Towards a framework for worm-defense evaluation

Senthilkumar G Cheetancheri* Denys L Ma Karl N Levitt Todd L Heberlein
Security Lab, Department of Computer Science NetSquared Inc.
University of California, Davis Davis

ABSTRACT

Computer worms are a serious problem. Much research has been done to detect and contain worms. One major deficiency in most research is that the claims are supported by theoretic models or simulations only and not by realistic tests. Network testbeds such as emulab and deter can be used to conduct worm experiments on networks of a few hundreds nodes. However, setting up such an experiment is not trivial. In this paper, we describe a wrapper around emulab to deploy such experiments quickly. We also demonstrate its use by evaluating an example worm containment strategy.

General Terms

Worm defense, Testbeds, evaluation

Keywords

EMULAB, DETER, worm defense

1. INTRODUCTION

Computer worms are a serious problem, and it is likely that future worms will carry a lethal payload and will be more difficult to detect; for example, future worms are likely to be stealthy and could be polymorphic. Much research has been carried out in recent years to detect and respond to worms, where the goal is to produce a defense that in real time can detect a worm and quarantine sites not yet infected[11, 20]. One major deficiency in most of the research is that the claims are not supported by realistic tests. Most claims are supported by theoretic models or simulations only.

One reason for this lack of realistic testing is the impossibility of testing a worm defense on the real Internet, clearly the playground for worm writers, or even on a local network. A promising, and maybe the only one that is viable, approach is to test a worm defense on a testbed. One major difficulty with this approach is that a large number of test machines have to be configured and managed efficiently. Also care should be taken that the malcode used for testing doesn't leak into the real Internet. These are challenging tasks. Given the difficulty of reproducing live environments for worm-defense research, most researchers resort to simulations. At the outset they might seem sufficient but they have several limitations. For example, the effects of background traffic, traffic load on network components such as routers, bandwidth saturation, the timing idiosyncrasies due to the network stack, the diversity of the hardware in the real Internet,etc. – all issues that impact the operation of a worm and a worm defense system – are not reflected in simulations.

Weaver et al.[19] highlight the stochastic effects on the results of worm experiments when scaled-down networks are used. These effects can be overcome by repeating each experiment numerous times and averaging the relevant values. Similar work has yet to be carried out for a worm defense system, but the problem is likely to be more difficult than for experiments just involving worm studies.

Thus there is a great need for a way to faithfully reproduce live environments for worm- and worm-defense research. In this paper we make use of a network testbed called EMULAB[21] to satisfy this need and evaluate our framework for worm-defense evaluation within enterprise networks, i.e., networks with a few tens to few thousands of nodes. We describe an implementation of the framework and use it to evaluate an example defense strategy, but emphasize that the framework can support many different defense strategies. The framework is encapsulated in an API. This API accepts a topology description and a description of the defense system, and evaluates the defense system against worms. The worms can be characterized by a specification or operationally by a worm program.

The next section provides the motivation for this API and section 3 analyses some of the previous and relevant work concerned with worms and worm defense systems. Section5 shows how a defense strategy previously developed by the author in [10] can be evaluated using our framework; previously, this worm defense system was evaluated using a simulation, and this present work confirms the results of the simulation but in a realistic setting. Finally, section 6 shows future directions to pursue.

2. MOTIVATION

EMULAB[21] and DETER[1] are network testbeds that

*e-mail:First 8 letter of last name@cs.ucdavis.edu

Malware'06 April 10-12, 2006, Phoenix, Arizona USA

can be used for network security research offering a low cost option to operational testing. They provide hundreds of end host systems with various popular operating systems that can be bought up in a matter of minutes, saving both equipment and maintenance expenses. Virtual nodes are also supported on each physical node, thereby multiplying the effective number of nodes that can be used for our experimentation. Nodes can be remotely controlled in terms of the OS each one loads and the way they are interconnected. These capabilities and their similarity to the typical size of real-world enterprise networks make them a perfect theater for worm-in-enterprise research.

However, a large scale worm experiment is very difficult to setup. It typically takes a new user only a few hours to run the first "Hello World" experiment but several weeks to run the first worm-defense experiment. Due to the results of Weaver et al's[19] results, we need to repeat the experiments numerous times times to get credible results. However, while working on [17], we discovered that the set-up time for each experiment is significantly higher than the experiment duration itself. It usually takes 10–15 minutes to setup an experiment that ran for 2–3 minutes, depending on the size of the topology. Also, worm experiments require a large number of nodes that are not always available on the testbed. Hence, setting up the testbed for such numerous experiments manually becomes infeasible. We needed a way to automatically setup the testbed and perform experiments in batches.

To facilitate this, the testbed offers features such synchronization servers, program objects and group event control systems. However, it requires very careful programming of these sub-systems to repeatedly reproduce test environments. During our efforts to evaluate *The Hierarchical Model of Worm Defense*[10], we had developed several programs and scripts to automate these processes. Also, experience shows that the event system set-up doesn't differ much from one experiment to another. Hence, we reasoned that we could package and parameterize these scripts to be used by other users through a simple interface. Thus, taking the testbed one step closer to the community.

Nevertheless, using EMULAB, people can evaluate their worm defenses without using this API, but it is a very exacting task. The other, easy, end of the spectrum would be a command line or point and click tool. This tool would have a set of pre-programmed defense schemes that can be executed with a few pre-determined parameters to evaluate which scheme is best for their enterprise. However, this would not be as flexible as using EMULAB directly. Hence, we try to find a sweet spot in between these two extremes that would make life of researchers easy as well as provide them a framework with enough flexibility to tweak and tune their schemes.

3. RELATED WORK

Most of the research done on worm defense and quarantine strategies has relied on simulation to validate the algorithms[8, 20, 7, 11]. Simulations, however, cannot capture insights related to systems variability, network characteristics, worm behaviors, and other operational details that it abstracts. There are efforts to capture these characteristics in simulation efforts by Liljenstam et al[13] using the SSFNet[6] simulation tool. This is better because SSFNet tries to simulate the network stack behavior also. But in general, all these simulations are based on formulated models and cannot fully represent some of the more difficult network and mal-ware behaviors. For example, it is generally very difficult to simulate "smart" worms that exploit various network evasion techniques[16, 9].

Using emulation on testbeds such as EMULAB, on the other hand, fully captures the heterogeneity of the network and worm characteristics that simulation cannot accurately measure. Lippmann et al[14] developed a testbed in order to accurately model a government enterprise network and evaluate real intrusion detection systems. There are projects that have used EMULAB and DETER but unfortunately, they have not used these infrastructures effectively. Weaver et al[20] use DETER but as a parallel processing environment to run their simulation quickly rather than as an emulator. Wei et al [17] use EMULAB, but as a distributed simulation environment.

An ongoing effort at the EMIST[3] project provides various tools addressing testbed issues. Penn State University's EMIST ESVT[2] provides a GUI package for topology creator and generator, traffic and experiment interfaces and visualization tools. ESVT does not provide experiment synchronization and automation. EMIST Tool Suite from Purdue University, on the other hand, provides a Scriptable Event System(SES)[4] for synchronization and automation for individual nodes in the experiment. The EMIST Tool Suite, however, does not provide any tools for topology utilities and worm specific tools. Finally, both tools do not support real applications such as IDS and firewalls that are crucial to worm experiments. They also do not provide any methods to integrate additional components, such as real worm codes, real defense strategies, and live background traffic.

4. THE API

This API takes in 3 parameters, a network topology in NS format, a defense program and a worm. It returns a thorough analysis of the proposed defense strategy based on various parameters. Some these parameters are the total number of nodes infected, the time taken to stop the worm from spreading, the effects on the network such as latency, bandwidth occupied by the defense vs the worm vs the normal traffic, the effects of false alarms on the normal operational efficiency and the recovery time. Recovery time is the time taken for the network to return from a defensive posture during worm attack to its normal state of operation. This is a very important factor in real-world networks because there is a cost involved when the network is not operating in its usual fashion.

Figure 1 gives a white-box design diagram for this API. At the outset, we can see that user specifies the above mentioned 3 parameters apart from the ability to play background traffic using some third party tool. The following sub-sections describe the user-inputs required and the various components of the framework.

4.1 User Inputs

This sub-section describes the various user parameters and their specifications. These are the topology specification, the worm parameters and the user's defense program and optionally a background traffic generator.

4.1.1 The topology specification

The user specifies the topology of the test network as an NS2[5] file. This would represent the enterprise network of the user. The NS2 language was chosen because it lets us specify exactly various network parameters like the network bandwidth, latency, etc., and also allows for traffic shaping information. This topology information should also include the location of various servers, gateways, routers, firewall, IDSes, etc., in the enterprise.

One can reconfigure the interconnection of the experiment nodes in EMULAB by feeding it a script written in NS-testbed, an extension of the NS language. This extension contains several commands that are specific to the testbed. These commands control the event groups and program object sub-systems, the routing protocol used, the synchronization sub-system, etc.,. The user's topology is transformed into NS-testbed by the compiler in our framework. This compiler will be discussed in the next sub-section.

4.1.2 The defense program

This API provides a hook on which to hang the users' defense program. Since users' home directories get automounted on all experiment nodes, there is no need for any special installation procedures for these programs. It is sufficient if they are available in the user's execution path and the user just has to provide the program name. Our compiler picks up this defense program and inserts it to the NS-testbed script thus registering it with the testbed.

By providing such a hook, we allow for maximum flexibility for the user to implement their own defense and response mechanisms. The programs could be be anything from worm detection algorithms using correlation, decision trees, Bayes Net techniques to automatic signature generation to response mechanism using firewalls, IP black-listing, or any other novel technology that the users' want to evaluate. This is the parameter of the API for which we expect the user to spend the most effort and rightly so because this is the program we are evaluating.

Since the defense program is already included in the *defense event group* by our compiler, it would be called at the appropriate time from the *Event Control System* of our tool. Ideally, we expect the user to design their defense program as a server that responds to events, typically events that are symptoms of worm activity. Hence, the most propitious time to start up these defense programs would be at the beginning of the experiment.

If the defense programs use any tools or programs that are not available by default on the experiment nodes, these have to be installed manually and a disk image made prior to starting the experiment. Then this image can be specified in the NS-testbed file to be loaded on to the experiment nodes.

4.1.3 Worms Parameters

Our API by default provides a very parameterized worm generator. The parameters are as follows:

1. **Type of connection:** UDP or TCP.
2. **Speed of worm:** The number of scans per second.
3. **Scanning method:** Random or subnet scanning. If subnet-scanning is specified the user could also specify the Percentage of Out-of-Domain Scanning, *pods*, desired. If no *pods* is specified we experiment with *pods* from 10 to 100 with step size of 10. In fact, at 100 *pods*, the worm becomes a random scanning worm.
4. **A payload to the worm:** This could just be a random text. This is only to analyze the effect of payload size on the worm dynamics and network bandwidth rather than anything else. If it is a malicious function to be executed on the testbed, the user should also provide the vulnerable servers along with the topology specification. However, we discourage this, as this worm can get out of control and doesn't add any value to the experiments.

The users can choose one or more worms and parameterize them to be deployed against their defenses. Alternatively, the API also provides hooks to hang the users own scanning method for the worm. For example, a hit-list scanner, or a real Code-Red, Slammer, etc.,. These user scan functions need to be added into the framework's worm library beforehand. Then the API should be instructed to use this users' worm function while choosing the 'Scanning method' mentioned above.

After all, the users can just ignore all of these and provide their own worm programs and corresponding dummies.

4.1.4 Background Traffic Generator

The users can replay their normal enterprise traffic in the background on EMULAB while testing their defense. The background traffic can be played using tools like TCP-Opera[12] or NTGC[18] depending on whether the users want the traffic to be source or trace parameterized. This may also depend on the defense strategies. If the defense contains signature matching, the user may want to replay raw traces.

4.2 The API components

This sub-section describes the components of the framework. Figure 1 shows the interconnections between these components within the broken line box. The NS to NS-testbed compiler generates user defined topologies for the testbed. After proper topology configurations, the Dummy Vulnerable Server and Event Control System integrate user-supplied defenses and worms and conduct experiments for certain number of iterations predetermined by the user. The Data Analysis Tools collects various data about the experiments and generates evaluation statistics. These modules are transparent to the users, creating an appliance approach to worm defense experiments.

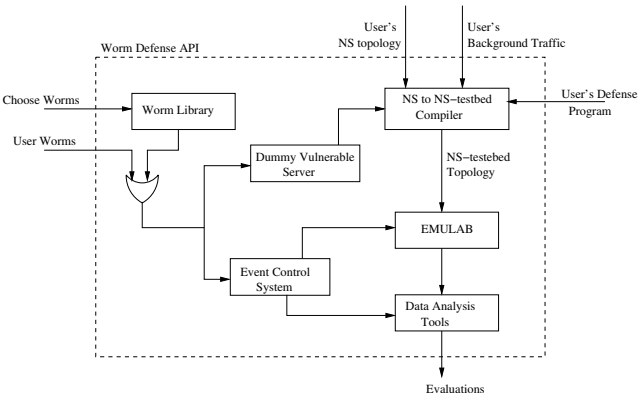

Figure 1: Design of the Worm-Defense Evaluator.

4.2.1 NS to NS-testbed compiler

The NS to NS-testbed compiler in the API, takes the user's NS file and compiles it to format suitable for the testbed. Apart from the usual tasks of specifying the OS to load, the routing protocol and assigning IP addresses to the nodes, the compiler should do the following two important tasks.

First, set up a synchronization mechanism for the experiment. This can be done by using specifying a node as the synchronization server and using the testbed's `sync_server` tool. This is required so that we can make use of the batch processing feature of the testbed. As mentioned in section 2 batch processing is the only practical way of running large experiments.

Second, set up 'Program Objects' and 'Event Groups' appropriately. The users' defense programs and the 'dummy vulnerable servers', called dummies, need to be inserted into the appropriate event groups, such as, *defense event group* and *vulnerable event group* respectively. This grouping is required so that the dummies and the defense programs can be restarted from a single `tevc` command in the 'Event Control System'. This helps us to bring the testbed to a clean state instantaneously without swapping out and swapping in experiments which takes about 10–15 minutes. A clean state of the testbed is the state when all experiment nodes are just booted up and no user processes are running, and all changes made to the routing tables, firewall rules, IDS signatures, etc., during the last run of an experiment are erased. Such a state is required for each run of an experiment. A typical worm experiment series has about 1000–1500 experiments lasting 2 minutes each on an average. This step helps to finish a series in about 36 hours that would otherwise take us about 12 days. That is a huge saving.

4.2.2 Dummy Vulnerable Servers

The dummies listen for traffic on a certain port. Once they are receive a packet of a specified type, a worm packet, they mark themselves as infected, save a time-stamp of the infection and spawn off a worm in their own node. This relieves us of the task of writing an exploit for the worm. By deploying our own dummies for vulnerable servers, we are also able to make use of them as data acquisition tool.

This doesn't compromise the experiment in any way. Dummies are a valid abstraction of vulnerable servers because we don't know how the real servers would be attacked. Even if we write our own complex exploit for a real server, it will not reflect reality as a real worm's exploit is probabilistically bound to be very different than ours. Rather, we are more interested in the worm dynamics and ways to mitigate the repercussions and dummies capture that effectively. We also note that, the dummies don't take much time to spawn off a worm in their own node. This is not very different from real exploits which spawn off a worms in a victim machine rather instantaneously.

As a pleasant side-effect, we end up with safe worms; worms that cannot spread out on the Internet where *our own dummies* are not installed.

4.2.3 The worm library

Our framework has a very flexible built-in worm generator. This takes in several parameters as already mention in section 4.1.3. Our worm generator can generate several families of worms, such as, random or subnet scanning worms based on TCP or UDP protocol, at various speeds and with different payloads.

It is a simple program running in a tight loop. It sleeps for $1/scanrate$ seconds, wakes up and sends a TCP/UDP message to the *dummy server* on a victim chosen according to its scanning method. In our implementation we use function pointers to scanning methods. This provides us the flexibility to easily extend our library as well as use any users' scanning methods.

4.2.4 The Event Control System

The Event Control System(ECS) runs on the sync-server of the experiment. It controls the start and stop of the experiment run, triggering the data analysis tools and rotating the log files.

This is a script that was hand-coded originally to help in evaluating our earlier defense models. Now, we have parameterized this script so that others can also use it. The parameters to the ECS are the worms that need to be launched, and the names of the event groups generated by the compiler. These values are passed on to this component internally transparent to the user.

When an experiment needs to be run, ECS starts all the program *event groups*. Given the worm's characteristics, the ECS bootstraps a worm on one of the dummies, thereby creating patient 0. This starts the worm outbreak. The ECS keeps track of the progress of the experiment by counting the time-stamps of the infections from the dummies. Since the same home directory is mounted on all experiment nodes all dummies write to the same directory and hence counting the time-stamps becomes easy as does the final data processing. Once the worm count reaches a stable value the ECS deems that the experiment run is complete. The collected data is stored in a retrievable fashion. The *event groups* are restarted and the next worm is launched. This is repeated for several pre-determined iterations. Once the entire experiment series is complete data analysis programs are run on the collected data to give us the evaluations.

4.3 Data Analysis Tools

The dummies write all data on the user's home directory. Current implementation of the dummies collect infection time and alert time, the time when defensive responses kick in. Currently, we have tools that chart the infection trace, the total number of nodes infected during each experiment run and the time taken to stop the worm from spreading. The users are welcome to do their own analysis of the data. In case, the user needs more data, the defense programs need to be programmed appropriately.

The next section presents an implementation of the proposed framework and evaluates an example defense model.

5. AN EXAMPLE - THE HIERARCHICAL MODEL OF WORM DEFENSE

To demonstrate the effectiveness of this tool, we consider a worm-defense model called, *The Hierarchical Model of worm defense* that was developed in [10].

Briefly, this model assumes that all participating nodes are arranged in a tree structure. The leaves are vulnerable but run some sort of an IDS system to detect attacks and and have some tunable firewall capabilities. The non-leaves are invulnerable to attacks and run the worm-defense programs. Once a leaf detects an attack, it send a message to

its parent. In essence, this message would contain the suspicious packets. Once the parent receives a threshold number of messages from unique children it takes two actions. One, it instructs all its children to turn on responses to this attack. Two, it sends a message to its own parent about the infection. Needless to say, for each non-leaf its threshold should be lesser than its number of children to get any benefit from this scheme. Thus the information about the attack travels up the tree and the instructions to respond percolates down the tree. Intuitively, the lower the threshold, better the defense.

5.1 Modeling the system

This sub-section models the example defense strategy to be fed to the API.

5.1.1 The topology specification

This model reflects a real enterprise network as closely as possible. The root node would be entry point to the enterprise. The leaves would be end-nodes, users machines and servers that are vulnerable to attacks. The non-leaf nodes are routers or gateways to individual departments inside the enterprise.

Our experiments contained 4 levels in the hierarchy, representing the UCDavis's College of Engineering network. Going down from the root to the leaves, each level represents, Gateway for the College of Engg., the departmental gateways, research lab routers and then finally the individual machines in that order.

This model consists of dummies at the leaves running host based firewalls and IPSs that can be tuned upon receiving instructions from their parents. All the non-leaves run the defense program, lets call them controllers, in the sense they control the defense. We also assume that they are invulnerable to attacks.

This model is so simple that it can be represented by just the number of levels in the hierarchy, the number of children and threshold of the nodes in each level of the hierarchy. We hand-rolled a program that would read this specification and give us a NS-testbed script. However, when we finally release the full implementation of this API, this program would be a more versatile compiler to handle NS scripts.

5.1.2 The defense program

The defense program is run on all the non-leaf nodes. Upon a worm infection, the infected dummy would alert its parent. Once the threshold is reached, the parent sends a similar alert to its own parent. In addition, it also extracts a signature from the suspicious packets received from its children. This signature is then sent to its children including the infected ones, and instructs them to block traffic matching this signature. Refer to [10] for further details of this model on back-off mechanisms, handling false positives, etc.,.

5.1.3 The worm program

Our experiments used the default worms provided by the framework. UDP random and sub-net scanning worms were deployed against our defense using a simple text string as the payload. No malicious programs were on the payload.

5.2 The experiment

All our nodes in the experiment ran FreeBSD 4.10 Jails. The controllers just copied the payload string into the signature distributed to the dummies. The dummies implemented the defenses using a combination of firewall and IPS. "ipfw" was the firewall of choice. This helped to divert packets arriving at a certain port to a program that could examine them for the malicious signature. "snort_inline" was the IPS of choice to examine the packets and drop packets that matched the signature provided by the controllers.

Our tool ran the experiments with 160 dummies and 21 controllers in 3 layers(1, 4 and 16) above them. We used both random scanning and sub-net scanning worms with *pods* of 10–100. In fact, a subnet-scanning worm with pods of 100 is the same as a random scanning worm. Each worm ran on a wide range of scanning speeds from 0.2 to 100 scans per second. Each experiment was run for 10 times, to smoothen out the stochastic effects. Thus 10 different worm kinds at 9 different speeds ran for 10 repetitions making for 900 experiments. It took about 18 hours to complete it. This is where the diligence of this tool comes to the fore. For sake of clarity, we only present the results of experiments with *pods* of 30, 60 and 100.

First we corroborated our worm spread pattern with no defense to the mathematical models and simulation results [10]. The results of this run is shown in Figure 2. This means that our worm program and the framework work as expected. Then we turned-on the defense mechanism. Figure 3 shows how the defense overtakes the worm spread. We used a worm with a speed of 2 scans/second.

5.3 Results

With the current set of experiments and the data analysis, we were able to draw several insights into the hierarchical defense strategy. These are:

1. The root node alerts all of its children to turn on defenses within a definite time from the first infection. All the experiments ran to completion. This shows that the system is convergent and does not run-away due to any feed-back effects. This verifies our mathematical proof given in [10].

2. No matter how fast a worm spreads, we can stop its spread with the same number of infections. This lets us decide the threshold parameters at the controllers based on our tolerance for infections. Figure 5 shows this result. Each data point shows the average number of infections and the standard deviation.

3. The experiments showed us that this scheme works better for suppressing subnet-scanning worms that are more biased towards scanning within the subnet than those that scan outside the subnet. This also means this scheme performs poorly against random scanning worms. Figure 5 shows this feature of the model. This is obvious once we realize that alerts go out of the subnet faster than the subnet scanning worms.

4. **Scale down factors:** Figure 4 and 5 together show that the stochastic effects of scaling down of networks can reduced by increasing the experiment repetitions. The experiment originally carried out with just 5 iterations gave us large standard deviations, whereas the one with 10 iterations gave a considerably lower deviations from the average value.

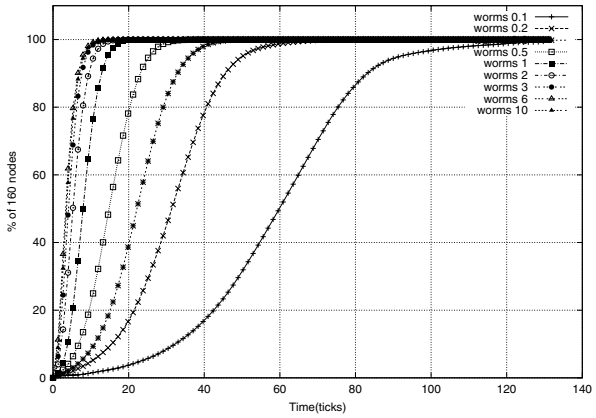

Figure 2: Random scanning worms with no defense

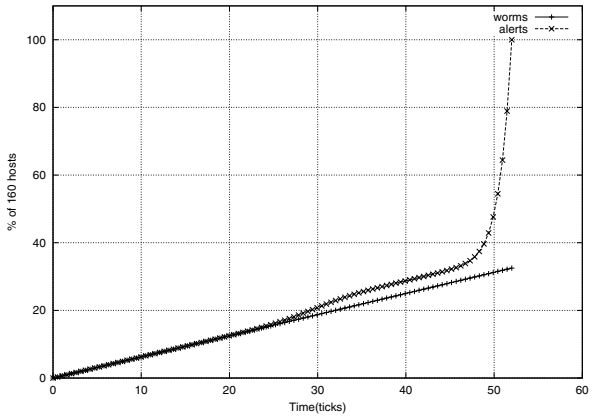

Figure 3: Alerts are distributed faster than the worms

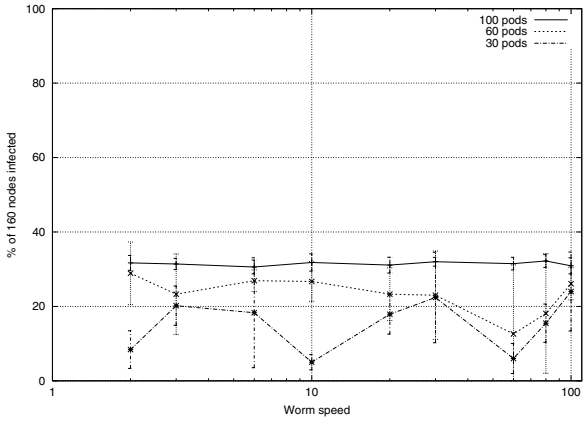

Figure 4: Only 5 iterations

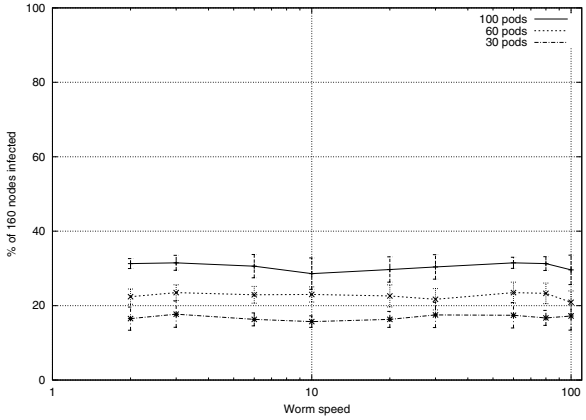

Figure 5: Increased to 10 iterations

5. It is only the threshold levels that makes or breaks the network. A low threshold helps to save a lot of machines but in reality it might help raise several false positives. Figures 6 and 7 show the effect of different thresholds.

6. FUTURE WORK

Currently, the data analysis tools only analyze the kinematics of worms. We need to design and implement the traffic analysis tools. As mentioned earlier, the effects on the network such as latency, bandwidth occupied by the defense vs the worm vs the normal traffic, the effects of false alarms on the normal operational efficiency needs to be analyzed. For this, we need to design program stubs to be inserted at the appropriate locations on the testbed.

In this paper we presented only an operational work-around to counter the scale-down effects on worm experiments. Two solutions to over-come this are to increase the experiment size and increase the number of iterations. Obviously, we can't emulate the entire Internet and we also can't repeat experiments indefinitely. We need to find a compromise between these two for our framework to be applicable to Internet wide worm problems.

We also want the users to be able to choose different kinds of networks from a library. The library would provide a set of environments like university, a commercial organization or a defense network. There are differences between these networks. University networks usually tend to be quite open with little or no firewalls enforced. It usually tends to have several web-servers hosted by individual departments as well as individuals. A commercial environment tends to be quite hardened on the outside but highly interconnected on the inside. Companies also have trusted connections with their suppliers. A defense establishment's network tends to be highly compartmentalized with rigid firewalls on the perimeter as well between different departments.

7. ACKNOWLEDGMENTS

A great deal of thanks to Mike Hibler of emulab/netbed, Univ of Utah for implementing 'on-demand' solutions to run divert-sockets on FBSD jails that helped us run snort_inline and ipfw on jails. Without them, we can't scale to real-sized networks. Thanks to Nick Rogness, also for 'overnight' enhancements to snort_inline.

8. REFERENCES

[1] Deter - a laboratory for security research. Internet. `http://www.isi.edu/deter`.

[2] "EMIST ESVT Software Version 2.0.". Internet. `http://emist.ist.psu.edu/ESVT2/download\_esvt2.html\#Overview`.

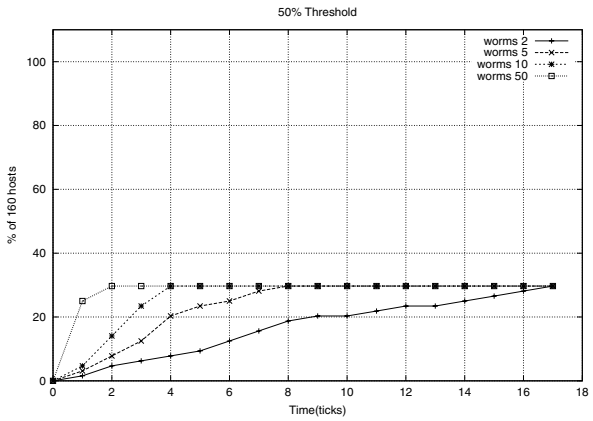

Figure 6: Worm Containment with 50% Threshold

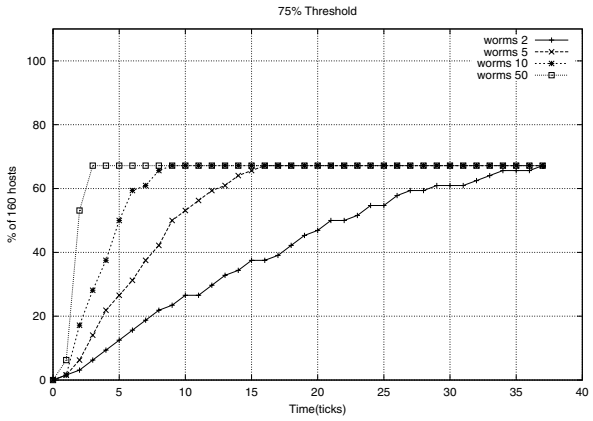

Figure 7: Worm Containment with 75% Threshold

[3] "EMIST Project Overview". Internet. `http://www.isi.edu/deter/emist.temp.html`.

[4] "EMIST Tool Suite". Internet. `http://www.cs.purdue.edu/homes/fahmy/software/emist/index.html`.

[5] "NS: Nework Simulator". Internet. `http://www.isi.edu/nsnam/ns`, Last Accessed: Feb 06, 2006.

[6] "SSFNet". Internet. `http://www.ssfnet.org` Last Accessed: June 21, 2005.

[7] M. Abdelhafez and G. Riley. "Evaluation of worm containment algorithms and their effect on legitimate traffic". In *Third IEEE International Workshop on Information Assurance (IWIA)*, March 2005.

[8] L. Briesemeister and P. Porras. "Microscopic simulation of a group defense strategy". In *Proceedings of Principles of Advanced and Distributed Simulation (PADS)*, June 2005.

[9] L. Briesemeister, P. A. Porras, and A. Tiwari. "A Formal Model of Worm Quarantine and Counter-Quarantine under a Group Defnese". In *Proceedings of Malware'06*, Apr. 2006. To appear.

[10] S. G. Cheetancheri. "Modelling worm defense systems". Master's thesis, June 2004. University of California, Davis.

[11] D.Noijiri, J.Rowe, and K.Levitt. "Cooperative Response Strategies for Large Sacle Attack Mitigation". DISCEX, 2002.

[12] S.-S. Hong and S. F. Wu. "On Interactive Internet Traffic Replay". In *Proceedings of the Eighth International Symposium on Recent Advances in Intrusion Detection*, 2005.

[13] M. Liljenstam, D. M. Nicol, V. H. Berk, and R. S. Gray. "Simulating Realistic Network Worm Traffic for Worm Warning System Design and Testing". In *Proceedings of the I ACM Workshop on Rapid Malcode (WORM03)*, Washington, DC, October 2003.

[14] R. P. Lippmann et al. "Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation.". In *Proceedings of the 2000 DARPA Information Survivability Conference and Exposition (DISCEX)*, 2000.

[15] J. McAlerney. "An Internet Worm Propagation Data Model". Master's thesis, Sept. 2004. University of California, Davis.

[16] T. H. Ptacek and T. N. Newsham. "Insertion, Evasion and Denial-of-Service: Eluding Network Intrusion Detection". Technical report, Secure Networks Inc.,, Jan 1998.

[17] Y. A. Ting, D. Ma, J. Rowe, and K. Levitt. "Evaluation of Collaborative Worm Containment Strategies". Work in Progress.

[18] Y. A. Ting, D. Ma, J. Rowe, and K. Levitt. "NTGC: A Tool for Network Traffic Generation Control and Coordination". Work in Progress.

[19] N. Weaver, I. Hamadeh, G. Kesidis, and V. Paxson. "Preliminary Results Using Scale-Down to Explore Worm Dynamics". In *Proceedings of the II ACM Workshop on Rapid Malcode (WORM04)*, Washington, DC, Oct 2004.

[20] N. Weaver, S. Staniford, and V. Paxon. "Very Fast Containment of Scanning Worms". In *Proceedings of the USENIX Security Symposium*, Aug. 2004.

[21] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.