# DESIR: Decoy-Enhanced Seamless IP Randomization

Jianhua Sun, Kun Sun
Department of Computer Science
College of William and Mary

# Roadmap

- Introduction
- Threat Model
- System Architecture
- Implementation
- Security Analysis
- Performance Evaluation
- Conclusion

# Roadmap

- Introduction
- Threat Model
- System Architecture
- Implementation
- Security Analysis
- Performance Evaluation
- Conclusion

# Background

- Network reconnaissance attacks have been effective due to the **static** nature of current network and system configurations
- Existing IP randomization based solutions shift network attack surface, including:
    - IP and MAC addresses, open ports, network topology

# Limitations of previous approaches

- Effectiveness of IP randomization is reduced due to the small number of alive IP addresses at one time
  - Small security entropy
- Existing active connections may be disrupted when the IP addresses of the servers are changed
  - Negative impact on user experience

# Contribution Highlights

Solve two major challenges!

- **Service Security** against malicious users
  - Fortify IP randomization with a large number of decoys that shuffle their address along with the real servers

- **Service Availability** to legitimate users
  - Develop a seamless network connection migration mechanism to keep alive the pre-existing connections

# Roadmap

- Introduction
- Threat Model
- System Architecture
- Implementation
- Security Analysis
- Performance Evaluation
- Conclusion

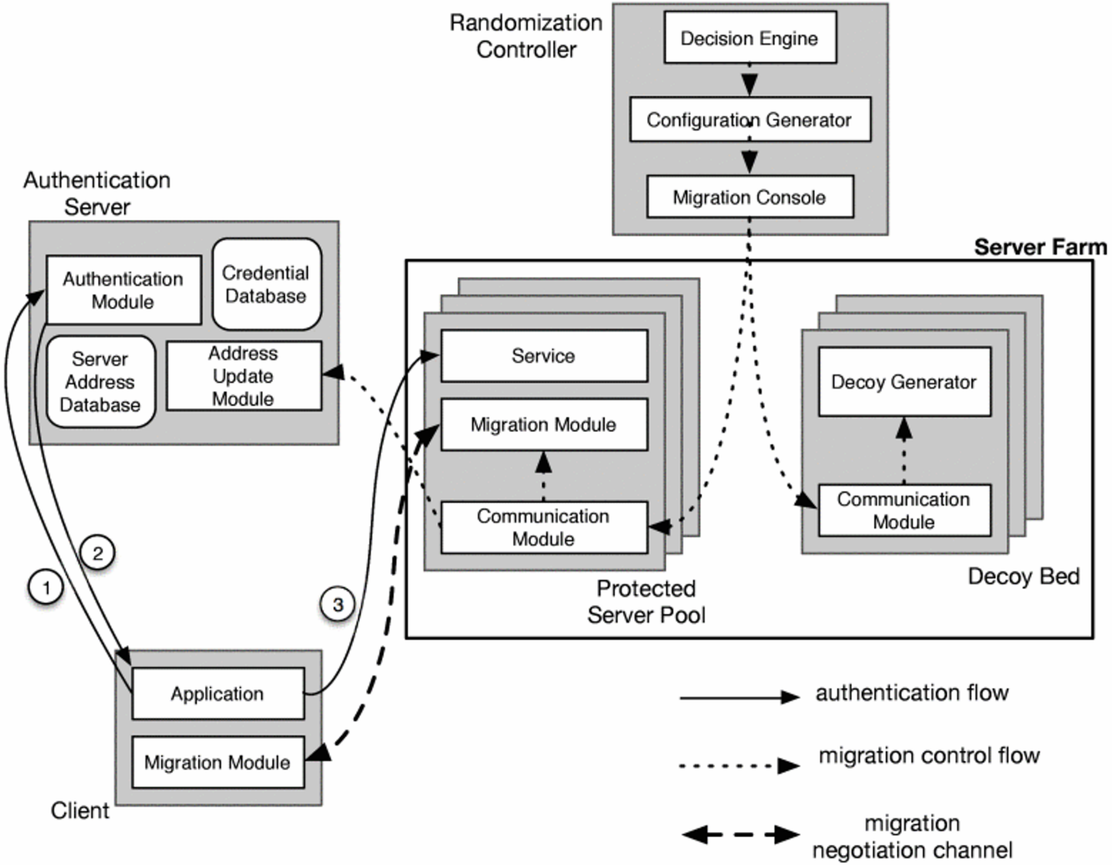
# Threat Model and Assumptions

- Focus on **persistent reconnaissance** attacks
  - Not consider insiders that deliberately disclose the current server IP address to attackers
- Adversaries are not in the same subnet with legitimate users
  - Cannot obtain the server IP addresses through packet eavesdropping
- Secret keys are shared between the legitimate users and the servers
- The protected network consists of a large number of IP addresses to accommodate decoy nodes

# Roadmap

- Introduction
- Threat Model
- System Architecture
- Implementation
- Security Analysis
- Performance Evaluation
- Conclusion

# System Architecture



# Randomization Controller

- Decision engine
    - Determine randomization frequency, choose algorithm to generate new network configurations

- Configuration generator
    - Control overall topology and regenerate new configuration
    - Guarantee there is no interference in IP address assignment

- Migration console
    - Distribute the new configuration to the servers and decoy subsystem

# Decoy Bed

- Communication module
  - Receive new configuration settings from the randomization controller
  - Determines the overall architecture of the decoy network

- Decoy generator
  - Regenerates the decoy network
  - Flexible to deploy both high-interaction and low-interaction decoys

# Migration Module

- Connection interception
  - Introduce a pair of internal and external addresses to detach transport layer identify from network layer identity

- Connection translation
  - Intercepts packets in the network layer and translates the internal addresses in the packet headers to or from the external addresses for outgoing/incoming packets

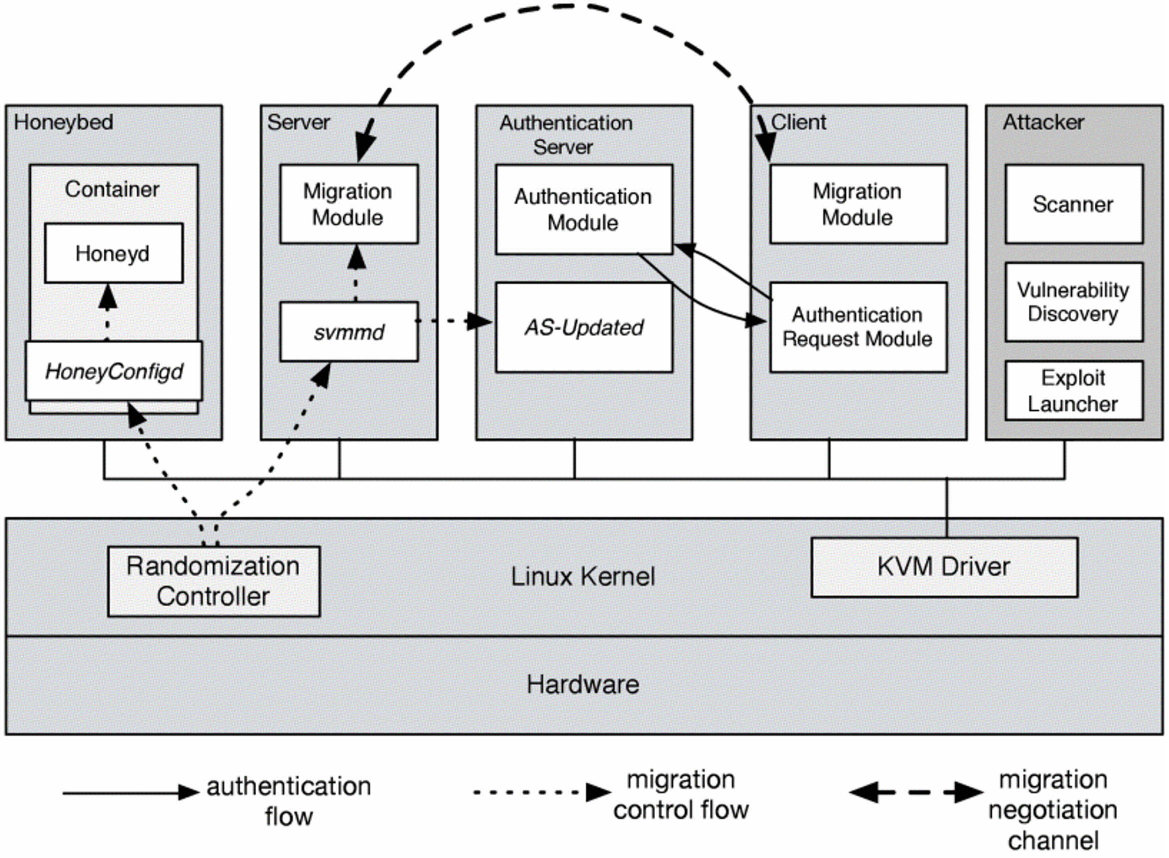
- Connection migration
  - Coordinates the moving of server associated with active connections to another IP address

# Roadmap

- Introduction
- Threat Model
- System Architecture
- Implementation
- Security Analysis
- Performance Evaluation
- Conclusion

# VM-based implementation

- **Host configuration**
  - Ubuntu 14.04 and KVM
  - Intel Core i7-4712HQ CPU, 16GB RAM
- **Five VMs:** decoy bed, real server, AS, client, attacker
  - Each VM allocated one host CPU and 2GB memory
  - Decoy VM runs Ubuntu 12.04
  - Remaining VMs run Fedora 15 with Linux kernel 2.6.38



# Three-level decoy bed

- Virtual machine level
  - Virtual machines with fully functional OS and applications
- Operating system level
  - Containers deployed using LXC in Honeybed VM
- Process level
  - Honeyd deployed in containers

# Seamless connection migration

- Connection interception
  - Intercept system calls for connection setup from the application layer to transport layer
  - For TCP connection, `socket`, `accept`, `connect`, `close`, `getsockname`, `getpeername`
  - For UDP connection, `send/recv`
- Connection translation - Use `iptables` to do NAT
  - Client side: DNAT on OUTPUT chain, SNAT on POSTROUTING chain
  - Server side: DNAT on PREROUTING chain, SNAT on INPUT chain
  - Use `mangle` table to block connection attempt to internal addresses

# Seamless connection migration

- Connection migration
  - Use two daemons within both endpoints to negotiate with each other the migration based on a predefined protocol
    1. Suspend the connection
    2. Restore after IP randomization is finished; create a VIF to which the internal connection is attached
  - Synchronously randomize the communication ports
  - Encrypt the negotiation messages with a shared secret key

# Roadmap

- Introduction
- Threat Model
- System Architecture
- Implementation
- Security Analysis
- Performance Evaluation
- Conclusion

# Scanning unaware of IP randomization defense

- Static IP address layout, no need to scan a single IP twice
  - **Sampling without replacement** problem
  - Expected number of probes to identify real server in a `n` IP pool: `(n+1)/2`
- Randomize the IP space after each probe. If the attacker takes a single-round scan, the expected number of probes is `(1-1/e)n = 0.63n`
- Attacker needs to pay 26% more efforts to locate the real server

# Scanning aware of IP randomization defense

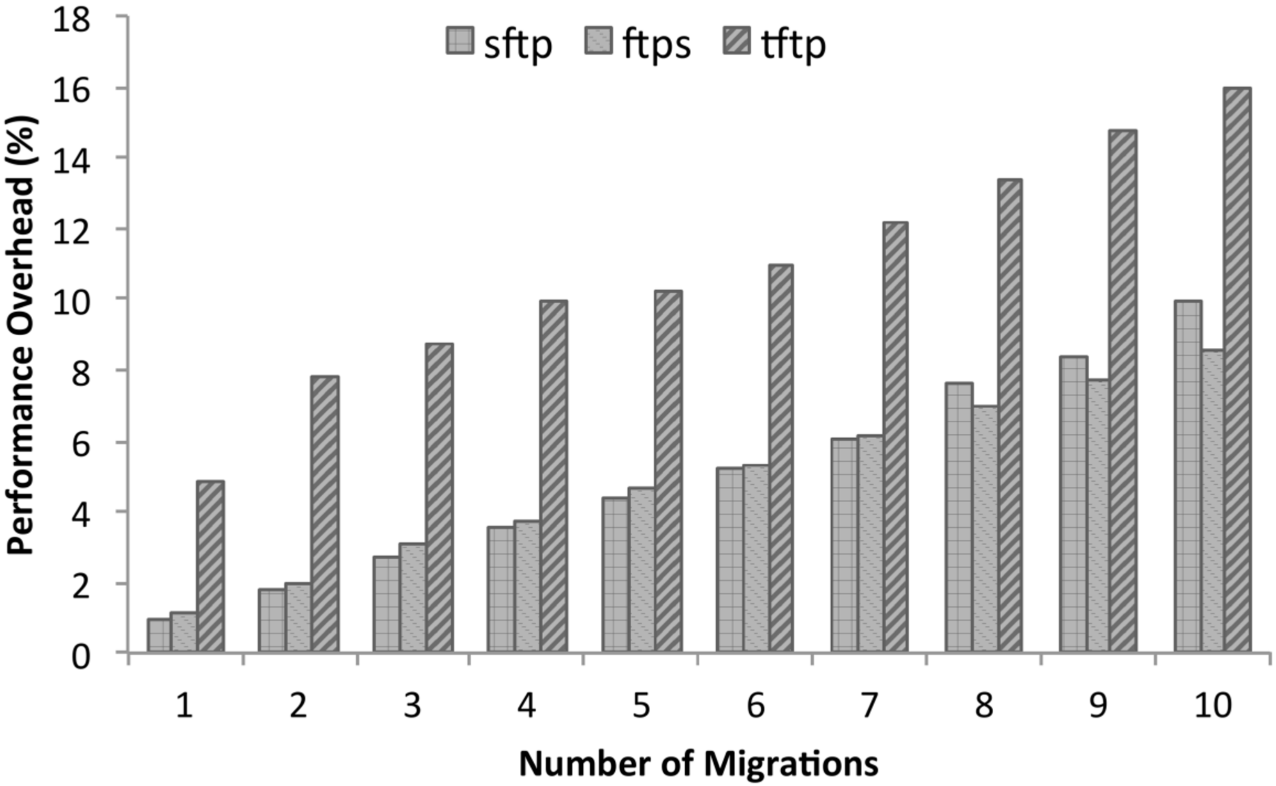
- Identifying the target server IP can be treated as a sampling with replacement problem
  - No matter whether the IP space is periodically re-randomized or not
- The number of probes `m` performed is a geometric random variable with probability `p=1/n`
- Therefore, the expected number of probes is `1/p=n`

# Roadmap

- Introduction
- Threat Model
- System Architecture
- Implementation
- Security Analysis
- Performance Evaluation
- Conclusion

# Microbenchmark - delay overhead

- Use `sftp`, `ftps`, `tftp` to transfer 1GB file
- `sftp`/`ftps`: 1% - 9% overhead
- `tftp`: 5% - 15% overhead

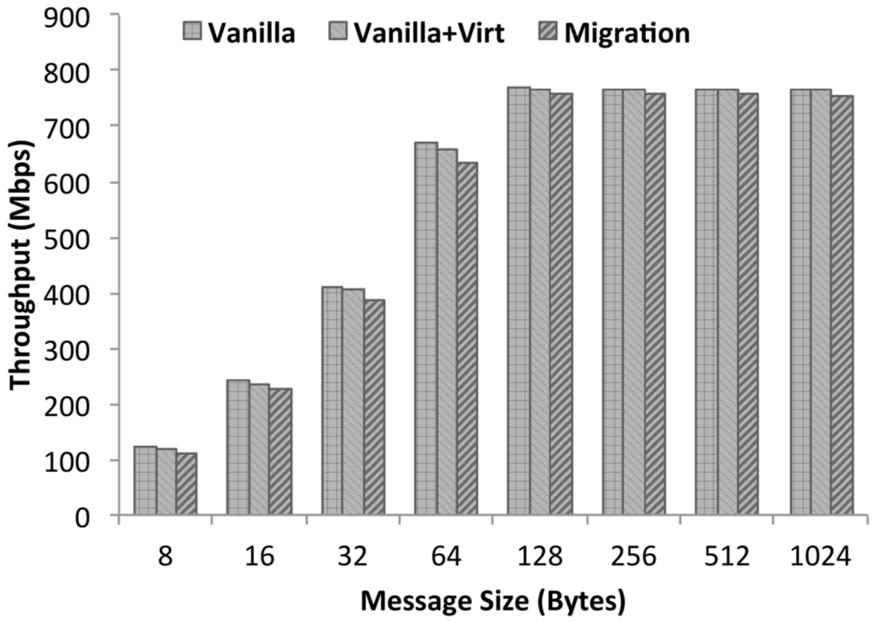


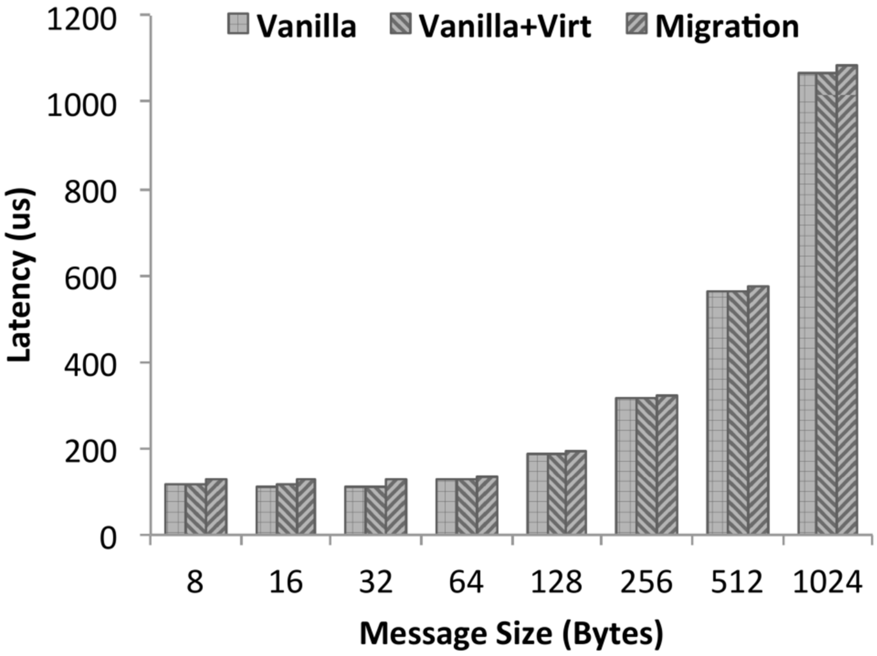
# System Overhead

- Use Netperf to measure migration related overhead in terms of network latency and throughput
- Three configurations
  - **Vanilla** - a stock Linux with Netfilter firewall rules loaded on boot
  - **Vanilla+Virt** - system with both Netfilter and migration module loaded, where the connections are not migrated but the socket system calls are intercepted
  - **Migration** - with both Netfilter and migration module loaded and all connections are migrated

# System overhead

- **Vanilla+Virt** - connection interception incurs no overhead
- **Migration** - 2% to 7% overhead - incurred by connection translation



# System Scalability

Connection migration overhead breakdown

| 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 |

- Average time to suspend a connection: `14 ms`; to restore: `35 ms`
- Virtual interface accounts for 90% of memory consumption, `1.06 KB` each
- `5 MB` memory overhead when migrating 5000 connections

# Roadmap

- Introduction
- Threat Model
- System Architecture
- Implementation
- Security Analysis
- Performance Evaluation
- Conclusion

# 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 and 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 acceptable network and system performance overhead