insider threat," in *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, Dec 2003, pp. 170–179.

[24] HoneyBOT, http://www.atomicsoftwaresolutions.com.

[25] Dionaea, http://dionaea.carnivore.it.

[26] L. Spitzner, "The honeynet project: Trapping the hackers," *IEEE Security & Privacy*, vol. 1, no. 2, pp. 15–23, 2003.

[27] Sebek, https://projects.honeynet.org/sebek/.

[28] Argos, http://www.few.vu.nl/argos/.

[29] R. Berthier, "Advanced honeypot architecture for network threats quantification," in *PhD Dissertation*. Department of Computer Science, University of Maryland, 2009.

[30] T. K. Lengyel, J. Neumann, S. Maresca, and A. Kiayias, "Towards hybrid honeynets via virtual machine introspection and cloning," in *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, 2013, pp. 164–177.

[31] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.

[32] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05, 2005, pp. 25–25.

[33] U. Lublin and A. Liguori, "KVM Live Migration," http://www.linux-kvm.org/wiki/images/5/5a/KvmForum2007$Kvm_Live_Migration_Forum_2007.pdf.