CloudRand: Building Heterogeneous and Moving-target Network Interfaces

Seungwon Shin
KAIST
claude@kaist.ac.kr

Zhaoyan Xu
StackRox Inc.
z@stackrox.com

Yeonkeun Kim
KAIST
yeonk@kaist.ac.kr

Guofei Gu
Texas A&M University
guofei@cse.tamu.edu

Abstract*—Some fundamental reasons why our networked systems are still vulnerable to network attacks are because (1) they are more open than necessary; (2) they are *homogeneous*, i.e., the same way to exploit a vulnerability on one machine is easily applicable to many other machines (which is particularly a severe issue in cloud computing environments when virtual machines images are heavily reused/cloned); (3) current networked services are merely *static targets*, i.e., they are easily predictable and do not change. While network authentication and access control mechanisms such as firewall and VPN can help reduce the openness (mostly at network perimeter level), they do not help much on the latter two factors. To bridge the gap and greatly complement existing network authentication/access control mechanisms, we propose CloudRand, a new framework to make networked systems/services in the cloud *heterogeneous* (every host has a different networking interface) and *moving targets* (such interfaces keep changing and they are unpredictable to untrusted entities). Inspired by the previous work on host-level (memory or instruction) Address Space Randomization (ASR), we build a lightweight solution to randomize network service interfaces. Thus, even derived from the same image, each virtual machine can have very different network service interfaces and they keep changing to further reduce the attack surface. CloudRand is an application-independent security service, orthogonal to existing application/network security mechanisms such as authentication, encryption, and access control. To fit into different environments such as clouds or enterprise networks, we provide various prototype systems at different levels for flexible deployment choices, e.g., host level (kernel drivers for both Linux and Windows), network level (based on Click modular router or software-defined networking technology), virtual machine hypervisor level (based on Xen), and application level (browser plugin). Our extensive evaluation shows that this solution has low overhead, and it can it can significantly reduce the network attack surface and successfully defeat malware epidemic attacks.

I. Introduction

Our computers and networks on the Internet are still susceptible to all kinds of cyber attacks. As a result, our society has significant economic loss due to these attacks. Earlier in 2011, it is officially estimated by the British government [16] that cybercrime costs the United Kingdom more than 27 billion pounds (i.e., 43.5 billion dollars) a year. It is clear that the global loss is significantly much higher than that. Recently, more severe and persistent worries also come from the new revolutionary computing platform, cloud computing environment. According to a recent Gartner report[1], the global cloud computing market will reach 411 billion by 2020. Despite its rapid development, security and privacy issues remain the major obstacles to cloud computing adoption [3], [13]. Moreover, concerning the usability, current cloud environments [13] may allow close communication with the bare metal instead of providing only software services. Thus, users can control nearly the entire software stack. Meanwhile, due to the wide use of virtual machine image migration and clone, there may exist a large number of virtual machines with similar operating systems and application configurations and as well as similar exploitable vulnerabilities. They both give fertile soil to the growth of malware epidemic attacks and once one guest machine inside of the cloud is comprised, it could bring catastrophic effect to the whole infrastructure.

Some fundamental reasons why our networked systems are still vulnerable to cyber attacks are as follows: (i) openness, which means that hosts are probably too open (in terms of the network surface) than necessary to such unwanted traffic; (ii) homogeneity, which means that hosts are too homogeneous so that the same exploitation successfully installed on one machine can be easily applied to another hosts (this is particularly a severe issue in cloud computing as discussed before); and (iii) static target, which means that the network interfaces of hosts (e.g., service ports) are static and predictable which makes the hosts vulnerable to attacks that successfully guess that right exploitation on the right service.

A lot of research and development has been conducted on reducing the openness of hosts/networks, e.g., existing network authentication and access control mechanisms (such as firewalls or distributed firewalls [8], VPN, router ACL). While they are quite effective to reduce openness (mostly at the network perimeter level), they barely help much on the latter two issues, i.e., homogeneity and static target. As a result, they are not enough to reduce the cyber attack surface. For example, while firewalls/VPNs can reduce external unauthorized attacks, they can hardly deal with internal attacks/propagations. Mechanisms such as router ACL that mostly use IP addresses to allow/block remote access are not flexible to handle users using DHCP (IP addresses will change), NAT (many users will share the same IP address), or even users using multiple machines (at different locations, or during travel, etc). Most of these techniques still allow access to common service ports such as 80 (web), 22 (SSH), which are still static targets exposed to attacks. In addition, Microsoft Windows will open many default service ports, e.g., 135, 137-139, 445, for NetBIOS (while these ports can be blocked from external network access, they are typically open to

[1]https://www.gartner.com/newsroom/id/3815165

internal network access). All these open service ports and the underlining homogeneous applications represent a significant attack surface exposed to potential risks. While they can rely on application level protection (e.g., application authentication protocols, application communication encryption), these are likely application specific and may not easy to apply to other applications without code modification, which makes them unsuitable as a generic application-agnostic network level solution. In addition, application-level protection may not stop exploitation attacks such as brute force SSH login attacks, not even mention infamous DoS attacks. We provide more details on related work and their weakness/differences in Section II.

In this paper, we propose *CloudRand*, a new lightweight and incrementally deployable framework, to provide secure networked systems/services for cloud computing environment or even enterprise network.[2] The key insight is to build heterogeneous and moving-target network port interfaces thus to decrease (and simultaneously shift) the existing attack surface to adversaries while still providing dependable service to system users and owners. More specifically, inspired by the previous research of system-level instruction set or memory layout randomization [14], [7], [17], we extend the idea into a wider network scenario and dynamically randomize the network interfaces, i.e., network port numbers. In doing so, we make our network port interface heterogeneous (every host has a different one). It appears chaotic to attackers (shifting all the time), and thus it forces the adversaries to significantly increase their work effort for every desired target.

In short, this paper makes the following contributions:

First, we propose to build heterogeneous and moving-target network port interfaces to significantly reduce the attack surface. Compared with existing network authentication and access control mechanisms that mainly reduce the openness, our solution is complementary and orthogonal, focusing on reducing homogeneity and providing moving-target defense. It is lightweight, incrementally deployable, and a nice add-on to cloud defense in depth.

Second, we design and implement a prototype system, CloudRand. As a cloud security service, it does not need to change or reconfigure service applications, which makes existing cloud services easy to adopt this *additional* layer of protection (defense in depth), in addition to their existing authentication or access control mechanisms. We make novel use of hash chain techniques to provide flexible randomization/translation and easily control the expiration of the service without the need of blacklist or revocation lists. Furthermore, we introduce process-binding techniques to reduce the probability that attackers misuse the provided application transparency to propagate from trusted nodes in the cloud. Finally, to fit into different environments such as clouds or enterprise networks, we provide various implementations at different levels, e.g., host level (kernel drivers for both Linux and Windows), network level (based on Click modular router or SDN technology), virtual machine hypervisor level (based on Xen), and application level (browser plugin).

Third, we extensively evaluate our prototype system in terms of effectiveness, efficiency, and flexibility. We show that CloudRand can significantly reduce network attack surface and successfully defeat malware penetration/epidemic attacks. In addition, CloudRand has a very low runtime overhead, e.g., it is more than six times faster than using SSL encryption in network communications in our evaluation.

For the rest of the paper, we first introduce related work and clarify their difference from our work in Section §II. We detail our design of CloudRand in Section §III and the implementation in Section §IV. We provide extensive evaluation in Section §V and discuss limitations and other issues in Section §VI. We conclude our work and point out future directions in Section §VII.

II. Related Work

System Randomization Techniques: Our work fits into the large body of research that applies automated diversity transformation to software/system to increase the difficulty for an attacker to exploit a security vulnerability. Several well-known randomization techniques at host side have been proposed, including Address Space Randomization (ASR) [14], [9], [30], system call randomization [10], instruction set randomization [7], [17]. For example, the basic idea of ASR is to introduce artificial diversity by randomizing the memory location of certain system components to defeat code injection attacks such as buffer overflow. The effectiveness and weaknesses of these techniques are well studied [22], [24], [12]. Recently, n-variant systems [12] are proposed. This n-variant framework executes a set of automatically diversified variants on the same inputs, and monitors their behavior to detect divergences. By constructing variants with disjoint exploitation sets, it is very hard to carry out large classes of important attacks. Different from all these work, our CloudRand technique is a network-level solution. Unlike those host-level techniques that might cause systems/applications crash on an unsuccessful attack attempt when an actual exploit is input into the system, CloudRand can prevent an unsuccessful attack to send the actual exploit. Compared to the above techniques, our CloudRand solution is more lightweight and preventive.

Similar to ASR, Network Address Space Randomization (NASR) is also proposed [6] with the goal to limit or slow down (but not prevent) hitlist worm propagation. The idea is basically to change Internet-wide IP address frequently so that the hitlist information used by worms will be stale shortly. This technique is only targeting to slow down (but not prevent) hitlist worm and has several practical limitations (e.g., it requires Internet-wide coordination of IP addresses) that make it hard to be actually deployed on Internet. Our CloudRand technique has a broader applicable scope (not just for defeating hitlist worm) and is much more practical for deployment on local networks or Internet than NASR.

Enterprise Network Protection Techniques (for inside threats): Firewalls and Network Intrusion Detection System

[2]We focus on cloud environment for the rest paper. However we note the techniques can also be easily deployed in enterprise networks as discussed in section IV.

(NIDS) are good approaches to protect internal hosts from outside attacks, but they have a limitation of not being able to detect attacks from the inside. To overcome this limitation, distributed firewalls were proposed to protect hosts from attacks both of inside and outside [8] [28] [31]. Even though they provide an ability to block some suspicious internal traffic, they still do not block well-known network ports such as port 80 for Web service. Essentially, they did not make internal malware propagation much harder. The same way that successful infiltration can still succeed on other internal hosts due to the homogeneity and static firewall configuration at each host. And this makes possible for inside malware, to spread over the well-known ports. With CloudRand, malware can not spread itself because CloudRand randomizes every network ports for service, even if it is not detected.

Network Authorization and Access Control Techniques: There are several standard techniques used for network authorization and access control, such as VPN (Virtual Private Network) and Router ACL. CloudRand is fundamentally different from them. First, existing network access control solutions are mainly designed to prevent unauthorized clients based on their IP addresses. That is, it is hard for them to prevent threats from hosts which are dynamically changing their IP addresses. And it is very common in current network environments (e.g., dynamic IP address and relocating VMs in a cloud computing). As a contrast, the ultimate goal of CloudRand is to reduce attack surfaces by randomizing network interfaces in server side. Thus, our CloudRand solution can reduce threats even if they are from dynamically changing IP addresses. Second, CloudRand uses lightweight randomization techniques such as hash functions, which have lower overhead than full cryptographic authentication protocols and packet payload encryption/decryption mechanisms. Finally it is worth noting that CloudRand is not intended to replace them but to complement them because their protection focuses are orthogonal.

Port-knocking [19] and Single Packet Authentication (SPA) [11] are two proposed techniques for protecting hosts from network scanning and they use multiple packets (or single encrypted packet) to identify a client whether it is benign or not. There are significant differences between them and CloudRand: (1) In Port-Knocking, the sequence of contacting network ports is static. Therefore if an attacker knows this sequence, he can use this information for future attacks. However, because CloudRand changes network port dynamically, it is nearly impossible for an attacker to guess future open ports. (2) Port-knocking suffers from packet out-of-order delivery and it frequently happens in current networks. But CloudRand does not have this problem. (3) Fundamentally, the goal and protection granularity are different. The granularity of Port-knocking and SPA is at host level and primarily used to authenticate a specific client/host. However, CloudRand is primarily used to provide a fast-flux networking interface for the server and the protection granularity is at per-application or per-service level.

Port Randomization/Translation Techniques: A similar study to our CloudRand technique is presented in [27], in which a port hiding technique is proposed to defend Web applications against DoS attacks. The authors suggest hiding a server port number and allowing a legitimate client to use the hidden server port number as an authenticator to access the Web application. Our work is different from this study in several ways: (1) The goal is different. Their approach can be only used to protect Web applications from DoS attacks, while our solution can be used to protect any applications in hosts/cloud to reduce attack surface with this moving-target defense. (2) The randomization technique is different. We apply a novel use of reverse hash chain to design port randomization for temporal clients with automatic expiration without even changing the key/seed, however their approach requires periodically update keys. (3) They use Javascript redirection technique at client side to translate Web requests and forward to the right server port. This cannot be applicable to services other than Web application. Instead, we provide more general and comprehensive translation techniques that can be applicable to all applications.

III. System Design

A. System Overview and Illustration

CloudRand is a security service that can be provided at hypervisor level to hosts (virtual machines, VM) in the cloud. If a VM wants to use the service to randomize its network interface (in terms of service ports), it first registers the service with CloudRand and lets its network (client) users aware of the use of this protection. From then on, the service ports on the VM will be dynamically/periodically randomized. To avoid modification/reconfiguration to the service software in the VM, CloudRand does not require the server application to change its actual listening port every time. Instead, CloudRand (in the hypervisor) will perform on-the-fly port translation/rewriting and redirect wanted traffic go through. For instance, at day 1,[3] one HTTP/Web service is announced at a random port 12345, i.e., any connection to this service should go through destination port 12345 only. A legitimate client (e.g., a VM in the same cloud, or a partner cloud, or just a remote user) is aware of this randomized port number (more precisely the randomization algorithm). Thus, it will communicate with the server through 12345. The application on VM does not change its listening port but allows the hypervisor to rewrite/redirect all the traffic from port 12345 to port 80 on-the-fly. Other unwanted traffic to the wrong port (e.g., 80) will be ignored (and the sender can be blocked after several failed attempts). In the next period (e.g., day 2), the service port number is changed from 12345 to another random number, e.g., 48205. Again, the legitimate client knows about this change and can still smoothly access the service. To be flexible, CloudRand allows the service to provide two kind of accesses to users: long-term or short-term. In the later case, the capability of learning the right port to access will automatically expire.

[3]For convenience, in this scenario, we assume that CloudRand changes network ports daily.

An illustrative working scenario is shown in Figure 1, in which CloudRand is deployed in *our cloud*. With the CloudRand protection to significantly reduce the attack surface, attack trials from Internet (i.e., T1-1) and *partner cloud* (i.e., T2-1) can not infect VMs in our cloud. However, those unprotected VMs in *partner cloud* could be infected by attack trails from Internet (i.e., T1-2) and/or neighbor VM (i.e., T2-2).

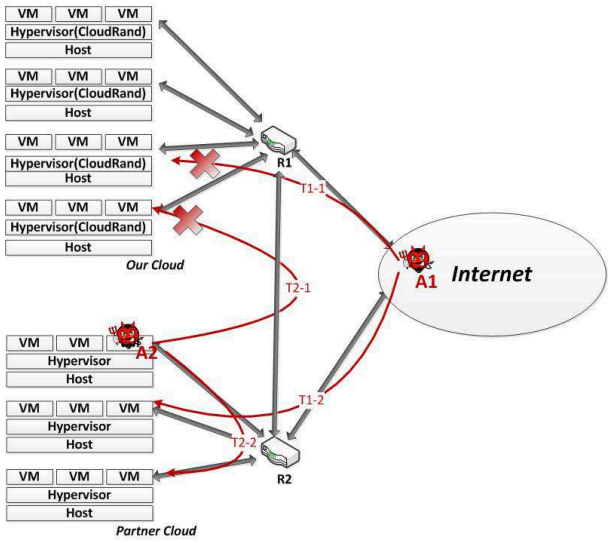

Fig. 1. Illustrative Scenario of CloudRand

B. Randomization Algorithm Design

Depending on the actual need, CloudRand allows a server to provide network port interface randomization protection to two kinds of client users: those trustworthy ones with long-term service or those less trusted with short-term service.

To provide a client with long-term randomization service, the port randomization algorithm works as follows (shown in Algorithm 1)

Input: S (random value for seed)
Input: P (original network port)
Input: T_s (starting date)
Input: T_n (current date)
Input: $d = T_n$ - T_s
Output: H^d (hash chain value)

while $d > 0$ **do**
  $H \leftarrow H\ (S,\ P)$;
  P = H;
  $d = d - 1$;
end
H^d = H;

Algorithm 1: Randomization Algorithm

In many cases, it is necessary to give some client right to contact CloudRand protected servers for a limited time. For example, when it is hard to decide whether specific clients are fully trusted or not, it is better to give a temporal access which will expire after a desired times of use. To achieve this goal, we reverse the use of hash chains stated earlier. And we maintain another CloudRand translation policy entry for temporary access purpose. We pre-calculated a long chain of hash value, $H(y)$ ... $H^n(y)$ (here y is some initial random number which will not be shared with clients), where n is a relatively large number (set by the administrator according to the maximum expiration time for a client). Instead of using hash chain values from $H(y)$ to $H^n(y)$ as in the Cloud Passport case, we reversely use the chain from $H^n(y)$ (in day 1) to $H(y)$ (in day n). In order to allow an access that expires in e days, assume the current hash value is $H^m(y)$, we provide the client $H^{m-e+1}(y)$.

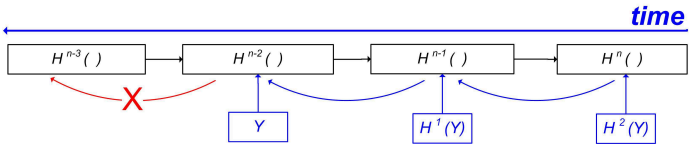

Fig. 2. Using reverse hash chain in port randomization

Figure 2 shows an example with $e = 3$, and the client obtains $Y = H^{m-2}(y)$. For the first day, the client uses $H^2(Y) = H^m(y)$ to calculate the port number.[4] On day 2, the client uses $H^1(Y) = H^{m-1}(y)$. Similarly, on day 3, the port number is calculated using $Y = H^{m-2}(y)$. If three days have been passed, the client will fail to access because it cannot obtain previous hash value $H^{m-3}(y)$ (it only knows H^{m-2}). Since a hash value is not reversible, our CloudRand granted access is guaranteed to automatically expire after a desired time of use.

In principle, one can use any existing hash algorithms such as MD5. In our current implementation, we use a fast-speed and specialized-in-integer (because network port number is integer value) hash function [29]. Additionally, it is important to prevent randomized network port values from conflicting with each other if possible, although it is very rare and probably never happen in practice. To avoid port collisions on a single service, we provide the option of using $H^d(seed, original\ port) \oplus OriginalPort$ as the destination port for a client to use, similar to [27].

It is worth noting that the CloudRand protected server needs to make its client users aware of the randomization algorithm (either long-term or short-term). We do not consider this as a practical limitation. In reality, a server that needs protection from CloudRand typically has that capability and can coordinate with its users using any existing approach/protocol (e.g., as part of the service agreement, establishing a very simple web service for the purpose), or through their already existing communication channels, or through out-of-band channels such as email, web, phone, short message.

C. On-the-fly CloudRand Translation

One technique challenge comes from the requirement of server application *transparency*. The binding port for server

[4]Port number is calculated by original port $\oplus$ current hash value

side software is usually statically specified. Some notorious service ports are fixed and not easy to change, e.g., Windows NetBIOS sharing. Even for those configurable services, many regular users may not know how to change default ports.

We propose to provide our application-transparent CloudRand service at the hypervisor level inside the cloud. Specifically, we use Xen[4] as our hypervisor platform and control Dom0 to execute port translation and traffic protection for other DomU. For instance, at the server side, once one registered service application starts, CloudRand performs corresponding randomization algorithm to determine current port number and adds one new rule to its CloudRand service table for this VM, such as $port\ 12345 \rightarrow port\ 80$. If a packet targets at the randomized port 12345 to the destination VM, CloudRand redirects and rewrite the packet to the port 80 on-the-fly. Any other traffic, including the traffic directs to port 80, may be simply discarded or further monitored (because they are suspicious).

D. In-Cloud Process Binding

Previously we have discussed the service application transparency. Similarly, we may also provide client application transparency. In this section, we first assume the communicating client is also in the cloud, either the same cloud or a collaborative cloud, and we leave the discussion of a remote client to next section. Clearly, the in-cloud client side can use the same translation mechanism as long as CloudRand service is granted. Thus, client application can work even without knowing the randomization algorithm.

However, there is a subtle issue we can discuss here. While providing great convenience, arguably the client application transparency can be misused if the client is compromised (e.g., by malware). Thus, the malware program inside the client can also access the right port of the server (because CloudRand at client side does blind translation). To reduce such risk, for client-side CloudRand, we provide process-binding translation, e.g., the port translation is binded to the corresponding certain legitimate program. To realize automatic process-binding, one can start with providing a list of commonly used legitimate client-side programs that will be used to contact CloudRand protected service. It is not surprising that for specific protected service, the number of legitimate client software is typically very limited and easily enumerable. For example, if the web service on port 80 is protected by CloudRand, programs such as `IE`,`Firefox`, `Google Chrome`, can be considered as legitimate and automatically binded for port translation. Meanwhile, users can always customize the list and add new programs as they want.

Implementing such function is not straightforward at hypervisor level due to the lack of semantic information of the OS [25], [26]. Even with introspection tools such as XenAccess [20] so that we can get access to raw memory of each guest system, the task of extracting the kernel data structure from the mapped memory to bind packets with its source program is still challenging. In CloudRand, we implement our own techniques to extract fine-grained process-to-port binding information, similar to [25], [26].

Specifically, in Linux, the open socket information is expressed as files owned by each process. Thus, by examining the kernel exported symbols stored in the `System.map`, we first search and extract virtual address of all the related kernel symbols, including `inet_hashinfo` for network service and linked list of `task_struct` for each process. For the structure `inet_hashinfo`, it maintains all local socket binding information. We further search the linked list of sub-structure `inet_bind_hashbucket` and enumerates all the nodes inside of `inet_bind_bucket`. In this way, we obtain the information of the port and the binded socket. On the other side, XenAccess provides basic introspection function to allow us introspect all the process information. From the base address of structure `task_struct`, we traverse the its sub-structure, `files_struct`, to find all the sock files used by each process. If any of them matches the previous extracted socket file, we can correlate the port with the certain process.

For windows, we begin with examining the loaded modules and find the location of the network driver `tcpip.sys`. In the kernel region, we locate the `KdVersionBlock` (fix `0xffdff034` offset for XP) and derive the address of `PsLoadedModules`. After iterating the module lists and getting the pointer of `tcpip.sys`, we further find the data structure `AddrObjTable` for TCP, `TCBTable` for UDP, that maintains the linked list of objects containing network ports and process IDs for open sockets/connections.

IV. Extending CloudRand from Hypervisor to Multiple Layers

We have implemented CloudRand at hypervisor level on top of the open-source cloud computing environment Xen [4]. We take advantage of existing *IPTables*[2] in Xen Dom0 to monitor, redirect, and filter out unwanted traffic.

It is worth noting that the idea behind hypervisor-level CloudRand can be easily extended to other environments/layers. To maximize the flexibility of real-world deployment, we extend CloudRand translators onto different levels such as application level, host (kernel) level, and network (router/switch) level. Table I summarizes the design space, pros/cons, and examples of our current CloudRand implementations.

When applying CloudRand in real world clouds/networks, we need to install our components at network, hypervisor, application, and/or OS level. Since each cloud computing environment may be different from each other, it could be hard to predict which level is the best selection to minimize the modification to the systems. Thus, we provide multiple level solutions of CloudRand for the cloud/network administrator to flexibly choose the most suitable solution based on their actual environments and needs.

A. Network Level Translator

Our current implementation based on Click modular [18] router is shown in Figure 3. We implemented a new *PortRandom* element in C++ to the Click modular router. This element

Approach	*Layer*	*Pros*	*Cons*	*Our example implementation*
CloudRand-aware program	application	easy to start using	application/user be aware of the CloudRand algorithm	simple client-side CloudRand translator/software
application plug-in	application	simple	support only specialized user programs	FireFox extension
kernel device driver	kernel	no change to applications, cover all applications on host	require installing kernel driver	Linxu/Windows kernel CloudRand driver
hypervisor modifications	hypervisor	no change to existing program/os, cover all inside VMs	only effective when using virtual machines	Xen-based CloudRand
router/switch upgrade	network	no change to existing program/os	no internal communication protection	Click modular router
software-defined network	network	no change to existing program/os/hypervisor, cover all network components	require flow rule installation overhead	SDN controller application

TABLE I
DESIGN SPACE OF CLOUDRAND

has two stages to perform port randomization and translation. In the initial stage, it obtains policy data from a Cloud Manager and registers to a mapping table. In the service stage, the PortRandom element will (i) accept incoming TCP or UDP that comes through IPClassifier element which is a built-in element of Click; (ii) confirm whether destination IP/port of incoming network packet is registered in the mapping table or not. If so, it will translate and rewrite this port to the original port. If not, it checks if this packet is targeting to original port. If it is, it reports this to CloudRand manager, otherwise it discards the packet. In other cases (not for the CloudRand service), it will simply work as a regular network router.

We also implement a network-level CloudRand prototype as an Software-Defined Networking (SDN) application. With SDN decoupling the control plane from the data plane, the job of relaying packets to a random port can be migrated to a controller. Thus, as shown in Figure 4, we implement an SDN app that (i) monitor the appearance of a new flow (step 1-2), (ii) calculates a randomized port if its destination is a protected server (step 3), and then (iii) dynamically configures the switch so that it translates the port number of every following packets (step 4-6). To notice a new incoming flow and configure a switch, we use the OpenFlow protocol [5]. For example, we listen a `packet_in` message to handle an incoming flow and send a pair of `flow_mod` messages with `set` actions to modify the TCP/UDP port number of packets in the corresponding session. In our SDN extension evaluation, we also use Open vSwitch, which is a software switch supporting the OpenFlow protocol and mostly employed in cloud environments [21].

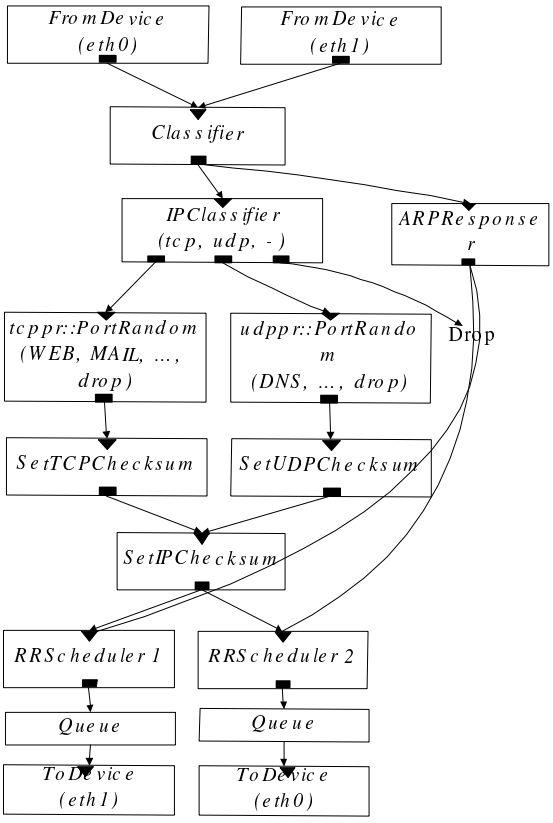

Fig. 3. Click configuration of the network-level CloudRand Translator.

B. Application Level Translator

For application-level translator, currently we implemented a Firefox browser plugin/tool-bar as shown in Figure 5. If a client tries to connect to (CloudRand protected) web servers, he can simply type server URLs in the CloudRand tool-bar. Then, it automatically translates the port to connect the web server.

C. Host Level Translator

This host-level CloudRand translator has two modules: a kernel monitoring module and a user-level policy module. For every incoming and outgoing packet, CloudRand translator will do the examination to identify the validity of each packet. The examination and translation are at the kernel level, which means every packet will go through our kernel monitoring module first. Meanwhile, the user-level component will maintain the trusted process list and interact with kernel-level

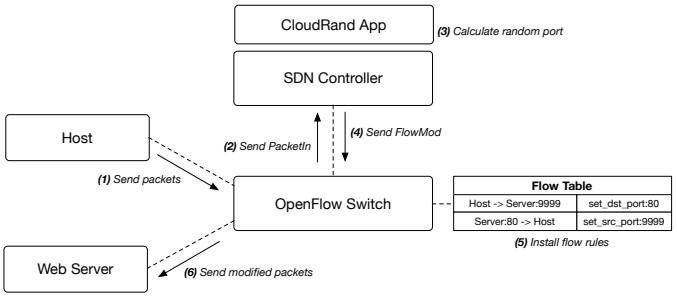

Fig. 4. SDN extension and workflow of CloudRand

Fig. 5. FireFox browser extension for CloudRand translator.

component to block illegal packets if necessary. Currently, our implementation can successfully randomize and translate the port number of TCP and UDP packets on Windows and Linux.

- *CloudRand translator for Windows*
  We develop a kernel-level filter-hook based on the Windows IP Filter Driver, `IpFltDrv.sys`, which exists in both Windows XP and Windows Vista. We manipulate (randomize/translate/rewrite) the packet fields such as *Dest Port Number* and *Checksum*. Another data structure,`ProcessGuard`, specifies the processes which are allowed to use client-side port translation.
- *CloudRand translator for Linux*
  The implementation on Linux is built on top of the well-known firewall, `IPTables`. Through loading the kernel module `IptableNat`, we can easily redirect the packet to the translated destination port using `Redirect` option. We note that this is essentially the similar use of NAT/PAT function supported by `IPTables`. That is, existing NAT/PAT capable device can be slightly modified to support CloudRand port translation function. Through another `IPTables` kernel module `IptOwner`, we can specify `pid` option to allow only trusted specific process to use client-side Port Translation function. User-level policy module is similar to Windows implementation, which dynamically correlates the process information to current running `PID`.

V. Evaluation

The CloudRand framework provides an effective and lightweight solution to greatly reduce the attack surface of our networked systems. In this section, we conduct an evaluation to demonstrate its efficiency and effectiveness. Section V-A evaluates CloudRand's ability to defeat attacks such as malware propagations. Section V-B deliberately measures the overhead impact of the CloudRand on hypervisor and other extended scenarios (e.g., host/network level).

A. Effectiveness Evaluation

CloudRand system can effectively defeat malicious epidemic attacks from outside or inside networks. To evaluate its effectiveness, we build a network with the topology shown in Figure 6(a). We set up the test in a safe virtual environment and run Agobot [1](to connect to our controlled command and control server). The botmaster tries to command the Agobot to attack two Windows machines, Target A (without protection) and Target B (with CloudRand protection). Both Target A and B have *DCOM* vulnerability which could be successfully exploited by Agobot. Since Target A does not employ CloudRand protection, it can be easily exploited by Agobot, as shown in Figure 6(c). On the contrary, since Target B has CloudRand service to randomize the vulnerable port 135 and 445, it is not infected, as shown in Figure 6(b).

Next, to evaluate the effectiveness of CloudRand to stop, or at least slow down, malware propagation in a relatively large-scale network, we perform a packet-level simulation using GTNETS network simulator [23]. We use a tree network topology with a total number of hosts at 2,400. The 2,400 hosts divide into 24 subnets and each subnet could be considered as a virtual cloud. Their IPs range from 192.168.1.1 to 192.168.24.100. We simulate a UDP worm with scanning rate of 50 packets per second and keep running each simulation for 10 minutes. At the same time, the simulated UDP worm only targets at 10 possibly vulnerable ports (such as 33,37,42,53,80,123,137,138,139, and 194) to perform its exploitation. In the experiment, we start from one worm-infected machine in the whole network and let it infect the rest. We vary the protection coverage of CloudRand to show the different results, as shown in Figure 7. We can clearly see that when all networks deploy CloudRand, the worm can hardly propagate at all. With higher ratio deployment of CloudRand, the number of final infected machines in simulation decreases significantly. This clearly confirms that CloudRand is an effective protection scheme for malware epidemic attacks. And since it is incrementally deployable on a network, each subnet can benefit from the technique.

In addition to defeating malware propagation, CloudRand system can prevent specific ports from being blindly scanned and thus efficiently filter large volume of unwanted traffic while still providing legitimate clients access. To demonstrate this, we perform an examination of how well a trusted client's request can be handled in the situation when large volume of unwanted attacks (e.g., scan attempts) are occurring. For unwanted traffic generation, we simply generated large number

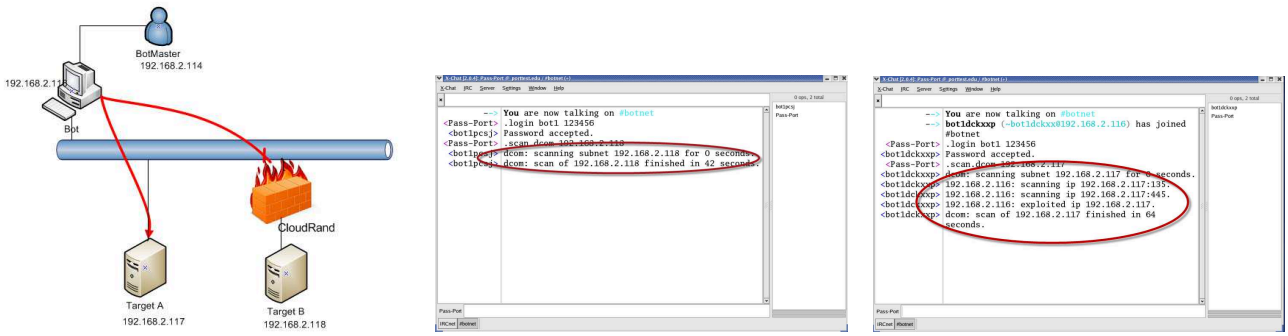

Fig. 6. Effectiveness of CloudRand to defeat malware propagation attacks.

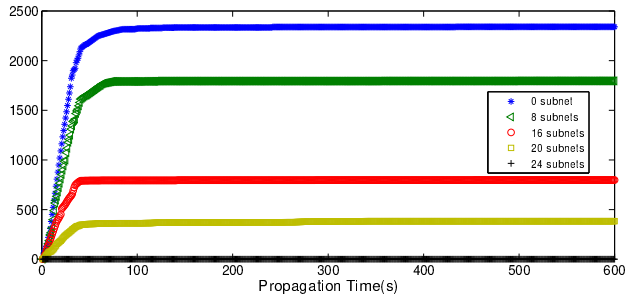

Fig. 7. Worm propagation under different CloudRand protection ratio.

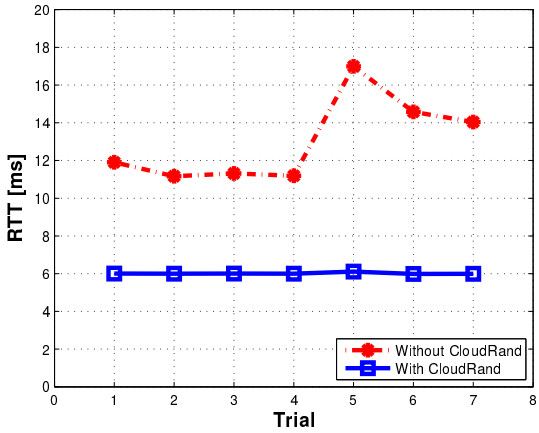

Fig. 8. Effectiveness measurement of CloudRand in the presence of large unwanted traffic volume.

of TCP SYN requests to invalid ports, similar to a SYN flooding attacks. In particular, we send around 6,000 packets per second for these unwanted traffic in this test. Figure 8 shows the RTT (round-trip time) performance comparison when network-level CloudRand protection (deployed in a Click software router connecting an attack machine and a normal Apache web server) is turned on and off. We perform several trials, and in each trial we measure 1,000 times of RTT and then calculate the mean.

From the figure we can see that in the presence of large volume of unwanted scanning attempts, we can achieve roughly more than 2 times faster RTT when the CloudRand protection is enabled. This is because in this case the router can efficiently filter unwanted traffic and let the legitimate packets go through. In the case of no CloudRand protection, legitimate clients have to compete with large number of unwanted traffic at the router.

This test clearly demonstrates the effectiveness of CloudRand to defeat unwanted network penetrations and provide better service for legitimate clients.

B. Overhead Evaluation

In this section, we comprehensively measure the overhead impact of our CloudRand system. All the experiments were performed with two machines. The first machine is used as a server providing CloudRand service and the specification of this machine is Intel Core Duo Processor at 2.93 GHz with 3GB RAM. Our implementations of on host level, hypervisor level, and network level are realized in this machine. On host level, each DomU guest ran Windows XP SP2 or Redhat Linux Enterprise system. On Hypervisor level, Dom0 ran a Centos 5 linux system.

1) Overhead Evaluation on Hypervisor: In our hypervisor level implementation, deploying CloudRand framework may introduce extra overhead for each inside virtual machine. In this section, we conduct several evaluation to measure the overhead brought to each hypervisor/VM. The overhead mainly comes from three aspects: (1) CloudRand service registration; (2) Packet-level examination and redirection/rewriting; (3) Client-side process monitoring/introspection.

The basic overhead for each hypervisor machine contains the invocation/updating/termination of CloudRand service. In our measurement, CloudRand consumes around 0.3 second to register a new CloudRand-protected service at each hypervisor. The randomization is synchronized every 24 hours between hypervisors. The cost of updating one record is around 0.1 second. When user asks for disabling the CloudRand service, it spends around 0.01 second.

To measure the (translation/rewriting) overhead brought to each packet, we measure a metric of packet Round Trip

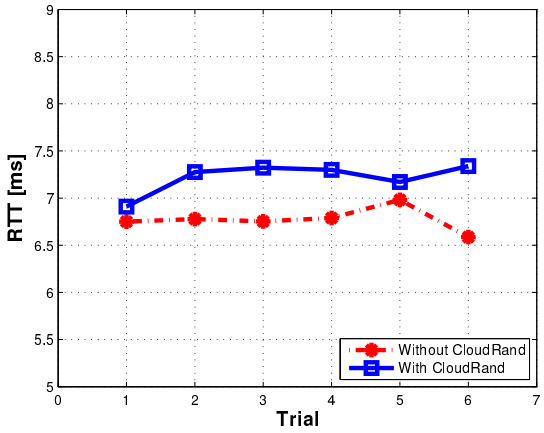

Fig. 9. Overhead measurement of CloudRand at Hypervisor

Time (RTT). In our test, we measure RTT for 6 trials and we use `wget` program [15] to send network packets[5]. We have captured all network packets between the test client and server(s) to estimate the RTT. More specifically, we measured RTT as the time difference between the first data packet sent by the wget client and the first response packet received from the server. For each trial, we sent $1,000$ packets to two virtual machines with/without CloudRand protection at the same time and calculate the mean time of all packets. The result is shown in Figure 9. As the RTT is concerned, the extra overhead brought is around 2.3% in the best case (around 10% in the worst case). It shows that our CloudRand is evidently a lightweight solution and the user can barely realize the existence of port translation.

Another overhead need to measure is the (in-cloud client-side) periodic process introspection time. We construct an environment with four domU virtual machines installed on one dom0 Xen hypervisor. Among them, two domU virtual machines are under protection by CloudRand. One Windows machine runs 19 processes and another Linux machine runs 26 processes. Meanwhile, 4 out of all processes are registered as CloudRand-protected processes.

We measure the average introspection time taken to allow registered process initiate a CloudRand protected connection (TCP or UDP). The consuming time for both TCP and UDP cases is recorded in Table II. As the result shows, the introspection overhead is affordable for mainstream cloud platforms to protect the service in real time. It demonstrates that our CloudRand can be easily deployed into existing Cloud computing environment with relatively low overhead.

2) Overhead Evaluation on Extended CloudRand: In this section, we measure the overhead of extended CloudRand implementations on both host (kernel driver) level and network level.

The basic overhead of host-level CloudRand is shown in

[5]Note that we use `wget` program to measure RTT in all the following test cases.

Guest OS	*# of Processes*	*TCP Time(ms)*	*UDP Time(ms)*
XP SP2	4/19	10.6	8.3
Ubuntu 9.04	4/26	9.7	9.1

TABLE II
INTROSPECTION OVERHEAD ON HYPERVISOR LEVEL

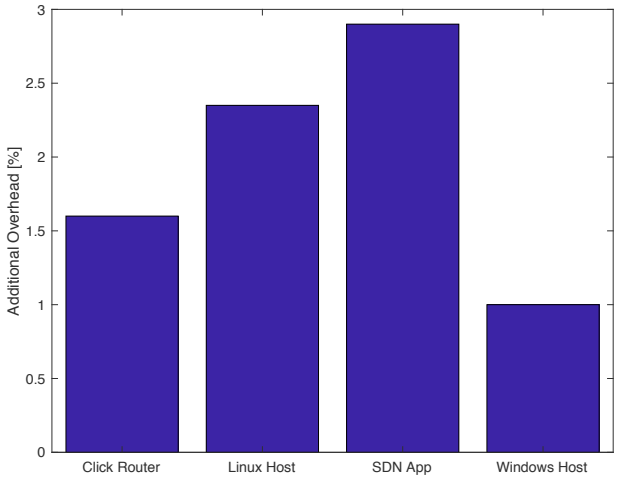

Fig. 10. Overhead of Packet Redirection/Rewriting using CloudRand

Table III. At the invocation phase, Windows system need to load CloudRand kernel driver and policy module. Linux system enables the `iptables` module and configures it according to CloudRand requirement. The termination simply stops all the CloudRand service.

	Invocation(s)	*Update/Policy(s)*	*Termination(s)*
Windows Host	1.63	2.43	1.03
Linux Host	0.013	0.012	0.008

TABLE III
BASIC DEPLOYMENT OVERHEAD OF EXTENDED CLOUDRAND TRANSLATOR

For each incoming/outgoing packet, the introduced overhead of redirection/rewriting are summarized in Figure 10. We measure RTT time with the mean of $1,000$ packets trials. For host-level measurement, we turn on our click module and let it act as normal router at the network perimeter. Accordingly, we enable our Windows/Linux kernel driver for each protected host. For network-level measurement, we construct two different test environments for evaluating both the click module and the SDN application versions of CloudRand. The result of network and host level measurement is compared with the case without CloudRand protection. In this case, we measure additional overhead, which is caused by CloudRand, and describe the overheads in terms of percentage.

As shown in Figure 10, it is evident that the overhead of CloudRand measured by RTT is extraordinarily small (less than 2.5% for host-level implementations and between 1.5% and 2.9% for network-level implementations). With such reasonably small overhead, the extended CloudRand can provide us with more flexible and complete deployment

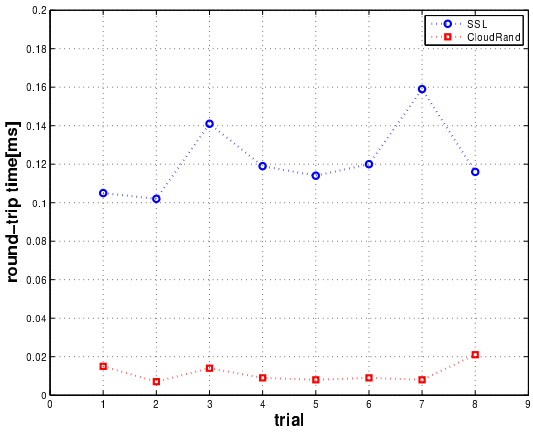

Fig. 11. Overhead measurement between CloudRand and SSL.

choices for various scenarios, such as enterprise network or even worldwide Internet. We conclude that our CloudRand is a lightweight, comprehensive, and deployable solution in real world.

3) Overhead Comparison between CloudRand and Encrypted Network Channels: Besides measuring the overhead of CloudRand itself, we are also interested in measuring its overhead compared with commonly used network encryption mechanisms such as Secure Sockets Layer (SSL). It is worth noting that CloudRand is never intended to replace these encryption mechanisms. Instead, it is quite possible that CloudRand is used to complement them to make a more secure cloud computing environment. In this situation, we need to measure the additional overhead added by CloudRand in the environment.

To investigate this, we compare the overhead of CloudRand with SSL, a widely used network encryption protocol. In our test environment, we enable the SSL module of Apache web server (with default parameters and no server certificate checking) and make `HTTPS` requests at the client side. We query one web page for $1,000$ times and calculate the average round-trip time of packets. As a comparison, we query the same web page with normal `HTTP` requests from the same client machine (with CloudRand enabled) and perform the port translation at the kernel layer of the CloudRand server.

Figure 11 shows the overhead of CloudRand and SSL. We can clearly see that the overhead of CloudRand is much smaller than that of SSL; SSL is more than 6 times slower than CloudRand in the experiment. This result implies that when we apply CloudRand to a cloud computing environment to complement network encryption mechanisms, CloudRand causes almost negligible additional overhead to the environment.

VI. Discussion

In this section, we review the issues that we do consider but not fully describe in previous Sections.

Synchronization issue. In order for correct port randomization and translation, we require the client and server have loose synchronization when applying the hash function. Since a typical time period is one day, this requires very loose synchronization which will not cause an issue for most hosts. In case of a failure because of a time lag, the legitimate client can also try the next day's hash value. In any case, hosts can easily use NTP (Network Time Protocol) to synchronize the time to solve the synchronization problem.

Session management at the change of port. CloudRand changes its port number periodically to prevent hackers from guessing current randomized network port. However, it is necessary to maintain old network connections, to maintain previously established connections (in previous time period). In this case, session management table which maintains established connections can solve this. Usually, these port changes do not occur so often since port number alterations occur on a daily basis or even longer.

Compatibility with some applications. Some applications such as FTP may have port information embedded in the application layer packets. Thus, if we only rewrite transport layer port numbers, there will be problems (not correctly rewrite application layer port information). This is not unique to CloudRand. Actually, our system has essentially the same problem faced by NAT devices. Most of NAT/PAT solutions already consider this issue and they can perform application aware rewriting of port information at application level (if they are in clear text). Our CloudRand can solve the problem in similar way.

CloudRand deployment on Internet. Although we discuss CloudRand mainly in the context of clouds (or enterprise networks), it is clear that it could be easily extended and incrementally deployed onto the whole Internet. Using CloudRand will definitely increase the diversity of the current Internet, significantly reduce the cyber attack surface, and dramatically decrease malware epidemic attacks.

Other Limitations. CloudRand is not designed to be a perfect security solution. It has limitations. For example, it does not prevent legitimate users to leak randomizing algorithms (keys) to others nor prevent attackers to obtain them from other approaches (e.g., network sniffing or social engineering), which are also common assumptions for most (if not all) cryptographic key related protection mechanisms.

Meanwhile, it cannot prevent the possible invasion caused by credential stolen through process injection or spawned process. Furthermore, CloudRand itself does not indicate which users should or should not obtain randomizing algorithms, leaving it as a policy level task for administrators. After all, CloudRand simply does what it does to reduce, instead of fully prevent, the attack surface with the moving-target idea. Since it is orthogonal to most existing security mechanisms, we believe CloudRand is a valuable add-on to the defense-in-depth strategy.

VII. Summary and Future Work

As a new element in defense in depth, we presented CloudRand, a lightweight framework and system to protect cloud computing environments by reducing the attack surface. We implemented CloudRand prototype systems as a comprehensive toolkit. Our extensive evaluation showed that our solution adds low overhead to both of the system and the network, and it can successfully defeat unwanted network attacks from both of inside and outside. Our solution is also incrementally deployable in clouds or on Internet, and each cloud/network can benefit from (and thus be motivated to deploy) the techniques.

For future work, we will study new coordinated, cross-layer system interface randomization techniques to defeat cyber attacks.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation (NSF) under Grant no. 0954096, 1642129, 1700544, and 1740791. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Agobot (computer worm). http://en.wikipedia.org/wiki/Agobot.

[2] Iptables. www.netfilter.org.

[3] Security is chief obstacle to cloud computing adoption,study says. http://www.darkreading.com.

[4] Xen. http://www.xen.org.

[5] The openflow switching consortium. http://www.openflowswitch.org/, 2009.

[6] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis. Defending against hitlist worms using network address space randomization. In *WORM '05: Proceedings of the 2005 ACM workshop on Rapid malcode*, pages 30–40, New York, NY, USA, 2005. ACM.

[7] E.G. Barrantes, D.H. Ackley, S. Forrest, T.S. Palmer, A. Stefanovic, and D.D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. of the 10th ACM Conference on Computer and Communications Security*, Oct 2003.

[8] Bellovin and M. Steven. Distributed firewalls. In *USENIX login*, 1999.

[9] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium*, Washington, DC, August 2003.

[10] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, Dec 2002.

[11] CipherDyne. fwknop: Single packet authorization and port knocking. *http://www.cipherdyne.org/fwknop/*.

[12] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th conference on USENIX Security Symposium (Security'06)*, Berkeley, CA, USA, 2006. USENIX Association.

[13] M. Armbrust et al. Above the clouds: A berleley view of cloud computing. Technical Report UCB/EECS-2009-28, University of California at Berleley, Feb 2009.

[14] S. Forrest, A. Somayaji, and D.H. Ackley. Building diverse computer systems. In *Proc. of the 6th IEEE Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[15] GNU. GNU wget. http://www.gnu.org/s/wget/.

[16] Michael Holden. (Reuters) Cyber crime costs 27 billion pounds a year. http://uk.reuters.com/article/2011/02/17/uk-britain-security-cyber-idUKTRE71G34H20110217.

[17] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM.

[18] Eddie Kohler, Rober Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. In *Proceedings of 17th Symposium on Operating System Principles (SOSP)*, pages 217–231, December 1999.

[19] Krzywinski M. Port knocking: Network authentication across closed ports. In *SysAdmin Magazine 12*, 2003.

[20] Bryan D. Payne, Martim Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Annual Computer Security Applications Conference (ACSAC'07)*, 2007.

[21] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, 2015. USENIX Association.

[22] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM.

[23] Georgia Tech Network Simulator. http://www.ece.gatech.edu/research/labs/.

[24] Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the feeb? the effectiveness of instruction set randomization. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 10–10, Berkeley, CA, USA, 2005. USENIX Association.

[25] A. Srivastava and J. Giffin. Tamper-resistant, applicationaware blocking of malicious network connections. In *Proceedings of 11th International Symposium On Recent Advances In Intrusion Detection (RAID)*, 2008.

[26] A. Srivastava and J. Giffin. Automatic Discovery of Parasitic Malware. In *Proceedings of 13th International Symposium On Recent Advances In Intrusion Detection (RAID)*, 2010.

[27] Mudhakar Srivatsa, Arun Iyengar, Jian Yin, and Ling Liu. A client-transparent approach to defend against denial of service attacks. In *SRDS '06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 61–70, Washington, DC, USA, 2006. IEEE Computer Society.

[28] J. Lane Thames, Randal Abler, and David Keeling. A distributed firewall and active response architecture providing preemptive protection. In *Proceedings of the ACM Southeast Conference*, 2008.

[29] Thomas Wang. Integer Hash Function. http://www.concentric.net/~Ttwang/tech/inthash.htm.

[30] J. Xu, Z. Kalbarczyk, and R.K. Iyer. Transparent runtime randomization for security. Technical Report UILU-ENG-03-2207, University of Illinois at Urbana-Champaign, May 2003.

[31] Cliff C. Zou, Don Towsley, and Weibo Gong. A firewall network system for worm defense in enterprise networks. In *Technical Report at University of Massachusetts ECE (TR-04-CSE-01)*, 2004.