TimeVM: A Framework for Online Intrusion Mitigation and Fast Recovery Using Multi-Time-Lag Traffic Replay

Khalid Elbadawi
School of Computing
DePaul University
Chicago, IL 60604
badawi@cs.depaul.edu

Ehab Al-Shaer
School of Computing
DePaul University
Chicago, IL 60604
ehab@cs.depaul.edu

ABSTRACT

Network intrusions become a signification threat to network servers and its availability. A simple intrusion can suspend the organization's network services and can lead to a financial disaster. In this paper, we propose a framework called TimeVM to mitigate, or even eliminate, the infection of a network intrusion on-line as fast as possible. The framework is based on the virtual machine technology and traffic-replay-based recovery. TimeVM gives the illusion of "time machine". TimeVM logs only the network traffic to a server and replays the logged traffic to multiple "shadow" virtual machines (Shadow VM) after different time delays (time lags). Consequently, each Shadow VM will represent the server at different time in history. When attack/infection is detected, TimeVM enables navigating through the traffic history (logs), picking uninfected Shadow VM, removing the attack traffic, and then fast-replaying the entire traffic history to this Shadow VM. As a result, a typical up-to-date uninfected version of the original system can be constructed.

The paper shows the implementation details for TimeVM. It also addresses many practical challenges related to how to configure and deploy TimeVM in a system in order to minimize the recovery time. We present analytical framework and extensive evaluation to validate our approach in different environments.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive software*; D.4.8 [**Operating System**]: Performance—*modeling and prediction*

General Terms

Design, security, virtual machine, traffic replay, performance

ASIACCS '09, March 10–12, 2009, Sydney, NSW, Australia.

Keywords

intrusion, mitigation, replay, recovery

1. INTRODUCTION

A simple intrusion can cause an organization to suspend its network activities and subsequently lead to a financial disaster. To increase system availability, many organizations deploy firewalls and intrusion detection systems (IDS) for the purpose of blocking or detecting attacks [13]. However, due to the existence of novel attacks, unknown vulnerabilities, or misconfiguration in firewall or IDS devices, the possibility of successful attack and hence system unavailability is significantly high. Therefore, a quick response to eliminate - or even mitigate - the attack's infection and recover the system is necessary for all critical services.

To achieve the above goal, log-replay technique can be used. In which, a snapshot of the current system state is captured at specific times, and the system events and states are recorded in a log file. A replay process uses the snapshot along with the information logged to replay the sequence of states/events a system passed through during execution. Once the replay process is done, the system is recovered.

Log-replay-based recovery can be utilized to improve the network service availability. Upon detecting the attack and discovering its sources, one can roll-back the system execution to an earlier point and extract the cause of infection. This technique has a great merit that it does not require a complete awareness of the attack, such as zero-day attack.

Several researchers have focused on log-replay-based recovery [11, 8, 15, 18, 21]. However, these works are limited because they were designed for debugging or software failure. Another problem with most of these techniques is that they require OS modifications or a specific hardware platform in order to log all system events in kernel [6] and process levels [10, 7, 9]. This is not only complex and expensive, but it also requires very large log files.

Log-replay-based recovery is usually implemented in virtual machines. Virtualization has been used extensively in the area of fault-tolerance, security, and system recovery. This is because virtualization has several advantages that make it better suited for providing reliability and security [3, 20]. One advantage is that virtualization provides a strong isolation between virtual machines. If an operating system has been compromised by an attack, the whole processes are then compromised. However, with virtualization, if a virtual machine (VM) has been compromised, other VMs that run other applications will continue to run without any serious

threat. Another advantage is that it is much easier to manipulate the state of a virtual machine than the state of a physical machine. The state of the virtual machine can be saved, cloned, encrypted, migrated, and restored, none of which is easy to do with physical machines.

In this paper, we propose an approach for log-replay-based recovery called TimeVM. TimeVM consists of a set of identical virtual machines. Also, rather than logging system stats and events, TimeVM logs only the network traffic. The traffic logged is sent (as a replay) to these virtual machines in different time window. This means that these machines represent multiple historical snapshots of the original virtual machine at different times in the past. If the original machine has been contaminated, there is at least one virtual machine in the history that is not contaminated yet. The steps of system recovery are performed as follows. (i) Rolling back by selecting a clean virtual machine, (ii) extracting the infected traffic, and (iii) rolling forward by replaying the entire traffic to the selected virtual machine. As a result, a typical up-to-date uninfected version of the original system has been constructed.

There are several challenges in designing and configuring TimeVM. The first challenge is how to replay the network traffic successfully. The second challenge is how to determine the suitable number of virtual machines we need and how to space them such that the recovery time is minimum. We formulate this issue as an optimization problem and we use an available evolutionary algorithm to solve it. TimeVM can be deployed for many network servers such as HTTP web server, SMTP server, Telnet Server, etc.

The rest of the paper is organized as follows. Section 2 summarizes some of the previous works on log-replay-based recovery along with some other techniques for improving the availability of network services. Section 3 gives a high level overview of TimeVM. The analytical model and formulation of recovery time optimization problem is presented in section 4. In section 5, we give a detailed view on TimeVM implementation. In section 6, we evaluate our framework using different configuration. Finally, we conclude in section 7.

2. RELATED WORK

In literature, there are several techniques in software failure and attack mitigation in order to provide high system availability. We first discuss those techniques that are related on log-replay-based recovery and then we discuss other techniques that improve the availability of network services.

Instant Replay [11] is a general replay for parallel programs. It logs the relative order of significant events as they occur without logging the data associated with such events. In the same track, Flashback [18] is a lightweight operating system that provides fine-grained replay capability at the application level. It uses shadow processes to roll back in-memory state of a process at specific execution point, and log a process' interactions with system to support deterministic replay. The purpose of Flashback is to help debug software. Both techniques are limited on the application level.

Flight Data Recorder (FDR) [21] is an offline full-system recovery that replays the last one second of execution before a crash. FDR continuously logs all the inputs coming into the system, such as I/O, interrupts, DMA transfers. This approach is not suitable for long-run system, such as network servers.

BugNet [15] focuses on replaying only user programs and shared libraries to find application level bugs. It logs the register file contents at any point in time, and the load values that occur after that point. This allows BugNet to collect enough information to perform deterministic of program's execution, without having to replay what goes on during interrupts and system calls. The disadvantage of BugNet is that it is limited on application level.

Revirt [8] is a logging and replay system that runs on a VM. If an attack is detected, Revirt can replay the whole system for analyzing the intrusion. Revirt is implemented in a specified virtual machine called UMLinux.

ExecRecorder [7] is a full-system replay for post-attack analysis and recovery. It has the capability to replay the execution of an entire system by checkpointing the system state and logging architectural nondeterministic events. However, ExecRecorder is implemented only in a uniprocessor system. Moreover, the log file size varies from one system to another. In our approach, the log file depends on the traffic load. TimeVM does not log the payload of its packet. It only logs the payload of client packets. This makes our log file very small compared to other techniques.

Dunlap et al. published a new system called SMP-Revirt [9]. SMP-Revirt is also a full system log and replay on multiprocessor virtual machine that is capable to run on commodity hardware. It utilizes hardware page protections to detect races between virtual CPUs. Although SMP-Revirt system can recover the entire system, it requires a huge amount of storage space. As they reported, 300GB disk can be filled in few days.

TCP or connection migration is another dimension to provide high availability for network services. Snoeren et al. [17] uses TCP migration options to record the TCP state for connection resumption. Their approach inserts a HTTP-aware module between the application and the transport layers to log the inter-layer interactions. The disadvantage of this approach is that it requires a modification to the TCP implementations. Migratory TCP [19] and Ray et al. [16] allow online connections to be migrated from one server to another server. When a server is under an attack, the migration process is triggered, which causes the client to reconnect to a replica server. The drawback of these two techniques is that it requires an extension protocol in both sides.

3. TIMEVM ARCHITECTURE

3.1 System Overview

In this section, we give a high level view of our proposed framework architecture. Before describing TimeVM components, we first need to describe the idea behind it. TimeVM is based on virtual machine technology and log-replay-based recovery. It can be used to provide high availability for network services, such as HTTP, FTP, SMTP, etc. TimeVM requires that the network services should be installed on a virtual machine. This virtual machine will be cloned into several virtual machines. Moreover, TimeVM records all traffic between the original VM and clients, and keeps it in a special file called *traffic log*. The recorded traffic will be sent by several replay processes to the cloned virtual machines in different times. In other words, each cloned VM receives the same traffic that was received by the original VM but after a predefined time period called *time lag*.

Now, If an attack occurred to the original virtual machine and it has been detected and discovered after a time say T,

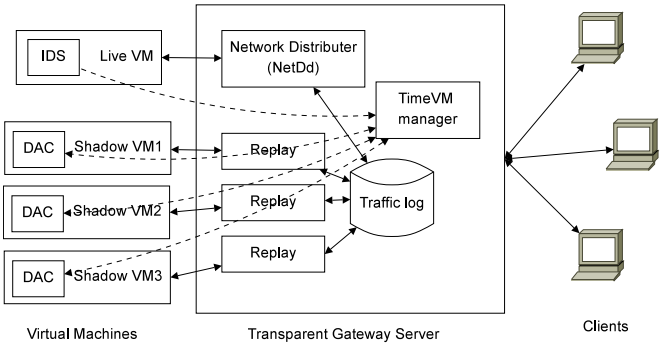

Figure 1: Framework Architecture

then TimeVM will perform the following actions. It triggers all replay processes that are configured with a time lag greater than T to accelerate the process of packet replaying. It invokes a cleanup process to remove all virtual machines that have a time lag less than T. When all recorded traffic have been replayed, TimeVM reconfigures the system such that the nearest cloned VM becomes the original VM, the next nearest cloned VM becomes the first cloned VM, and so forth. Finally, TimeVM creates new virtual machines (as a duplication of the new original VM) to substitute the removed ones and configures them with an appropriate time lag. It is worth to mention that the information about the source of the attack (such as IP address and/or port number) will be provided to the replay processes so as to prevent sending the infected traffic back again to the cloned VM's.

Figure 1 illustrates the main components of TimeVM. The framework is composed of a gateway server and several virtual machines. One virtual machine will be used to represent the original VM. We will call this virtual machine *the Live VM*. Other virtual machines that are a duplication of the Live VM are called *the Shadow VMs*. Note that all these virtual machines - the Live VM and the Shadow VMs - are hidden from the outside world. Clients cannot connect to the Live VM directly; instead they can only connect to the gateway server. In other words, when a client sends a request to the gateway server, the server in turn forwards the request to the Live VM. When the server receives back the response of the Live VM, it will deliver it to that particular client. The client will see that the request has been processed by the gateway server and not by the Live VM.

The gateway server is a Linux server and it runs three processes: TimeVM Manager, Network Distributor Daemon (NetDd) and Replay process. TimeVM Manager is responsible to manage and control NetDd and Replay processes. It is also responsible to communicate with each virtual machine in the system. NetDd is responsible to log and redirect the traffic between the Live VM and clients. It is acting as a transparent NAT server. Replay process is responsible to *send/replay* the logged traffic to a Shadow VM. There are several replay processes and each one is associated with a single Shadow VM. The implementation details of these processes will be given in section 5.

NetDd and Replay processes share a common file, which is the traffic log. The format of the log file is quite similar to the libpcap file format in the sense that each packet has two records: the packet header and the packet data as shown in Figure 2. The packet header consists of the following fields:

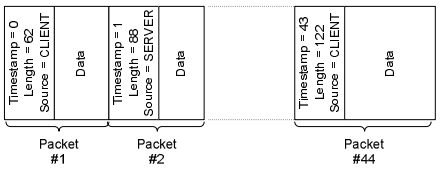

Figure 2: Log file format

packet timestamp, packet length, and packet source. Packet timestamp records the time when this packet is captured, and it plays a very important role when packets are replayed. Packet length represents the total length of the packet data. Packet source indicates whether this packet comes from a client or from the Live VM. We need this field because it facilitates the work of the replay process. From Figure 2, `SERVER` means that the source IP address is belonging to the Live VM, and `CLIENT` means that the source IP address is belonging to a client machine.

The content of packet data depends on the packet source. If the packet comes from a client then the packet data will consist of the following information: IP header, protocol header (such as TCP, ICMP), and payload. However, if the packet comes from the Live VM, then the packet data will consist only of IP header and protocol header. The information in IP and protocol headers is sufficient for replay process to replay the server traffic. One advantage of not logging the server responses is that it reduces the traffic log dramatically.

3.2 Identifying Attacks

Every virtual machine is set up with a host-based IDS along with DACODA [6]. We use a host-based IDS, such as Minos [5], to raise an alert when an infection has occurred. Note that the IDS is running in the LiveVM only. We use DACODA to perform further investigation for each single byte received by a network interface. DACODA has the ability to generate a new signature for zero-day worms. It also has the ability to generate a report such as where the infection comes from. Therefore, our system relies on DACODA in discovering and identifying the bogus traffic received by the LiveVM. The only limitation of DACODA is performance. Therefore, we must limit its running time as explained next.

At the time that the IDS generates an alert, TimeVM invokes DACODA at every Shadow VM's and then waits for a response. Please notice that receiving a response from a Shadow VM (let say Shadow VM number k) means that all Shodow VM's from 1 up to k are already infected, and all Shadow VM's from $k+1$ up to N, where N is the total number of Shadow VM's, are not infected yet. Therefore, upon receiving a response, TimeVM suspends DACODA for the cleaned Shadow VM's, and then it triggers the replay processes that are belonging to these cleaned ones to switch to fast replay mode. The information obtained from DACODA will be provided to these replay processes in order to prevent infected traffic from being replayed back.

4. OPTIMIZING RECOVERY TIME

The major goal of our framework is to provide fast mitigation and recovery when an infection is identified by using an IDS along with DACODA. Infection identification cannot

happen instantly. Many attacks may involve more than one packet or more than one network session that it makes difficult for an IDS to detect them at the early stage. Moreover, DACODA requires an amount of time to report where the infection comes from. As a result, it is potential that several Shadow VM's might be infected either.

To optimize the recovery time, we need to configure the time lag for each replay process and the number of Shadow VM's in the system properly. These two values are highly dependable on the time required by the IDS and DACODA to detect and discover an infection. For example, if the infection can be detected in a very short period of time then it is reasonable to have only two Shadow VM's (because the first Shadow VM will be used to discover the infection and the second one will be used to replace the Live VM) with small values of time lag. If the detection time is random, then we need to set up a number of Shadow VM's. Therefore, we need to know how many Shadow machines are needed and how to space them in time such that the average recovery time is minimum.

Before representing our analytical results, we need to define some notations. Let λ be the mean packet arrival rate at the system. Let λ_a be the mean arrival rate at a Shadow VM when its replay process is running in fast mode. Let TTD (Time To Discover) be the time required by an IDS and DACODA to discover an infection. Let T_{fm} be the time taken by a replay process to be in fast mode. Let R be the time required for the whole system to recover. Notice that R is a sum of two quantities of time: (1) the time required by IDS subsystem to discover an infection, and (2) the time required by a selected replay process to finish packets' replaying, *i.e.*, $R = TTD + T_{fm}$.

4.1 Optimizing R for two Shadow Machines

Let us assume that TimeVM consists of two Shadow VM's such that the first Shadow VM will discover an infection. Therefore, the selected Shadow VM will be the second Shadow VM. Let us denote that the time lag of the first and the second Shadow VM as T^1_{lag} and T^2_{lag} respectively. Now we need to find out what the value of R is. As shown in Figure 3, an infection has been detected at time t_2 and it has been discovered at time t_3. After time t_3, the replay process that is associated with the selected Shadow VM is triggered to switch to fast replay mode. Actually, there is a fraction of time required to trigger the replay process, but we are going to ignore it here. The replay process keeps replaying packets till it reaches time t, at which the Live VM and the selected Shadow VM are running at the same time. At this moment, the system is recovered.

The time quantity of T_{fm} depends at which rate the replay process replays the traffic. Since the traffic may compose of encrypted packets along with non-encrypted packets, therefore the replay rate depends on the amount of encrypted packets in the traffic log and the time required by TimeVM to decrypt/encrypt the packets.

To estimate the replay rate, we use a Markov chain to model the encryption overhead, as illustrated in Figure 4. We define α as the percentage of having encrypted packets in the traffic log. A transition from state i to itself with a probability of $(1-\alpha)$ means that a non-encrypted packet will be replayed. A transition from state i to state $i+1$ means an encrypted packet will be replayed. The encrypted packet will be re-encrypted with an average service time of $1/\upsilon$.

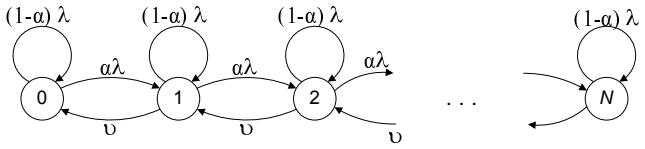

Figure 4: Markov chain to estimate the overhead of packet's re-encryption

The service time solely depends on the packet length and the CPU power. Suppose that the packet length follows exponential distribution and the CPU power is constant, then the mean service time to encrypt a packet will be exponentially distributed. Taking this assumption into consideration, the replay rate (with encryption/decryption), denoted as λ_e, can be obtained by computing the system throughput. Thus,

$$\lambda_e = \upsilon \left(1 - \frac{1 - \frac{\alpha\lambda}{\upsilon}}{1 - (\frac{\alpha\lambda}{\upsilon})^{N+1}} \right)$$

Hence, the general formula for packet's replay rate is

$$\lambda = \begin{cases} \lambda & \text{if } \alpha = 0 \\ \lambda_e & \text{Otherwise} \end{cases} \quad (1)$$

To obtain T_{fm}, we utilize the fact that the number of packets processed by the Live VM at time t is equal to the number of packets processed by the second Shadow VM. By using Little theorem [12], we have

$$\lambda \times t = \lambda \times (t - T^2_{lag} - T_{fm}) + \lambda_a \times T_{fm}$$

Then,

$$T_{fm} = \frac{\lambda \times T^2_{lag}}{\lambda_a - \lambda} \quad (2)$$

Therefore, R can be expressed as

$$R = \frac{\lambda\, T^2_{lag}}{\lambda_a - \lambda} + TTD \quad (3)$$

It is clear from equation (3) that the system recovery time increases as T^2_{lag} increases if we fixed TTD. However, in real practice, it is difficult to determine TTD. Also, TTD is directly influenced by T^1_{lag}. Let us assume that $TTD = \{d \in \Re^+ \cup \{0\}\}$ is a random variable that follows a certain distribution $f(d)$ with the mean $E(TTD)$. Then, the expected recovery time is given by

$$E(R) = \frac{\lambda\, T^2_{lag}}{\lambda_a - \lambda} + E(TTD) \quad (4)$$

Notice that the system can successfully mitigate or eliminate an infected traffic only if T^2_{lag} is greater than d. To setup the value of T^2_{lag}, there are two factors that determine its value. The first factor is the probability, ρ, that the system can certainly be recovered. For example, if $\rho = 0.90$, it means that 90% of detected attacks can be mitigated or eliminated by using TimeVM. This probability can be expressed as $\Pr\{d < T^2_{lag}\} = \rho$. We will call ρ a *confident factor*. The second factor is the maximum storage space, S_{max}, required to store the traffic. S_{max} can be expressed as $S_{max} = \lambda \times T^2_{lag} \times m$ where m is the average packet size, and it should not exceed the maximum space available in the system. If the system has enough space to store large files,

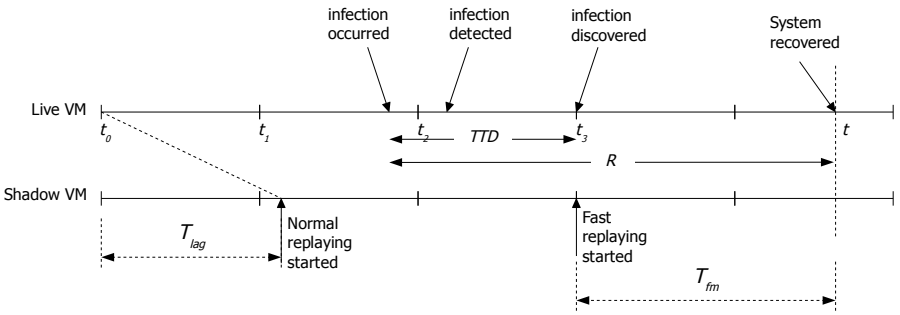

Figure 3: Timelines for the Live VM and the Shadow VM

then we setup T^2_{lag} based on the first factor. Otherwise, we setup T^2_{lag} based on the available space in the system.

It is worth to mention that if the IDS is capable of detecting an infection instantly, it will be sufficient to deploy only two Shadow VM's. TTD, in this case, is only determined by the time required by DACODA to discover the infection, which can be expressed as function of T^1_{lag}, *i.e.*, $TTD = h(T^1_{lag}$. Thus, our objective is to set T^1_{lag} and T^2_{lag} to its minimum possible values in order to have optimized recovery time. Let us denote the minimum possible value is T_{min}.

T_{min} can be set by measuring the time required by the IDS to inform TimeVM manager and the time required by TimeVM manager to invoke DACODA. Using Equation 4, the recovery time is given by

$$E(R) = \frac{2\lambda T_{min}}{\lambda_a - \lambda} + h(T_{min}) \tag{5}$$

4.2 Optimizing R for Multiple Shadow Machines

In the previous section, T^2_{lag} could be very large in order to improve the reliability of the system. This implies that the expected recovery time will also be large. To reduce it, we need to setup several Shadow VMs in the system. Let N be the number of Shadow VMs, let T^i_{lag} be the time lag of the i^{th} Shadow VM number, and let $R(i)$ be the expected recovery time if the i^{th} Shadow VM was selected, and is expressed as

$$R(i) = \frac{\lambda T^i_{lag}}{\lambda_a - \lambda} + E(TTD)$$

The probability that the i^{th} Shadow VM is selected, denoted by $p(i)$, can be expressed as

$$p(i) = \Pr\{T^{i-1}_{lag} < d < T^i_{lag}\} = \int_{T^{i-1}_{lag}}^{T^i_{lag}} f(t)\,\mathrm{d}t$$

where $p(1) = 0$. Therefore, the average expected recovery time for the N Shadow VMs can be expressed as

$$\begin{aligned} E(R) &= \sum_{i=1}^{N} p(i) \times R(i) \\ &= \frac{\lambda}{\lambda_a - \lambda} \sum_{i=2}^{N} \left(T^i_{lag} \int_{T^{i-1}_{lag}}^{T^i_{lag}} f(t)\,\mathrm{d}t \right) \\ &\quad + E(TTD) \end{aligned} \tag{6}$$

Deploying a Shadow VM is associated with a cost c. This cost is expressed in terms of resource consumption and bandwidth usage. Let C represents the maximum cost allowed in the system, then the maximum number of Shadow VMs that can be deployed is constrained by C/c.

Another reason that we should deploy multiple Shadow machines is that the value of TTD is random. Therefore, we need to find out how many virtual machines are needed and how to distribute them on the time space such that the average recovery time is minimum. In other words, we need to find the values of $N, T^1_{lag}, T^2_{lag}, ..., T^N_{lag}$ such that $E(R)$ takes the minimum value.

This is an optimization problem with $N + 1$ variables. To reduce the number of unknown variables, we can configure N to take its maximum value which is C/c. Also, T^N_{lag} is determined by the confident factor ρ and the maximum storage available,S_{max}. The problem now can be formalized as follow. Given λ, λ_a, N, T^N_{lag}, and the distribution function $f(d)$, we need to find out the vector $\{T^*_{lag}\} = \{T^1_{lag}, T^2_{lag}, ..., T^{(N-1)}_{lag}\}$ which minimizes the objective function

$$minimize \sum_{i=1}^{N} p(i) \times R(i)$$

subject to the following constraint

$$T^1_{lag} < T^2_{lag} < \cdots < T^N_{lag}$$

There are several techniques in literature to solve the above optimization problem [4]. We used optimization toolbox in Matlab software to obtain the vector $\{T^*_{lag}\}$ for different distribution functions of TTD. We report the results on section 6.

5. TIMEVM IMPLEMENTATION

In this section, we give detailed description for the core components of TimeVM: TimeVM Manager, NetDd and Replay process.

5.1 TimeVM Manager

TimeVM Manager is responsible to manage and control NetDd and Replay processes. It provides a command-line interface for system administrator to configure TimeVM and virtual machines in the system.

TimeVM manager reads a configuration file that defines the IP addresses of Live VM and Shadow VMs, the number of interfaces in the Linux gateway server, and the maximum number of shadow virtual machines in the system. Then it performs the following tasks:

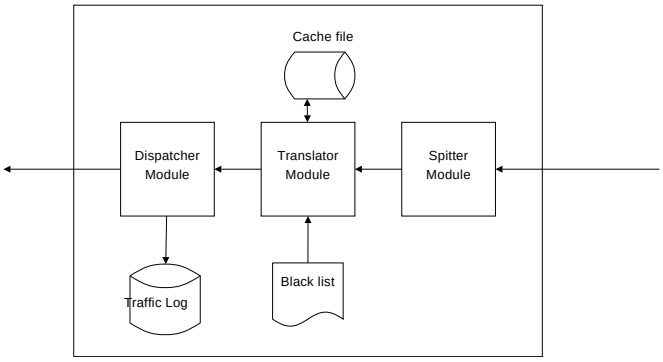

Figure 5: NetDd Modules

- It *registers* the IP addresses of the Live VM and all shadow machines to NetDd.
- It configures the iptables of the Linux server to prevent all TCP traffic from being processed by the protocol stack routines except for those packets that are generated by the Shadow VMs because it will be used by replay processes to perform traffic replay. Please note that other types of traffic, such as UDP, will also be handled by the upper layers of the Linux server. In our current implementation, we handled only TCP packets.
- It displays a shell prompt to allow system administrator to perform some administrative tasks like shutting down a virtual machine, creating a new virtual machine, configuring time-lag for a specific virtual machine, etc. Furthermore, it enables system administrator to execute Unix/Linux commands.

In future work, we will improve TimeVM so that a limited number of traffic will be handled by the gateway server. As a result, this will enhance the security level of the Linux server. This is very important because Linux server stores very sensitive information such as traffic log and a set of private keys used by certain functions/routines to handle SSL replaying.

5.2 Network Distributor Daemon (NetDd)

NetDd is responsible to redirect an incoming packet to its appropriate destination. It is acting like a transparent NAT server. Traffic comes from clients will be redirected to the IP address of the Live VM and the traffic comes from the Live VM will be redirected back to the clients.

As shown in Figure 5, NetDd is composed of three modules. The following is a description of each module:

- **Splitter Module:** This module is responsible to sniff incoming packets and check the source IP address of these packets. If the source IP address of an incoming packet is belonging to the IP address of a Shadow VM, then the module will ignore it since this packet will be handled by the protocol stack routines. Otherwise, the splitter module will forward the received packet to the translator module.
- **Translator Module:** This module is used to keep track of all TCP connections between clients and the Live VM. Each connection is stored in a session. A session is a data structure that stores session key, client state, server state, client's MAC address, cycle number, and a pointer to a function handler. Client state is the TCP state (such as `SYN_SENT`, `CLOSE_WAIT`, etc) with respect to a client. Server state is the TCP state with respect to the Live VM. These states determine the life time of a session. Cycle number will be explained later. The function handler is used to implement an additional processing for a current received packet.

  The module goes into the following steps when it receives a packet as illustrated in Algorithm 1. First, the module checks if the packet is an IP packet or not. If it is not, the module will ignore it and wait for the next packet. Second, it checks the packet protocol, and again if the protocol is not a TCP then the packet will be ignored because, with our current implementation, we focused only on handling TCP traffic. Third, the module constructs a 64-bit hash key based on the source IP address field in the IP header. If the source IP address is the IP address of the Live VM then the hash key will consist of a destination address followed by a destination port number followed by a source port number. Otherwise, the hash key will consist of a source IP address followed by a source port number followed by a destination port number. Forth, the module looks for the key in a TCP lookup table. If the key is found, the session's state and its information will be updated accordingly. If the key is not found, the module checks if the packet is a SYN packet. If it is a SYN packet then a new session will be created. Otherwise, the packet will be dropped.

  It is worth to mention that the module maintains two global variables: a counter to count the number of sessions that have been created without a completed three-way handshaking and a cycle number that takes two values 0 and 1. This means that the counter is incremented when a new session was created, and is decremented when the session has completed three-way handshaking normally or by RST packet. When this counter reaches a predefined threshold (in our implementation, we set it to 50), it triggers a timer and increments the cycle number by 1 (modulus two addition). When the timer expires, it invokes a garbage collector to delete all sessions that have not yet established a connection and have a cycle number not equivalent to the current cycle number. The later condition prevents the garbage collector to delete the newly created sessions.
- **Dispatcher Module:** The Dispatcher module is responsible to modify, log, and send the packets to the appropriate interface. As a design issue, there are two ways to send a client packet to the Live VM or the Live VM packet back to the client. The first way is to create two sockets per connection: one socket between a client and NetDd, and another socket between NetDd and the Live VM. This means that NetDd is acting as a proxy server. The second way is to use libpcap to receive and send packets after modifying some parts of a packet header. Since the first approach is very expensive, we have adopted the second approach.

Algorithm 1 Translator (INPUT: data as PACKET)

```
if data is not IP or TCP packet then
  return
end if
if source(data) == CLIENT then
  key = genkey(data.srcIP, data.srcPort, data.destPort)
else
  key = genkey(data.destIP, data.destPort, data.srcPort)
end if
session = lookup(session_table, key)
if session is null then
  if data.flag has SYN flag then
    create_session(session)
    store session information
    insert_session(session_table, session, key)
  end if
else
  update session information
end if
```

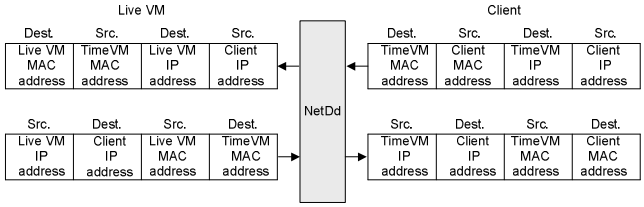

Figure 6: Packet forwarding

Figure 6 shows how a packet is exchanged between a client, NetDd, and the Live VM. When NetDd receives a packet from a client, it modifies the followings:

- It replaces the destination MAC address by the MAC address of the Live VM
- It replaces the source MAC address by the MAC address of the TimeVM (the MAC address of the Linux server)
- It replaces the destination IP address by the IP address of the Live VM.

When NetDd receives back the response of the Live VM, it modifies the followings:

- It replaces the destination MAC address by the MAC address of the client. We stored the client MAC address in the session data structure.
- It replaces the source MAC address by the MAC address of TimeVM (the MAC address of the Linux server)
- It replaces the source IP address by the IP address of TimeVM.

There is a little bit overhead in this approach. Modifying the fields of IP header requires re-calculating the checksum. We measured the modification and recalculation overhead by running two experiments. In the first experiments, we let a client host to connect directly to the Live VM. In the second one, the client connects to the TimeVM server first. As shown in table 1, the first column represents the timestamps when we sent an HTTP request directly to the Live VM without the interference of NetDd. The second column represents the timestamps for the second experiment. The timestamps are collected by using `wireshark` program at the client host. We conclude that the forwarding overhead is almost negligible.

N	timestamp w/o NetDd	timestamp w/ NetDd
1	13.751163	17.473093
2	13.751433	17.486035
3	13.751439	17.486086
4	13.751556	17.486241
5	13.763961	17.488031
6	13.764077	17.489360
7	13.764103	17.489973

Table 1: Forwarding overhead due to packet modification and checksum calculation

5.3 Replay Processes

In this section, we describe the most critical function of TimeVM which is traffic replay. Traffic replay is performed by replay processes. Each process is associated with a single Shadow VM as shown in Figure 1, and is configured with a *time lag* that indicates the distance in time between the Live VM and its associated Shadow VM.

In normal replay mode, a replay process should always preserve the time difference between the current system time and the current replayed packet time to amount equals to its time lag. One way to do that is to implement a polling mechanism that continuously invokes *gettimeofday()* system call until the difference between the packet time and current system time is less than or equal to time lag. Although this approach provides a high degree of accuracy, it consumes a lot of CPU time. Another approach is to use *usleep()* system call. We implemented the second approach as follows. Initially, each replay waits until it reads the first packet. Then, it goes to sleep for amount of time equals to its time lag. After that, it starts to replay the first packet. Before replaying the next packet, the replay process calculates *silent_time* which is the time difference between the next packet's timestamp and the previous packet's timestamp. If *silent_time* is greater than two times a threshold, the replay process goes to sleep for amount of time equals to *silent_time* minus threshold. Otherwise, the replay process continues to replay the next packet. The value of threshold represents the amount of time required for the operating system to execute *usleep*() system call. Using lmbench [14], *usleep()* overhead is approximately $12.50\mu s$ in a Pentium 4 machine 1.6 GHz with 512MB RAM, running Linux version 2.6.18.

In fast replay mode, the value of *silent_time* will be shortened by a predefined factor value. The factor is defined as a multiple of 2. Although, shortening *silent_time* eventually increases the rate of replayed packets per second, it requires some considerations. First, it should not affect the behavior of the network service. In other words, the server's response in normal replay mode and the server's response in fast replay mode (for the same set of packets) should be identical. Second, it should not overload the network service capacity.

Replaying packets correctly is a very complex task. In our early stage of the implementation, we used libpcap and libnet libraries to handle and manipulate packets. But then we have faced many complex design issues such as maintaining the receiver window size (defined during three-way

handshaking in a TCP session), handling TCP options, etc. Therefore, we changed our implementation path to use sockets instead of libpcap and libnet libraries. This is why we configured TimeVM to allow Shadow VM's packets to be processed by higher layers. Using sockets to handle packets implied to other complications in our design: how to handle different types of sessions (explained in the next paragraph), how to send packet payloads in-order if they received and logged out-of-order, and how to handle packet fragmentation.

Due to the nature of TCP protocol and variety of protocols building on top of TCP such as SSL and FTP, the replay process should be protocol-aware in order to properly replay TCP traffic. We have classified three types of sessions: (1) sessions with a static service number such as HTTP session, (2) sessions with multiple service numbers like FTP session, and (3) sessions with encrypted payloads such as SSL sessions. The functionality of each session is defined as plug-in for replay process. This allows us to extend our replay process to accommodate other application protocols.

To handle SSL sessions, we adopted the code written at [1]. The private keys that was stored in Live VM will also be stored in the Linux Server. These keys are needed in order to properly decrypt the encrypted traffic. To reduce the overhead of decryption process, we log a payload after decrypting it. A replay process needs only to re-encrypt the payload when it replays SSL sessions. In our current implementation on replaying SSL, we did not implement client authentication. Client authentication is useful if the server wanted to restrict access to some services to only certain authorized clients. One way to handle this limitation is to define a global trusted user locally in Linux server. The drawback is that it requires to modify application to accept that defined global user.

A TCP session is uniquely identified by one of two keys: the hash key as we described in section 5.2 and Session Identifier (SID). SID is a sequence number of integers that is always incremented whenever a new session is created. Please note that the data structure of a TCP session is not similar as the data structure of NetDd's session. In SSL, SID represents session ID defined during SSL handshaking. It is used when SSL protocol generates a new session and new cryptographic keys.

In order to replay packets in-order, each session maintains two variables: "*exp_seq_num*" and "*leftover*". *exp_seq_num* represents the expected sequence number of the next "client's" packet, and *leftover* is a counter that keeps track the number of packets stored in a hash table called "*unordered_table*". Now, when the replay process reads the next packet, it checks to see if the sequence number defined in the TCP header is matching the expected sequence number *exp_seq_num*. If they do not match, the packet will be stored in *unordered_table* with a key constructed as the concatenation of SID and the sequence number in the TCP header, and the *leftover* variable will be incremented by one. Otherwise, the packet will be replayed and sent to its associated Shadow VM. After replaying the packet, the replay updates the expected sequence number (by adding the payload length to *exp_seq_num*) and checks the value of *leftover*. If the value is greater than zero, then the process searches the hash table. If the constructed key [SID | expected sequence number] is found, the stored packet will be replayed and deleted, and the value of *leftover* will be decremented. The process keeps searching the hash table until either the constructed key is not found or *leftover* value becomes zero. Please note that, we do not need to keep track of the expected sequence number for the server packets since these packets are dropped. Algorithm 2 shows only the part that replaying a client packet.

Algorithm 2 Replay (INPUT: data as PACKET)

```
if source(data) == CLIENT then
  key = genkey(data.srcIP, data.srcPort, data.destPort)
  session = lookup(tcp_session_table, key)
  if session is null then
    if data.flag is SYN then
      session = create_new_node()
      store session information such as exp_seq_num, leftover
      insert_session(tcp_session_table, session, key)
    end if
  else
    compute payload size
    if payload size > 0 then
      if data.seqnum == session.exp_seq_num then
        replay data and update session.exp_seq_num
        update TCP state in TCP session
        while session.leftover > 0 do
          key2 = concat(session.sid, session.exp_seq_num)
          if data = lookup(unordered_table, key2) != null
          then
            replay data and update session.exp_seq_num
            session.leftover = session.leftover - 1
          end if
        end while
      else
        key2 = concat(session.sid, data.seqnum)
        store data in unordered_table with key2
      end if
    else
      update TCP state
    end if
  end if
end if
```

Handling IP fragmentation is another complex problem. In our current implementation, we assumed that there is no IP fragmentation. The Translator module in NetDd drops any fragmented packet. We put IP fragmentation as a future work.

6. EVALUATION AND NUMERICAL RESULTS

In this section, we report some numerical results of our analytical model. The purpose is to study the feasibility of deploying TimeVM in real practice. We first examine the impact of increasing packets' replay rate during fast replay mode on the system recovery time. For this case, we fix N to two shadow VM's, T_{min} to 2 seconds, and TTD to 10 seconds, *i.e.*, $h(t) = 5t$. Equation 4 can be rewritten as

$$E(R) = \frac{2T_{min}}{\pi - 1} + 5 \times T_{min} \quad (7)$$

where $\pi = \lambda_a/\lambda$. Here, π expresses the magnitude of rate increase, and it is more convenient to study the effect of rate increase than specifying explicit values for λ and λ_a. Figure 7 shows the relation between the expected recovery time and the ratio π. As shown, there is a significant improvement on the system recovery time as the ratio goes from 1 up to 2, then the recovery time improves slowly after 2.5. We can

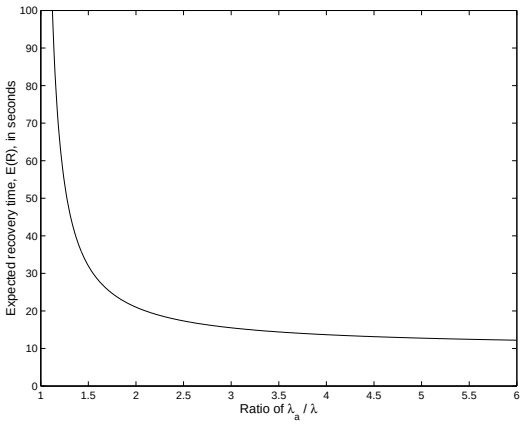

Figure 7: The impact of λ_a/λ on system recovery time

conclude that we do not need to pump high volume of packets to the network in order to reduce the average recovery time. Based on Figure 7, it is recommended to make the ratio of π between 1.5 and 2.5. In our implementation, we can adjust this ratio by tuning *silent_time* value.

In real practice, increasing the ratio π could lead to a disaster situation. If the Live VM is hosting a busy network server, then increasing packets rate could overwhelm the network servers on the Shadow VMs and lead to packets dropping. As shown in [2], the system utilization of HTTP server is linearly increasing as the number of requests per second is increased. This means that if the server utilization is 80%, then increasing the ratio from 1 to 1.25 will eventually increase the server utilization to 100%. As a future work, the value of π should be adjusted based on the server utilization.

Second, we examine the effect of N on system recovery time. As we mentioned before, if TTD is indeterministic, then we need to deploy multiple shadow VMs to reduce the system recovery time, and we assumed that TTD follows a certain distribution. Remember that TTD is composed of time to detect and time to discover. In literature, finding a suitable distribution for TTD has shown to be difficult. However, for the evaluation purpose, we consider two distributions that approximately simulate the real situations: Exponential and Gamma distributions. We select exponential distribution to simulate the case that it is more likely that TTD has a shorter time (*i.e.*, closer to zero) and less likely a longer time. We select Gamma distribution to simulate the case that TTD is more likely around a certain time.

Table 2 shows the value of $\{T^*_{lag}\}$ for different values of N when TTD follows an exponential distribution with the mean value 10 seconds. Table 3 shows the values when TTD follows a Gamma distribution with $\alpha = 5$ and $\theta = 2$. In both cases, we fix the ratio of λ_a/λ to 1.5 and the confident factor ρ to 0.95. As shown, there is a dramatic reduction in system recovery time when we deploy three Shadow VM's instead of deploying two Shadow VM's. Also, as the number of Shadow VM's increases the expected recovery time decreases. However, there is no significant decrement of $E(R)$ when N is increased from 5 to 6.

Figure 9 and 8 illustrate the values of $E(R)$ with high values of N (*i.e.*, $N \geq 7$), and with different values of π. It

N	T^1_{lag}	T^2_{lag}	T^3_{lag}	T^4_{lag}	T^5_{lag}	T^6_{lag}	$E(R)$
2	10.73	30.0					41.65
3	6.42	15.41	30.0				35.56
4	4.56	10.33	18.15	30.0			32.87
5	3.53	7.77	13.04	19.98	30.0		31.36
6	2.88	6.22	10.17	15.04	21.3	30.0	30.4

Table 2: T_{lag}'s values and $E(R)$ when TTD is exponentially distributed

N	T^1_{lag}	T^2_{lag}	T^3_{lag}	T^4_{lag}	T^5_{lag}	T^6_{lag}	$E(R)$
2	10.69	18.31					35.38
3	8.39	12.6	18.31				32.74
4	7.2	10.27	13.66	18.31			31.49
5	6.44	8.93	11.43	14.35	18.31		30.76
6	5.9	8.04	10.07	12.25	14.85	18.31	30.28

Table 3: T_{lag}'s values and $E(R)$ when TTD is following Gamma distribution

is obvious that $E(R)$ is significantly reduced as we deploy 3 or 4 Shadow VMs. Then, we notice that $E(R)$ is almost a horizontal line after $N = 5$. This implies that adding more than 5 virtual machines to the system may not be worthy because the cost of adding one additional VM is more than the gain (in terms of reducing the recovery time). The two figures also illustrate the impact of increasing π on the recovery time for different values of N. As we said before, the recovery time is improved significantly when the ratio is more than 1 and less than 2.5.

In the following paragraphs, we study the influence of confident factor on our framework. Figure 10 shows the effect of confident factor, ρ, on the storage requirement. As we can notice, the average storage size increases smoothly when ρ increases from 0.5 to 0.85. After this point, the average storage starts to increase significantly and rapidly after 0.95.

Next, we examine the effect of confident factor on system recovery time. Figure 11 depicts this effect on a system that has 1, 2, 3, 5, or 7 Shadow VMs and TTD is exponentially distributed with a mean value equals to 10 seconds. Figure 12 depicts the same effect for the same system configurations but TTD follows a Gamma distribution with $\alpha = 5$ and $\theta = 2$ (the mean value is 10 seconds). As shown, the expected recovery time increases as ρ increase. This is because, ρ configures the time lag of the last Shadow VM. Low value of ρ means that the difference in time between the Live VM and the last Shadow VM is small. However, the probability of unsuccessful recovery will be high. High value of ρ means that the time lag of the last Shadow VM is high, and the probability of unsuccessful recovery is low. Figure 11 and 12 also show that the impact of confident factor on expected recovery time does not depend on the distribution of TTD.

7. CONCLUSION AND FUTURE WORK

We propose a framework called TimeVM to achieve an online mitigation and recovery of network servers under-attack. The framework is based on traffic-log-replay recovery and virtualization technology. TimeVM consists of several virtual machines running on different time lags. These machines represent multiple historical snapshots of the original virtual machine at different times in the past. Whenever

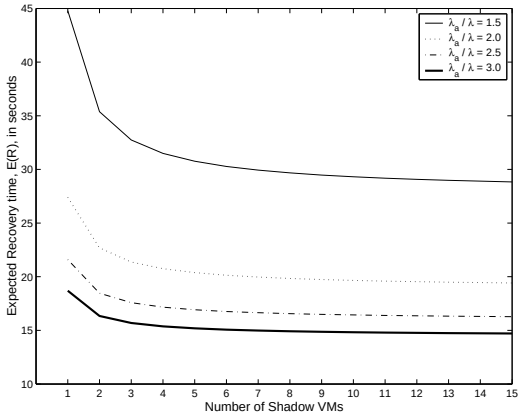

Figure 8: The impact of reliability factor ρ on system recovery time

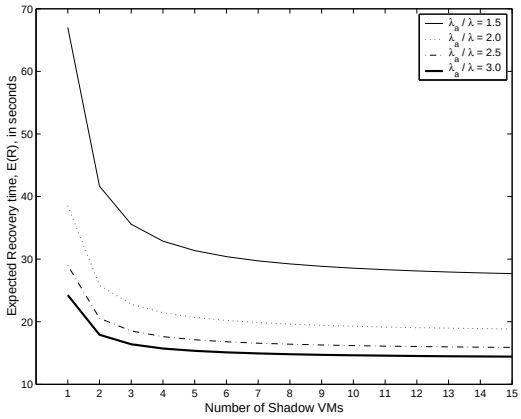

Figure 9: The impact of reliability factor ρ on system recovery time

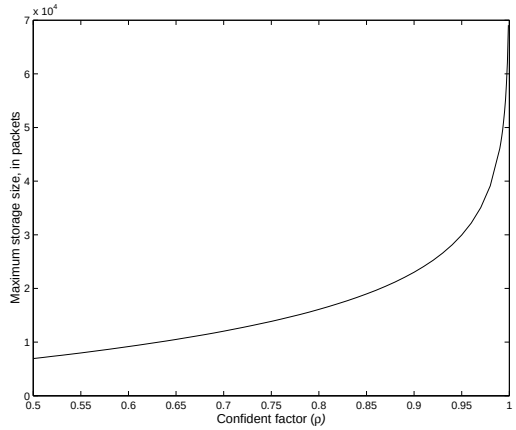

Figure 10: The impact of confident factor ρ on storage capacity

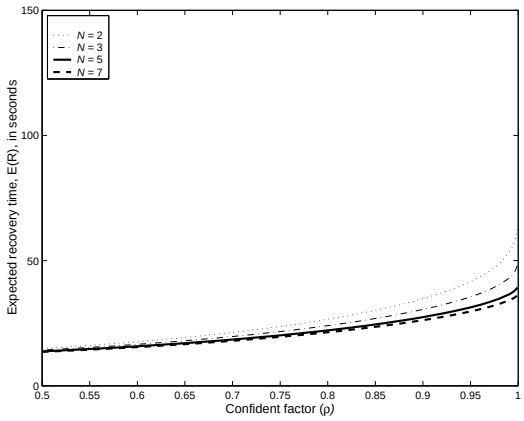

Figure 11: The impact of confident factor ρ on system recovery time when TTD is exponentially distributed

the system detects an infection, it selects the first shadow VM before the infection time, and then fast-replay the entire traffic after removing the infection traffic from the logs. This shadow VM will replace the contaminated VM as the infection is mitigated.

The paper addresses the theoretical and implementation challenges to respectively optimize the recovery time and replay network traffic. One significant contribution of our work is to achieve an online mitigation as fast as possible. This requires proper configuration of system parameters. Minimizing recovery time has been formulated as a constrained nonlinear optimization problem. We used an available evolutionary approach to solve this optimization problem.

The evaluation results look very appealing in practice. We show in section6 that replaying traffic as twice as fast as original traffic speed is sufficient to obtain the minimum recovery time. This offers a practical solution. The evaluation results also show that it is sufficient to deploy a small number of shadow VMs (*i.e.*, not exceed 5 VMs) to achieve a very reasonable recovery time in most cases. These results reveal that TimeVM can be deployed easily and economically in real systems.

We will continue to develop TimeVM to accommodate different type of traffic such as UDP, ICMP, and multicast traffic. Then, we build a complete prototype with different types of exploitation to get some experiences with TimeVM.

8. REFERENCES

[1] *http://www.rtfm.com/ssldump*.

[2] T. F. Abdelzaher and C. Lu. Modeling and performance control of internet servers. In *Proceedings of the 39th IEEE Conference on Decision and Control*, volume 3, pages 2234–2239, 2000.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[4] A. Cichocki and R. Unbehauen. *Neural Networks for Optimization and Signal Processing*. John Wiley and Sons, 1993.

[5] J. R. Crandall and T. F. Chong. Minos: Control data attack preventing orthogonal to memory model. In *In Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, 2004.

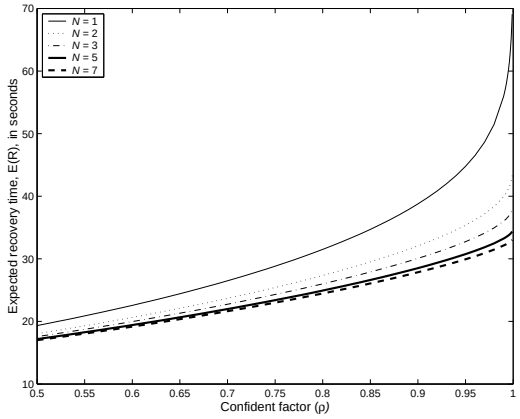

Figure 12: The impact of confident factor ρ on system recovery time when TTD follows Gamma distribution

[6] J. R. Crandall, Z. Su, S. F. Wu, and T. F. Chong. On deriving unkown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. *ACM CCS*, pages 235–248, 2005.

[7] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong. Execrecorder: Vm-based full-system replay for attack analysis and system recovery. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 66–71, 2006.

[8] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36:211–224, 2002.

[9] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, 2008.

[10] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic. Comprehensively and efficiently protecting the heap. *ASPLOC*, 2006.

[11] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.

[12] J. D. C. Little. A proof of the queueing formula $l = \lambda w$. *Oper. Res.*, p:383–387, 1961.

[13] M. V. Mahoney. Network traffic anomaly detection based on packet bytes. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 346–350, 2003.

[14] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX*, 1996.

[15] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, 2005.

[16] I. Ray and S. Tideman. A secure tcp connection migration protocol to enable the survivability of client-server applications under malicious attack. *J. Netw. Syst. Manage.*, 12(2):251–276, 2004.

[17] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, 2001.

[18] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 3–3, 2004.

[19] F. Sultan, K. Srinivasan, and L. Iftode. Migratory tcp: connection migration for service continuity in the internet. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 469–470, 2002.

[20] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Rethinking the design of virtual machine monitors. *IEEE Computer*, 38(5):57–62, 2005.

[21] M. Xu, R. Bodik, and M. D. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News*, 31(2):122–135, 2003.