2006

# Geometry-based Detection of Flash Worms

Sang Soo Kim
*San Jose State University*

# GEOMETRY-BASED DETECTION OF FLASH WORMS

**A Writing Project**

**Presented to**

**The Faculty of the Department of**

**Computer Science**

**San Jose State University**

**In Partial Fulfillment**

**Of the Requirements for the Degree**

**Master of Science**

**By**
**Sang Soo Kim**

**December, 2006**

Approved by: Department of Computer Science

College of Science

San Jose State University

San Jose, CA

______________________________
Dr. Mark Stamp

______________________________
Dr. Chris Pollett

______________________________
Dr. Agustin Araya

## Abstract

While it takes traditional internet worms hours to infect all the vulnerable hosts on the Internet, a flash worm takes seconds. Because of the rapid rate with which flash worms spread, the existing worm defense mechanisms cannot respond fast enough to detect and stop the flash worm infections.

In this project, we propose a geometric-based detection mechanism that can detect the spread of flash worms in a short period of time. We tested the mechanism on various simulated flash worm traffics consisting of more than 10,000 nodes. In addition to testing on flash worm traffics, we also tested the mechanism on non-flash worm traffics to see if our detection mechanism produces false alarms.

In order to efficiently analyze bulks of various network traffics, we implemented an application that can be used to convert the network traffic data into graphical notations. Using the application, the analysis can be done graphically as it displays the large amount of network relationships as tree structures.

## Table of Contents

## List of Figures

# 1. Introduction

From the early computing days, many computer worms have been developed and spread on the Internet, often causing disruptions on the network. As computer network architecture and internet security continue to advance to prevent the spread of worms, the worms, too, continue to evolve as well to find their ways to infect more hosts in reduced infection time.

In recent years, due to faster internet connections and higher computing power available, it is now possible to develop a more powerful computer worm that can possibly bring down the whole Internet in less than 30 seconds [8]. Such a worm is termed a “Flash Worm”, and this master’s project will consider a possible defense against flash worms.

To date, flash worms are only theoretical in the sense that no reported incidents have been filed. However, one conjectured implementation of a flash worm is to pre-scan the entire Internet to generate a hit-list of all potentially vulnerable hosts. The initial worm divides the hit-list into *n* blocks, infects one address from each block, and then communicates the list of remaining addresses for each block to newly infected host [2]. This process continues until all the addresses on the hit-list have been infected. Similar to the Slammer worm, which is capable of infecting “more than 90 percent of vulnerable hosts within 10 minutes” [6], the flash worm effectively utilizes network bandwidth; the higher the bandwidth, the faster the worm spreads. However, in contrast to the Slammer worm, this flash worm implementation does not scan for vulnerable target addresses while spreading.  Since searching for vulnerable target addresses is a slow process, eliminating this process allows the unprecedented spread rate of flash worms.

Because of the rapid speed with which a flash worm spreads, some of the prediction and mitigation algorithms applicable to other worms are not appropriate for flash worms.  In this project, we are proposing a new geometry-based detection algorithm of flash worms, in which the local network activity is monitored.  Based on the monitored activity, we plan to create a tree to indicate the spread of a possible flash worm.  This "spread tree" is designed to detect the tree-like structure a flash worm is likely to use to propagate the target host list. If the depth of a spread tree reaches some predefined threshold, shutting down the network will prevent the flash worm from spreading further.  To avoid false alarms, the spread tree must be pruned as well as updated continuously.

In the rest of this paper, we first describe the known conjectured implementations of flash worm.  Section 2 discusses the essential components of flash worm implementation structure.  Section 2 also explains how differentiating these components can result in different infection behaviors.  Section 3 details the design and implementation of our proposed detection method.  Section 4 shows our simulation test processes and the result. Section 5 summarizes and concludes the project.

### 1.1. Contribution of this project paper to the field

This paper provides a perspective of the various flash worm implementation scenarios.  It also discusses how we implemented our simulation of flash worm infections.  While many large-scale worm simulators are implemented by utilizing a worm model, our simulator is implemented to produce the packet-level data traffic among 10,000 network hosts. Furthermore, the paper proposes an effective flash worm detection algorithm and explains how we visualized the vast amounts of flash worm data using a post-analyzer.

# 2. Background

Because there have been no incidents of flash worm outbreaks reported in the wild yet, there is no solid example of an actual flash worm implementation. However, there are many conjectured implementations of flash worms by various researchers.

## 2.1. Common Structure of Flash Worms

Despite the differences, every flash worm implementation is expected to have the following common components:

- <u>Hit-list</u>

  While traditional internet worms scan for target hosts whenever they infect a new host, flash worms do not perform this often time-consuming process. Instead, flash worms utilize a hit-list, which contains the list of all the target host addresses.

  - Hit-list preparation process

    Prior to instantiating the spread of flash worms, the worm author has to scan the entire Internet address to determine the vulnerable hosts. If the author has a fast Internet connection and a high processing power, this scanning can be done in a day. However, to reduce the risk of being detected by defense systems, the author may want to purposely slow down the scanning process. Such stealthy scanning may take several weeks.

- Typical size of one Host Address

  Since most of the network hosts are in IPv4 domain at this time, the usual number of bytes required to represent one host address is 32. That means, a single UDP packet can store about 35 host addresses if the maximum byte size of a UDP packet is 1200.

- Low failed infection attempts

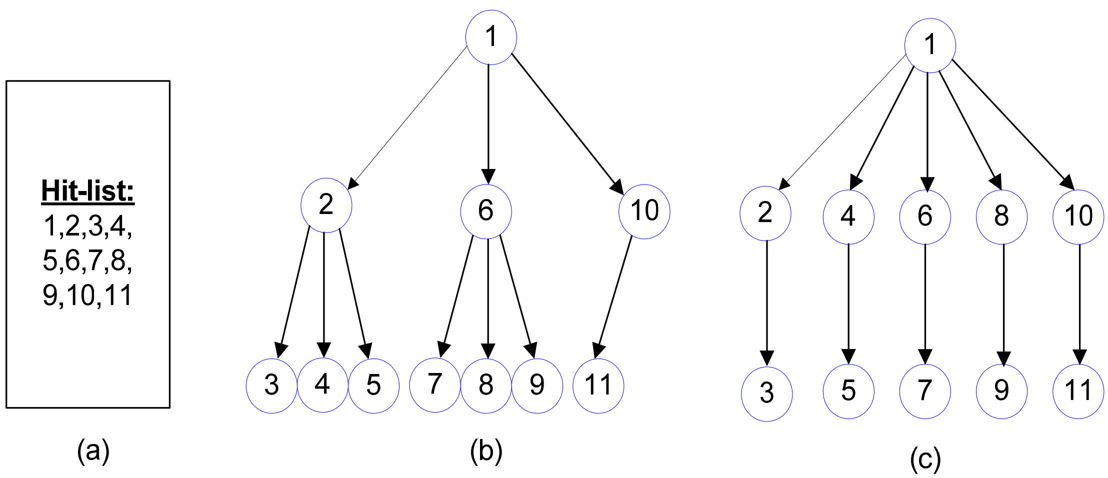
  In addition to the boost of spread speed, the use of the hit-list also protects flash worms from some existing worm containment defenses. For example, the containment defenses that look for unusually high number of missed connections cannot detect the spread of flash worms because most infection attempts are likely to succeed.

- <u>Maximum infection number</u>

  Given a list of addresses to infect, each host does not infect all the listed addresses. Instead, each host attempts to infect a fixed number of addresses at most and propagates the remaining addresses to other hosts.

- `Spread Tree

Even though the usage of the hit-list enables the blazing spread rate of flash worms, it also restricts the flash worms to spread following a tree-like pattern. We call this spread pattern the spread tree. For example, the hit-list of Fig. 1 (a) with maximum infection number of 3 creates the spread tree shown in Fig. 1 (b).




<Fig. 1: Hit-list and 2 Corresponding Spread Trees>
(a) The hit-list contains 11 target addresses; (b) The spread tree formulation if each node attempts to infect 3 other nodes at most; (c) Each node infects 5 nodes at most. Note that only node 1 has the enough target addresses to infect 5 nodes.

To help us analyze the flash worm infection behavior later, let us define some important components of spread trees:

- Parent-Child relationship
    - There is a one-to-one parent-child relationship if one host infects another host. For example, in Fig (b) above, node 6 is a *parent* of node 7, and node 7 is a *child* of node 6.

- Depth (or Generation) number

- The depth of a node is the maximum length of the path from the root to the node. For example, in Fig (b), node 7 has depth of 2 and node 10 has depth of 1.
- The depth can also be called *generation* number. For example, in Fig (b), nodes 2, 6, and 10 are all in the same *first* generation.
- The maximum number of generations required to infect *N* hosts if each host infects *K* other hosts can be determined by this formula:
  - **$O(\log_K N)$**

  For example, in Fig (b), the maximum generation number is $O(\log_3 11) = 3$.

- Node creation (or infection) time
  - Each node has its associated creation time. This is the time the node was first infected by a flash worm. Note that the nodes from a same generation should have similar creation time since they are expected to be infected at about the same time.

- Total infection time
  - The total infection time is the amount of time required to infect all the hosts listed in a hit-list. The total infection time can also be equivalent to the *node creation time* of the last leaf node.

## 2.2. Various design issues of flash worms

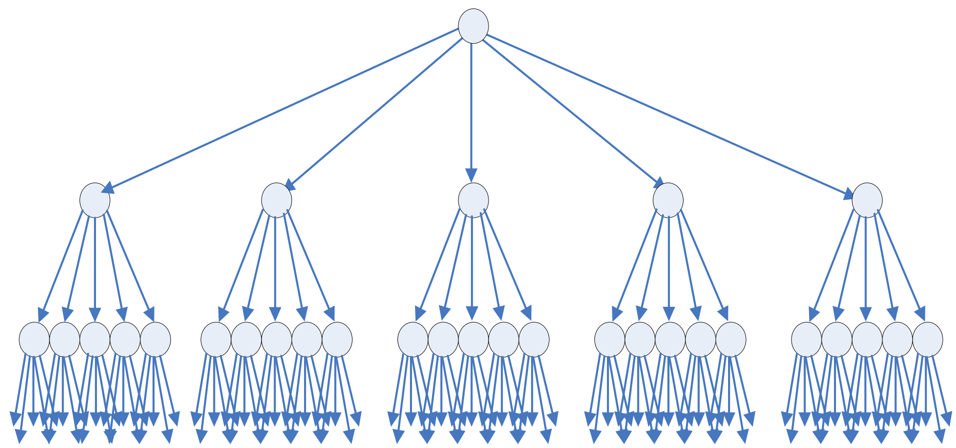
When designing a flash worm, the worm author has to take the following issues into the account as they will characterize the worm infection behavior:

### 2.2.1. Spread Tree Topology

The maximum number of infections each host attempts defines the shape of the spread tree. The authors of [7], who first published the work regarding flash worms, have laid out these three spread tree topologies.

#### 2.2.1.1. Shallow Spread Tree

In a shallow spread tree topology, each host tries to infect many other hosts. As can be seen in Fig. 2 below, it takes 3 generations to infect 155 hosts if each host attempts to infect 5 other hosts.



<Fig. 2: Shallow Spread Tree Topology>

<u>Advantage</u>:

It takes a short amount of time to infect all the target hosts since this topology requires a small number of generations to infect many hosts. Having a couple of hosts with OC-12 connection near at the root node, [8] claims that such a shallow spread tree can infect all the vulnerable hosts on the Internet less than 1 second.

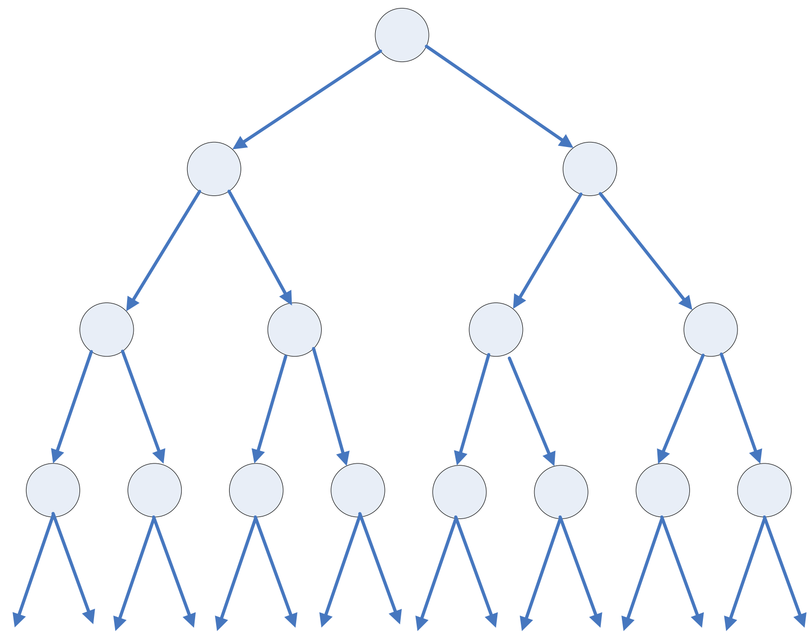
Furthermore, this topology is tolerant against a failed infection. A network host may not get infected by a flash worm for many reasons. The host may no longer have a security hole, or the infection packet may be lost. The problem is if a host is not infected, the hosts constituting its child nodes in the spread tree will not get infected either. Fortunately, the shallow spread tree topology is tolerant against such infection failures. For example, in the Fig. 2 above, suppose one host from first generation fails to get infected. Then, that means, only 1 out of 5 failed and we still have 4 other hosts to continue the infections.

Disadvantage:

As each host tries to infect many other hosts, this topology places significant computation burden on each host. Furthermore, the traditional worm defense systems which look for sudden increase in the number of network connections from a host may detect the worm activities.

2.2.1.2. Deep Spread Tree

In a deep spread tree topology, each host tries to infect only a small number of other hosts. If we were to infect the same 155 hosts again using the deep spread tree topology from Fig 3, it would require 7 generations this time. Recall that the previously discussed shallow spread tree would require only 3 generations.



<Fig. 3: Deep Spread Tree Topology>

Advantage:

Since each host infects a small number of other hosts, the aforementioned defense systems, which look for sudden increase in the number of network connections, cannot be effective against this topology.

Disadvantage:

When comparing to the shallow spread tree topology, this topology requires more generations to infect the same target hosts. Since there is a propagation delay in sending the target list from a generation to the next, more generations imply longer propagation delay. Therefore, this topology takes long time to infect all the target addresses.

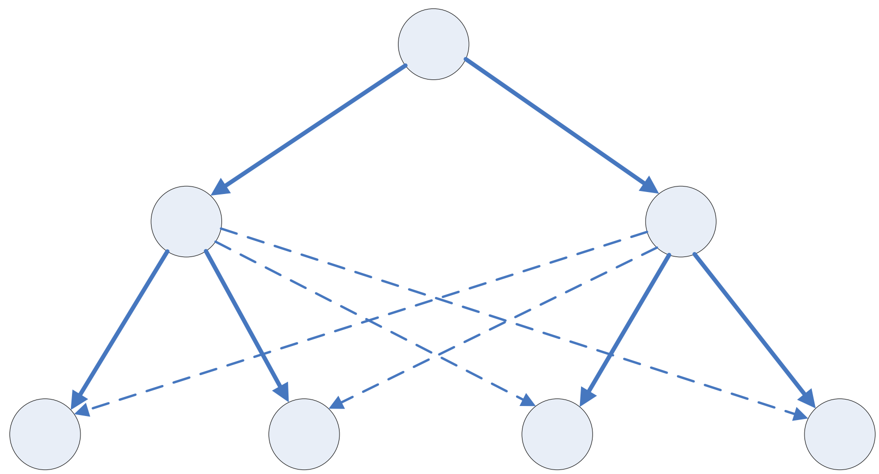
Furthermore, if one node from the early generation fails to get infected, the impact can be devastating.  For example, in the Fig 3 above, if one node from generation 1 fails to get infected, the half of the spread tree would not get infected.

- **Notes regarding our design ( I )**

The flash worm detection mechanism, which we propose in this paper, is effective only for this deep spread tree topology because the mechanism requires to see the spread tree grow to reach a certain depth.

2.2.1.3.  Robust Spread Tree

One of the difficulties of implementing flash worms is to obtain a perfect hit-list.  Since the Internet continues to change as new hosts are being added and deleted, some infection attempts may fail.  As discussed in the deep spread tree topology section, if an infection attempt fails near at the root of the spread tree, the half of the spread tree may not be infected.
To provide a degree of resilience to the failed infection attempts, the following topology can be used.



<Fig. 4: Robust Spread Tree Topology>

In this robust topology, each host tries to infect its neighbor's child hosts. Even though this redundancy may consume additional network bandwidth and processing power, it protects from a failed parent node to decide the fate for all of its child nodes.

### 2.2.2. Use of Threads

In addition to defining the spread tree topology, the flash worm author should decide whether to use threads. The use of threads at the application layer can impact the *total infection time*.

#### 2.2.2.1. No Thread

If a thread is not used, then a host has to wait until it receives all the target addresses from its parent. Until the receiving is completed, the host cannot infect other hosts.

The hosts near at the root node are expected to wait the longest because the target address list is usually huge. On the other hand, as the host is closer to the leaf nodes, the waiting time is expected to be significantly reduced.

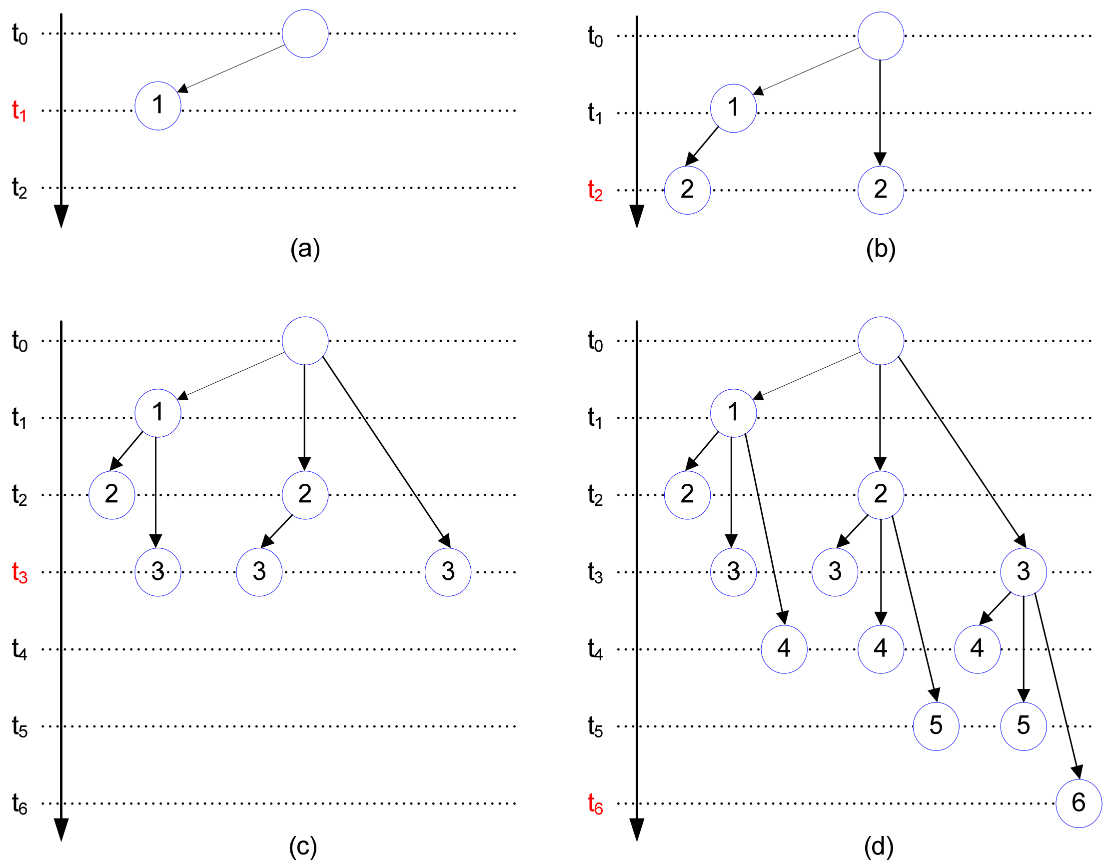
♦ **Notes regarding our design ( II )**

Our simulation implementation of flash worms does not use thread.

Therefore, there are propagation delays which vary with respect to the size of the target list.

2.2.2.2. Single Thread

In an implementation that can utilize a single thread, the host can create an instance of thread to infect its child node immediately after a target address is received. The Fig. 5 illustrates the procedure.




<Fig. 5: Single Thread Usage>
(a) At $t_0$, the root node infects one other node; (b) At $t_1$, the root node infects second node. The previous child node has received a target address from the root node and is ready to infect another node. The node infects another node by creating a thread. (Note that the node is still receiving the target list from its parent); (c) At $t_3$, three new nodes are infected; (d) At $t_6$, all nodes are infected. [2]

By using a single thread, a node does not have to wait until it receives all the target address, which significantly reduces the total infection time.

### 2.2.2.3. Multiple Threads

Multiple threads can be used in the environment where a node receives multiple target addresses from multiple sources at the same time. For example, in a robust spread tree topology, a node may receive multiple target addresses from different parent nodes.

### 2.2.3. TCP or UDP

Another important decision a flash worm author has to make is whether to use TCP or UDP. TCP establishes and maintains a connection before transmitting the data, which requires more packets to be sent than UDP. On the other hand, UDP does not establish a connection and data can be transmitted between two hosts right away. Therefore, a UDP-based flash worm can infect faster than a TCP-based flash worm.

Even though the use of TCP can be slower than using UDP, the connection-oriented nature of TCP provides guaranteed delivery of data whereas UDP does not provide this. As discussed in the spread tree topology section, missed delivery of infection packets in the early stage of worm infection can have a huge impact. Therefore, the worm author has to balance the infection speed against the risk of missed connections when choosing the network transmission protocol.

♦ **Notes regarding our design ( III )**

Our simulation of flash worms uses UDP. Only one UDP packet is required to infect a target host.

### 2.3. New Flash Worm Implementation Design

There are many hypothetical implementation designs of flash worms known today. However, there may be other flash worm implementation designs still unknown. Below, we propose a new flash worm implementation design.

Central Hit-list Server based Design

- Instead of distributing the address list from a parent node to a child node, the hit-list is available from a central server.
- When a new host is infected, it asks the central server for target addresses. Once the central server receives a packet from a target address, it removes the target address from the hit-list (because it knows that the target address is already infected).
- For simplicity and efficiency, every packet is in UDP. Assuming a UDP packet can store 1200bytes, a single UDP packet can store about 35 IP Addresses. That means, a simple Request-Reply UDP packets will get 35 target lists from the server.
- Advantage:
  - The central server can efficiently distribute the hit-list
    - No more initial huge delay in propagating a large hit-list
    - Resilient to failed infections; if a node fails to get infected, other node will infect its child nodes.

    - The hit-list can be updated dynamically as new vulnerable addresses are added.  Also, incorrect addresses can be removed after some timeout.
    - The current infection progress state can be monitored in real time.
- Disadvantage:
    - The hit-list server can be heavily loaded.

# 3. Implementation

In this section, we discuss how we implemented a simulator to generate the flash worm traffic. Then, we discuss how we developed a new application to visualize and better understand the flash worm traffic and to test our proposed detection mechanism.

## 3.1. Flash Worm Simulation

### 3.1.1. Finding a suitable simulator

In order to study the defense against flash worms, we needed to have a simulator to simulate the behavior of flash worms. Because learning to use a new simulator in depth is a difficult and often time-consuming process, we wanted to verify that a specific simulator is capable of simulating many nodes and generating the packet-level traffics prior to learning the simulator in depth.

**List of Simulators:**

Following are simulators that can simulate a vast number of nodes:

1) Scalable Simulation Framework
    a. http://www.ssfnet.org/homePage.html
    b. Outdated - Latest simulator update was in year 2004
2) Simnet
    a. http://simnet1.isi.jhu.edu/

b. The implementation of a worm model was available but the simulator did not have documentation.

3) OPNET

   a. http://www.opnet.com/

   b. This is a commercial-quality network simulator product. Student license was available that required a complex registration.

4) GTNetS

   a. http://www.ece.gatech.edu/research/labs/MANIACS/GTNetS/

   b. A network simulator from Georgia Tech University – complex to install and use

5) NS-2

   a. http://www.isi.edu/nsnam/ns/

   b. Although well documented, very complex to use.

Although it seemed to be very complex to use at first, we decided to use NS-2 because it was easy to install and had some form of the user documentation. However, the other simulators have the capability to do the project as well.

### 3.1.2. Extending simulator to simulate the spread of flash worms

#### 3.1.2.1. Modified NS-2 default environments

- By default, NS-2 uses a flat-routing scheme, in which every node knows the address of every other node, resulting in routing table size to the order of $n^2$ for $n$ number of nodes. To reduce the memory requirement of simulations over large network topologies, we created a hierarchical-routing scheme. For hierarchical-routing, each node knows only about those nodes in its level, thus the routing table size was reduced to the order of $log\ n$.

3.1.2.2. Added a new network packet:

We enhanced NS-2 to support the following new network packet:

- Packet Format

```
struct hdr_fworm {
      nsaddr_t    dst;
      nsaddr_t    src;
      nsaddr_t    target_range_first;
      nsaddr_t    target_range_last;
};
```

    - There are four fields in the packet
        - dst = packet destination address
        - src = source address
        - target_range_first = first range of address to infect
        - target_range_last = last range of address to infects
    - Instead of transmitting each target address of the hit-list, we decided to send the range representing the list. So, a single packet can infect and propagate the target addresses. The

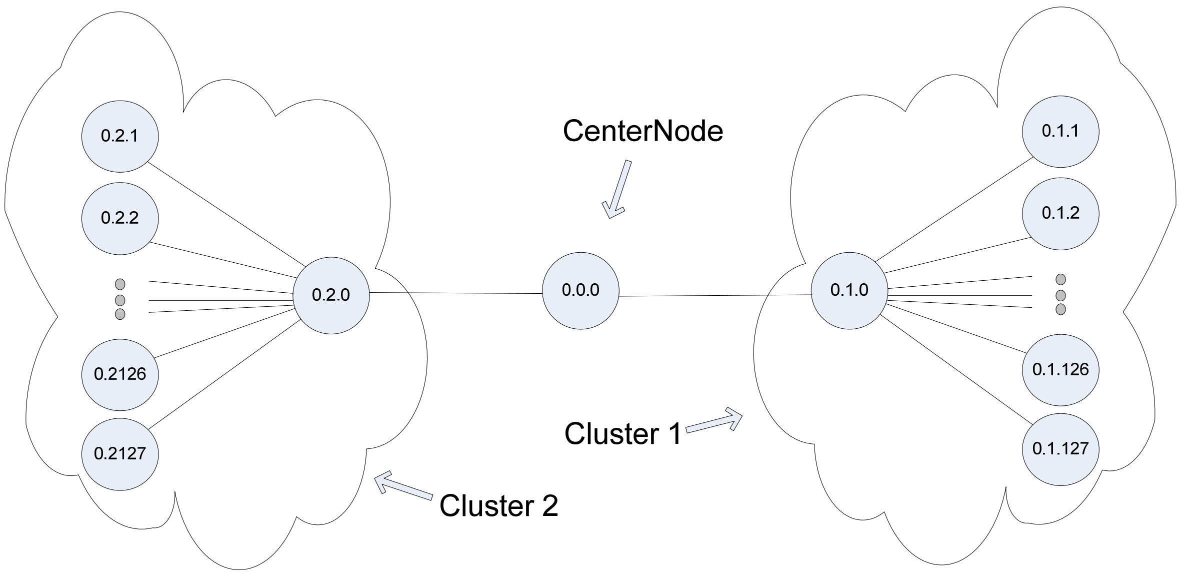
propagation delay to transmit the target addresses is simulated by forcing infected host to wait for the estimated delay before infecting other hosts. The delay varies with respect to the range size.

3.1.2.3. Added two new network entities: *Node* and *CenterNode*

- An entity *Node* represents a host that is vulnerable to flash worm infections.
- The entity *CenterNode* is a special node that is not vulnerable to flash worm infections. It is located at the center of the network hierarchy, and it acts as a router to forward flash worm packets amongst *Node*s. Every flash worm data has to go through this entity first. Since it can see all the flash worm traffics, it is also the ideal place to implement our detection mechanism.

3.1.2.4. Network Hierarchy Example

Fig. 6 displays an example of simulated network topology in NS-2.




<Fig. 6: Network Topology of NS-2, consisting of two clusters>

- There is always one *CenterNode* in the topology, which has the address 0.0.0 and is located at the center.
- Every packet is sent to CenterNode first. Then, the CenterNode forwards the packet to the correct destination.
- Each cluster can have maximum of 128 nodes.
- Of the address form 0.X.Y, ***X*** represents the cluster number and ***Y*** represents the node ID within the cluster.
- A node with the address form of 0.X.0 is a head of the cluster.
- Both the CenterNode and the head of a cluster have enough buffer size to handle heavy network traffics.

#### 3.1.3. Configurable Parameters

Following are the configurable parameters to simulate various flash worm scenarios:

- Number of clusters
  - The simulator can support up to 100 clusters. Since each cluster has 128 nodes, this means that the simulator can simulate 12,800 nodes.
  - Our simulation has been performed on a computer with 1GB of RAM. A computer with higher RAM space should be able to simulate more than 100 clusters.
- Infection number
  - This is the maximum number that each host tries to infect other hosts.
- Link delay
  - There is a network propagation delay between the CenterNode and the head of a cluster. Otherwise specified, we use *5ms* as the default delay.
  - There is also a network propagation delay between nodes and the head of the nodes. Otherwise specified, we use *10ms* as the default delay.
  - If using the default delay values, the end-to-end propagation delay between 2 hosts is 10 + 5 + 5 + 10 = 30 ms.

## 3.2. Flash Worm Detection Mechanism

Since flash worms spread so fast, the traditional signature-based defenses or manual human intervention cannot respond in enough time to stop the flash worm propagation. Therefore, much effort has gone into designing automated flash worm containment mechanisms. Besides the one we are proposing in this paper, there are other defense mechanisms against flash worms.

### 3.2.1. Other defense mechanisms

#### 3.2.1.1. Random Network Address

[1] proposes a defense system against flash worm in which the address of the Internet hosts are changed frequently. By doing so, the goal is to make the hit-list, which is pre-generated by the flash worm authors, unreliable and useless. However, the mechanism requires a significant modification of the existing Internet infrastructure.

#### 3.2.1.2. Honeypot

Traditionally, a honeypot is a fake network environment where various viruses and worms can be lured into a trap. If the address of a honeypot is in the hit-list, attempting to infect the honeypot may prevent a part of the spread tree to be infected. However, as discussed in the robust spread tree topology section, there are ways to get around this trap.

### 3.2.2. Geometry-based Detection of Flash Worms

#### 3.2.2.1. Basic Idea

The basic idea is to create a spread tree by monitoring network traffic activities, and if a predefined threshold level of depth is reached, shutdown the network. The spread tree is pruned periodically by removing the old nodes so that the tree would not grow continuously and trigger false alarms. By carefully choosing the threshold depth and the prune period, it should be difficult to trigger the false alarm by non-flash worm packets.

### 3.2.3. Extending NS2 to simulate detection of flash worms

#### 3.2.3.1. Node Structure

In order to create the spread tree, we use the following tree node structure:

- ```
  Struct node {
        int nodeID;       // ID (or address) of this node
        int depth;        // depth (or generation) number
        double timestamp; // node creation time

        node ** children; // list of children
        node ** parent;   // list of parents
        int numChildren;  // number of children
        int numParent;    // number of parents
        int maxChildren;  // max number of children
        int maxParent;    // max number of children
  ```
  ```

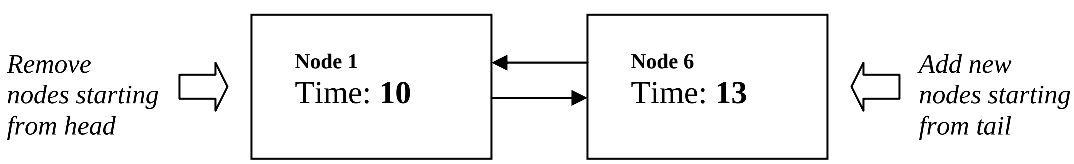
```
        node * before;     // for list sorted by time
        node * after;      // for list sorted by time
};
```

- Note that our node implementation can support multiple parents. This late added feature required more complex computation to be performed dynamically. For example, it required the support of parent-child loop avoidance and recursive depth determination which are discussed below. If the node were to support only one parent, the implementation could have been much simpler.

3.2.3.2. Double linked-list sorted by timestamp

To minimize the cost of updating the tree structure, we used the following double linked-list sorted by timestamp:

- Linked-list




- By using this linked-list, there is no need to search for old nodes in the pruning process.

3.2.3.3. Array of pointers to node sorted by Node ID

To allow us to easily access the node structure, we created an array of pointers to node structure that is sorted by Node ID.

- `Node * pNodes[size of max node];`
    - For example, given the a nodeID *X*, we could access the node structure by `*pNodes[X]`

3.2.3.4. Dynamic determination of root nodes

Through out the operation, there can be many splits or merges of spread trees. For example, a tree is split if the root node is removed; two trees are merged if one node infects one node on the other tree. Since it would be very cumbersome to maintain a list of all the root nodes, we decided to maintain no explicit list of root nodes. Instead, if the parent number of a node is 0, the node is the root.

3.2.3.5. Dynamic determination of depth

Instead of updating the depth of every child node when the depth of a parent node is changed, depth is determined recursively as following:

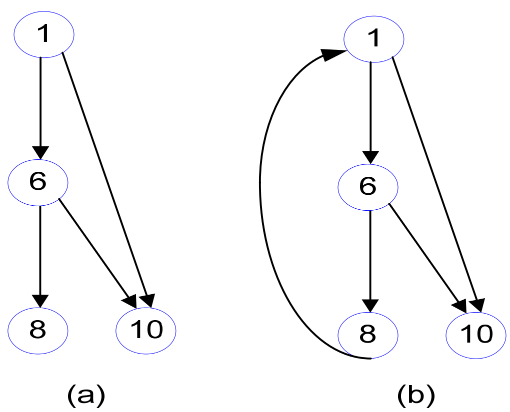
- depth = maximum depth of parents + 1

3.2.3.6. Exiting Parent-Child relationship

If there is an existing parent-child relationship between two nodes when creating a new relationship, we decided to do nothing. Alternatively, we could have updated the *node creation time* so that the node would be pruned at a later time. However, we thought doing so would consume significant computation power to reorganize the linked-list. For example, if two nodes are communicating using a TCP connection or running some application, there would be many rapid requests to update the *node creation time* in a short period of time. Therefore, we decided that it would be better not to update the node creation time.

3.2.3.7. Parent-Child Loop Avoidance

Before a new parent-child relationship is created, a check is performed to prevent the formation of a parent-child loop. See Fig. 7 below.




<Fig. 7: Loop Avoidance>
(a) No loop; (b) There is a loop, connecting node 1,6, 8

In Fig 7(a), there is no loop. However, in Fig 7(b) there is a loop among the nodes 1, 6, and 8. If there is a loop, we cannot determine the depth correctly since we cannot determine which node is the parent/root node. Therefore, if a new parent-child relation creates a loop, we remove that relationship right away. There can be potential attacks against this restriction, one of which we discuss later.

### 3.2.4. Configurable Parameters

Similar to the configurable parameters in flash worm simulation, there are configurable parameters for detection mechanism.

- <u>Depth Threshold</u>
    - This is the maximum number of depth allowed before an alarm is triggered to notify that spread of flash worms has detected.
    - Too lower number is likely to generate many false alarms; too high number would not detect flash worm early enough to react.

- <u>Prune Period</u>
    - This is the amount of time elapsed before next pruning occurs.

- <u>Prune Amount</u>
    - This is the amount of time we subtract from the current time, and any nodes created before that time is pruned. For example, if *Prune Amount* is 5 seconds and current time is 30 seconds, the nodes with creation time less than 30 – 5 = 25 seconds will be removed from the spread tree.

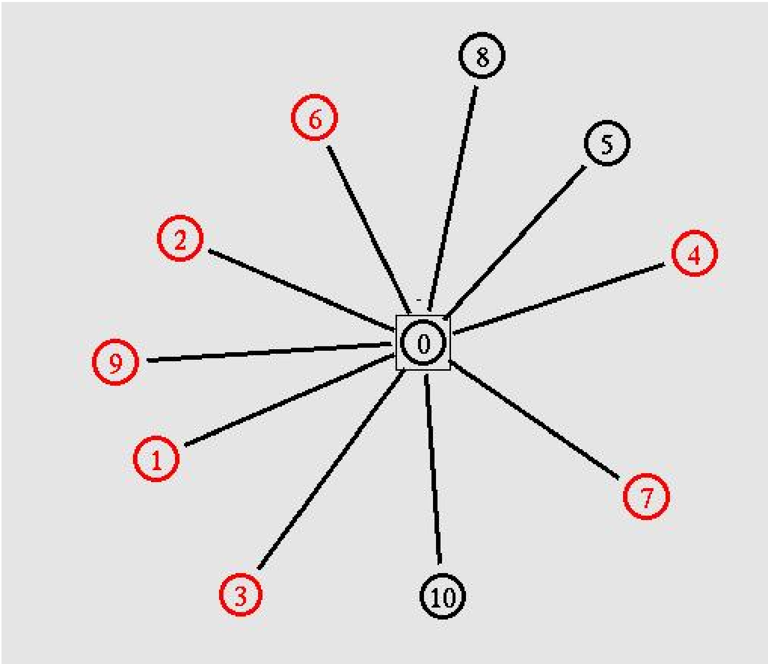
### 3.3. Post-analyzer

In this section we discuss how we implemented a post-analyzer to help visualizing the simulated result.

#### 3.3.1. Nam

NS-2 provides Nam, which is a graphical utility to display the network topology and simple network traffic activities. We have implemented Nam such that when a node is infected, the color of the node changes to red. Following is a sample output of running the simulation consisting of 10 nodes.

- nam result




<Fig. 8: Nam screen-shot>

- Node0 is the CentralNode
- Nodes 1,2,3,4,6,7,9 have been infected at the time
- Nodes 5,8,10 have not been infected yet.
- *This is a screen shot of a specific simulation time; eventually, every node will get infected.*
- Disadvantages:
  - If plotting a lot of nodes, Nam cannot position nodes evenly across the screen. Nam will display all the nodes close to the CenterNode and we will eventually see a big dark circle in the middle.
  - Nam cannot automatically order the nodes into a tree form.

### 3.3.2. Custom post-analyzer

To resolve all Nam's shortcomings, we have implemented a custom post-analyzer using MFC from Microsoft Visual Studio. Following are the features of the post-analyzer:

- Able to plot a vast number of nodes on a screen
- Able to zoom in (or out) to see detailed structure
- Able to see detailed information of a node when the mouse is placed over the node
  - The information is displayed in a tooltip

                - The information is consisted of *node address*, *node creation time,* and *depth*.  The tooltip can be easily modified to display more information.
    - Able to manually start, stop, and rerun the simulation at any time.
    - Able to run the detection mechanism
        - The detection mechanism is implemented in both NS-2 and the post-analyzer.  The detection mechanism in NS-2 can be used to monitor the flash worm packets as soon as they are generated by the simulator.  However, the configurable parameter for the detection mechanism such as *Prune Period* cannot be changed after the simulator is started and the testing of the detection mechanism always requires regeneration of flash worm traffic.  On the other hand using the post-analyzer, the configurable parameters of the detection mechanism can be changed and tested at any time.  The post-analyzer can also apply the detection mechanism on saved flash worm traffic so the flash worm traffic does not need to be regenerated.  Therefore, we prefer to use NS-2 only to generate the flash worm traffic and run the detection mechanism on the post-analyzer.

# 4. Tests and Results

In this section we explain our testing setup and provide an analysis on the result.

## 4.1. Generate flash worm traffic

Flash worm traffic was generated by running NS-2 simulator with following parameters:

- Configurable Parameters:
    - Number of clusters = 10
    - Infection number = 2

- We disabled the detection mechanism of the simulator. If the detection mechanism is disabled, NS-2 automatically outputs all the flash worm traffic data into a text file.

- Traffic Format:
    - The traffic is organized by the following fields:
        - *Event Time (in ms), Node Address, Parent Node Address, Target Address Size*
    - Each field is separated by a space.
    - For example, Fig. 9 displays the first five flash worm data.

```
line 1: 0.1 128 0 1279
line 2: 19.2 768 128 639
line 3: 19.2 129 128 638
line 4: 28.8 769 768 318
line 5: 28.8 1088 768 319
```

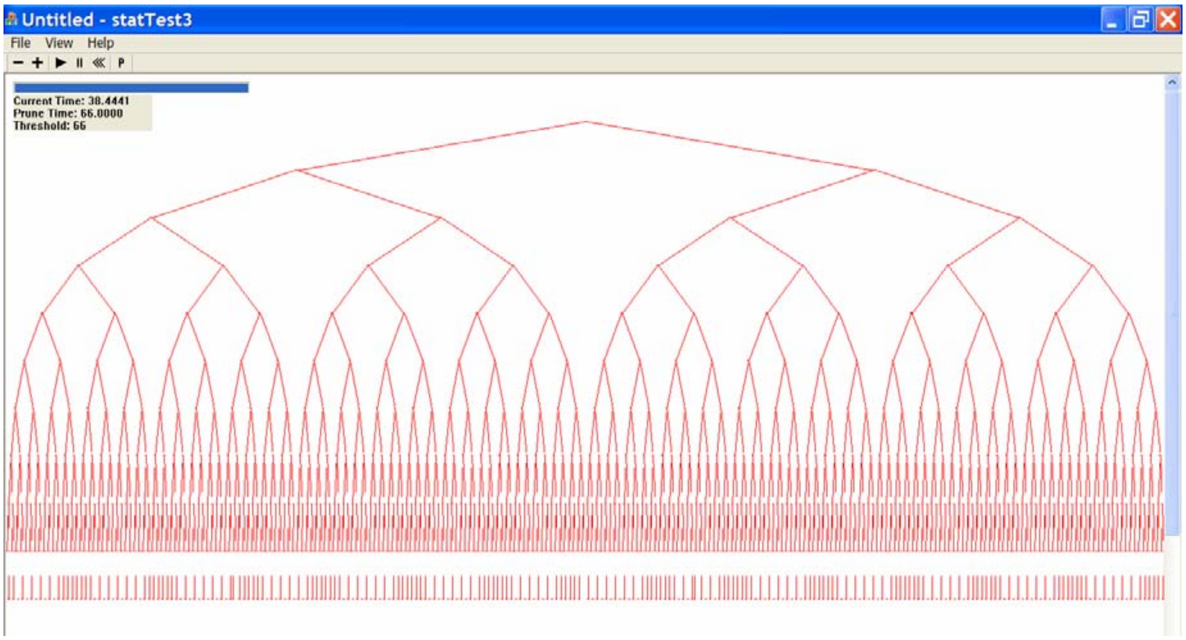
<Fig. 9: Sample Flash Worm Traffic>

- Line 1 has the initial flash worm data. The line "0.1 128 0 1279" means "*Node 128* was infected by *Node 0* at *time 0.1* and the *target address size is 1279*". Since we simulate 10 clusters and each cluster has 128 nodes, the initial hit-list size of 1279 is correct.
- Line 2 and 3 shows that Node 128 correctly infected 2 other nodes at 19.2 ms. Note that the target address list is divided into 639 and 638 and distributed to Nodes 768 and 129 respectively.
- Since our simulation does not utilize threads, the node has to wait until it receives all the target addresses from its parent. From line 1 and 2, we can see that it took 19.2 – 0.1 = 19.1 ms to transmit target list size of 639 from Node 128 to Node 768. Also, from line 2 and line 4, we can see that it took 28.8 – 19.2 = 9.6 ms to transmit target list size of 318 from Node 768 to Node 769. Therefore, a higher propagation delay is required to transmit a larger target list.

### 4.2. Plot and analyze flash worm traffic

After generating the flash worm traffic, we plotted and analyzed the traffic using our post-analyzer.

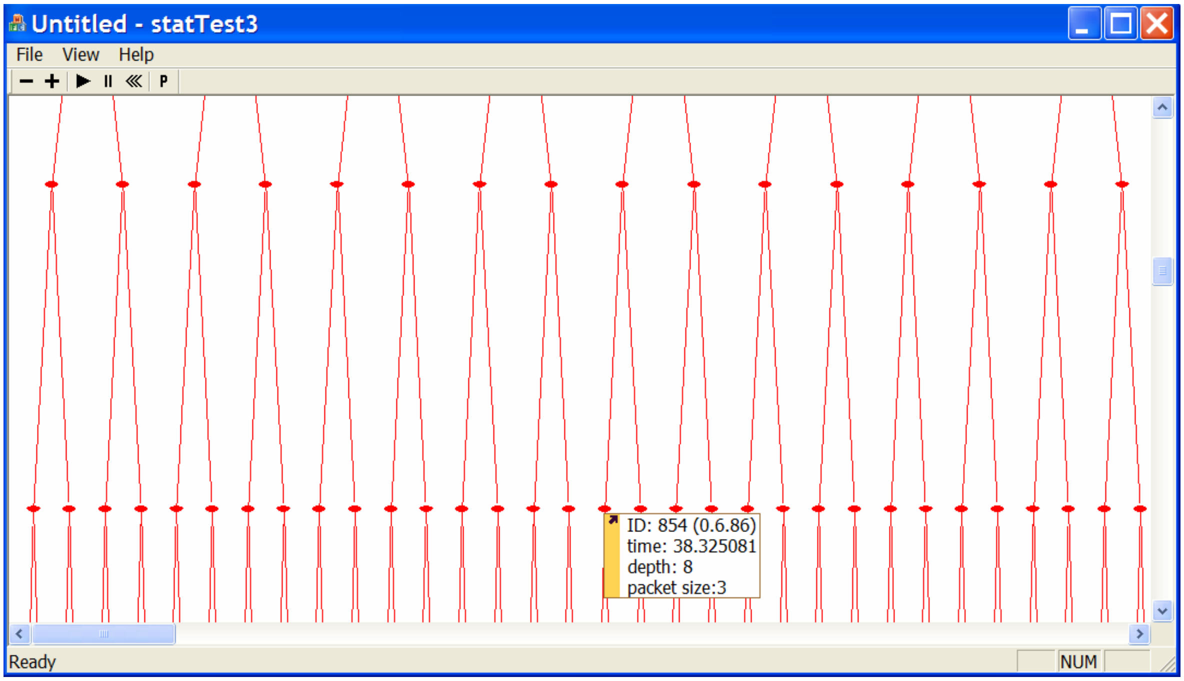
- Spread Tree Topology
  - Fig. 10 displays all the 1280 target nodes organized into a spread tree topology.




<Fig. 10: Complete Tree>
Without Pruning

  - In this test setup, there is no pruning because we specified sufficiently long pruning time period to display the complete tree structure.

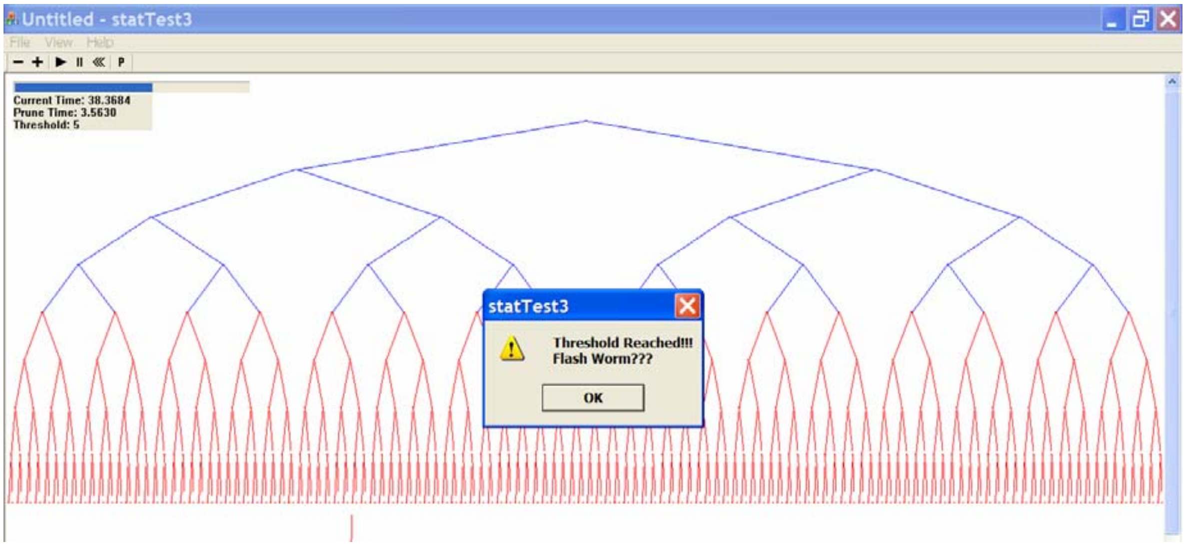
o Fig. 11 displays a zoomed-in screen shot. It also shows a tooltip, which describes the detailed information regarding Node 854.




<Fig. 11: Zoomed-in and ToolTip>

- Running detection method

  o Fig. 12 shows that our detection method found the spread of flash worms.




<Fig. 12: Detection Mechanism in Action>

- For the detection method parameters, Prune Time is set to 3.5630 and Threshold Depth is set to 5.
- The nodes and lines marked in blue color indicate the pruned entities. In the above example, the first 3 generations have been pruned.

### 4.3. Plot and analyze the actual internet traffic captured from Lawrence National Laboratory

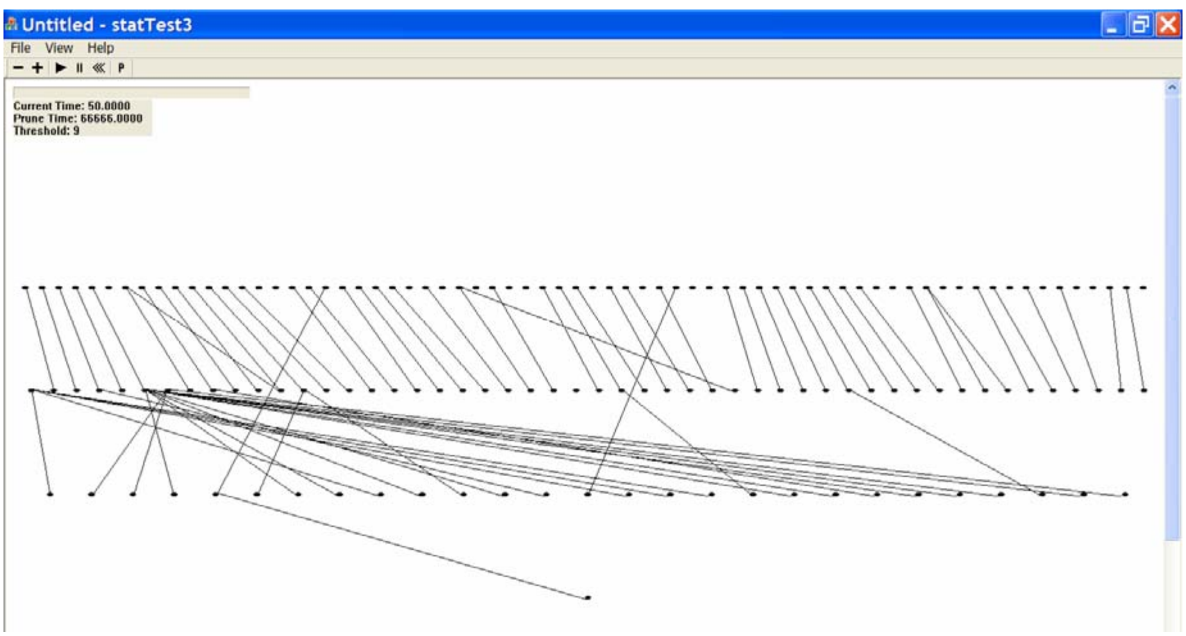
- In the previous experiment, we tested that our detection mechanism can successfully detect the spread of flash worms. However, we were also interested to find out how the detection mechanism would respond to other kinds of network traffics.

- Instead of generating various Internet traffics as we generated the flash worm traffic, we decided to use an archived Internet trace data from [9]. Captured from the Lawrence National Laboratory using *tcpdump* of real WAN traffic, the trace data went through a sanitization script to renumber IP addresses and to remove all private packet contents. The sanitization script also separated TCP traffic from UDP traffic and places them into different files. One can capture one's own Internet traffic using *tcpdump* and use the same sanitization script to produce the similar trace data.
- Traffic Format:
  - The traffic is organized by the following fields:
    - *Timestamp of packet arrival, Source host, Destination host, Source port, Destination port, Packet size*
  - We modified our post-analyzer to use above fields to create spread trees.
  - We did not utilize the port numbers to create different spread trees for different port number. However, in a real world deployment of our detection mechanism, the spread trees should be further separated by the port number because flash worms will probably use a specific port number to spread.

- TCP Traffic
  - The spread tree formulation of the TCP traffic is shown in Fig. 13. Even though more than 150 packet events were processed, there were many

attempts to create loops or to create repetitive relationships. The attempts to create loop were avoided and the attempts to recreate existing parent-child relationship were ignored. Therefore, we did not see rapid growth of the spread tree.

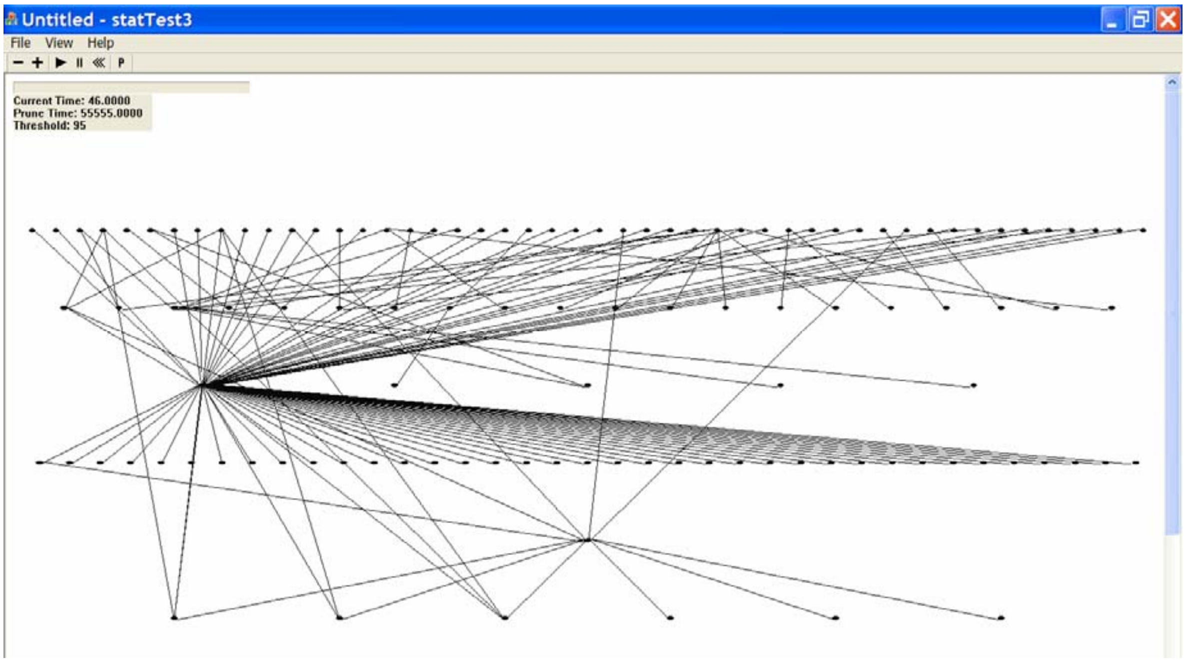
- The tree structure seemed to be the reserved formulation of the flash worm spread tree. The total number of nodes in a generation decreases as generation number increases. However, the exact opposite happens in flash worm spread tree.




<Fig. 13: Spread Tree of TCP traffic>

- UDP Traffic
  - In TCP traffic, many network communications were between 2 nodes. However, in UDP traffic, we saw more interactions among different nodes. See Fig. 14.

- o This time we processed about 300 packet events, but still it only created 5 generations.




<Fig. 14: Spread Tree of UDP traffic >

### 4.4. Breaking our detection mechanism

From the previous section, we saw that typical Internet traffic is not likely to trigger false detection alarms. However, there are ways to break our detection mechanism:

- Slow down the infection process
    - o Because nodes are pruned periodically, slowing down the infection process with respect to the pruning period, the depth of the spread tree may not grow rapidly.

- Use customized source and destination addresses
    - Because our current implementation does not allow loops in the tree, customized packets can be transmitted in a way that makes the valid flash worm packets to be ignored. For example, before sending a packet to infect a node from next generation, source and destination address fields can be switched to create a reverse parent-child relation in the tree first. When the actual flash worm packet is processed later, it will attempt to create a loop and will be removed.

## 5. Conclusion

In this project, we developed a simulator to generate flash worm traffic. We, then, developed an analyzer that can run our proposed geometric-based detection mechanism on the traffic. Our results show that the detection mechanism can be sensitive to the presence of the flash worm traffic while maintaining insensitivity to other regular internet traffics. Working closely with the packet classification products, we believe that our detection mechanism can successfully identify the spread of flash worms.

## 6. Future Work

First of all, to increase the maximum number of nodes simulated, a different version of NS simulator can be used. PDNS, for example, is a network simulator that can support parallel and distributed simulation over multiple PCs. The need to simulate larger network topology may arise when testing our detection mechanism on a shallow spread tree topology. Our current maximum of 10,000 nodes is not sufficient to create flash worm traffics with a high infection number. For example, if each host infects 10 other hosts, it only takes 4 generations to infect all 10,000 nodes.

Secondly, to reduce the occurrence of false alarms, more components can be used to identify the unique characteristic of the flash worm spread tree. In our current construction of spread tree, we only use depth as the determining factor to decide whether an outbreak of flash worm has occurred. However, studies can be conducted to find other factors. For example, there is a pattern that the total number of nodes in a generation increases from a generation to the next. It follows the form $X^G$, where X is the maximum infection number and G is the generation number. Such pattern may be use to reduce false alarms.

Also, in another effort to reduce the false alarms, research can be done on integrating the packet classification technology into our detection mechanism. The ultimate goal is to help identify the well known application traffic patterns so that the known “safe” network traffic would not be processed into a spread tree. As an example, we suspect that peer-to-peer traffic is likely to trigger false alarms because of the many parent-child relationships amongst a vast number of peers. If the packet classification technology can identify a peer-to-peer traffic, then the traffic would not be processed by our detection mechanism, which would reduce the chance of incorrect detections.

## Bibliography

[1] S. Antonatos, P. Akritidis, E. P. Markatos, K. G. Anagnostakis, “Defending against Hitlist Worms using Network Address Space Randomization.”

[2] Yuy Bulygin, “A spread model of flash worms”, http://isiom.wssrl.org/index.php?option=com_docman&task=doc_download&gid=14

[3] Cheetancheri. "Modelling a Computer Worm Defense System." http://seclab.cs.ucdavis.edu/papers/Cheetancheri-thesis.ps

[4] Ellis, Aiken, Attwood, Tenaglia. "A Behavioral Approach to Worm Detection." http://www.wormblog.com/2005/03/a_behavioral_ap.html

[5] Goldi, Hiestand. "Scan Detection Based Identification of Worm-Infected Hosts." http://www.tik.ee.ethz.ch/~ddosvax/sada/ScanDetection_ma_ws2004_2005.pdf

[6] Moore, Paxson, Savage, Shannon, Staniford, Weaver. “The Spread of the Sapphire/Slammer Worm.” http://www.cs.berkeley.edu/~nweaver/sapphire/

[7] S. Staniford, V. Paxson, and N. Weaver, “How to Own the Internet in Your Spare Time,” in *Proceedings of the 11th USENIX Security Symposium (Security’02)*, San Francisco, CA, pp. 149-167, Aug. 2002, http://www.icir.org/vern/papers/cdc-usenix-sec02/

[8] S. Staniford, V. Paxson, and N. Weaver, “Top Speed of Flash Worms,” in *Proceedings of the 2nd ACM Workshop on Rapid Malcode (WORM)*, 2004, http://www.caida.org/outreach/papers/2004/topspeedworms/topspeed-worm04.pdf

[9] The Internet Traffic Archive, http://ita.ee.lbl.gov/

[10] Vogt. "Simulating and Optimising worm propagation algorithms."

http://web.lemuria.org/security/WormPropagation.pdf

[11] Weaver, Staniford, Paxson. "Very Fast Containment of Scanning Worms."

http://www.icsi.berkeley.edu/~nweaver/containment/containment.pdf

[12] Worm Blog by J. Nazario, http://www.wormblog.com

[13] Wu, Vangala, Gao. "An Effective Architecture and Algorithm for Detecting Worms with Various Scan Techniques." http://www-unix.ecs.umass.edu/~lgao/ndss04.pdf