Modeling and Analysis of Worm Defense Using Stochastic Activity Networks

David M. Nicol, Steve Hanna, Frank Stratton, William H. Sanders

Information Trust Institute

University of Illinois at Urbana-Champaign

Abstract— **Stochastic activity networks (SANs) are a widely used formalism for describing complex systems that have random behavior. Sophisticated software tools exist for the modeling and analysis of systems described within a SAN framework. This paper presents a SAN model of a local area network's defense against Internet worm propagation, measuring the effectiveness of a defensive strategy based on removing hosts from the local network once an infection is detected. We consider the problem of deciding whether to allocate resources to remove an infected host (and thereby reduce the threat), or remove a susceptible but as-yet uninfected host, to directly save it from attack. Considering a parameterized range of policies that makes this decision based on the number of infections in the local network, we find marked preference for always removing one type of hosts when possible, over the other, regardless of the infection state. We futhermore see whether preference should be given to infected hosts or susceptible hosts depends on the relative speeds at which they are removed. Finally, we see that a worm attack can be effectively countered provided that the aggregate rate at which hosts can be removed is on the order of the aggregate infection rate at the time the defense is engaged. Our effort demonstrates the utility of using sophisticated modeling tools to study worm defense, and policy decisions surrounding it.**

I. Introduction

On July 19, 2001 the TCP based worm Code Red 1 version 2 began to spread. The worm exploited a buffer overflow in Microsoft IIS Server, and continued to infect computers around the globe. The worm had several secondary side effects in addition to spreading. First, if the system infected system's language was English, it would cause a worm-deposited message to be displayed when a web page was requested [1]. In addition, if the packets were forwarded through an unpatched Cisco-600 router, it triggered an unrelated exploit, which stopped packet forwarding [2].

Most of the Internet worms that have gained widespread public attention found new victims by choosing potential victims at random. The simplest scheme is to choose a target uniformly at random from among all possible IP addresses; more sophisticated preferential schemes target particular sub-networks with higher probability, and an infection within a network targets others in the network with higher probability than others.

The question of how to defend against a worm attack is quite timely. Code Red has been studied extensively [3], [4]. In [3], the authors analyze the behavior of the worm and the impact on the Internet. In [4] they develop a two factor epidemic worm model to analyze how traffic impeded upon spread as well as ISP intervention as an active attempt to filter Code Red traffic. The authors provide a unique contribution showing the number of attacks by source port and frequency.

Our contribution is to show how a random scanning worm with preferential scanning can be modeling using the stochastic activity network (SAN) formalism, use that model to investigate the impact of a defense based on removing infected and/or susceptible hosts from the network, and explore the effectiveness strategies based on this idea. In particular, we jointly simulate the the infection of a /16 sized network and "the rest of the Internet" with two different models which interact. Within the subnetwork we consider a strategy based on removing machines from the network, allocating a capability to do so between hosts that are already infected, and hosts within a superset that contains hosts that are susceptible, but are not yet infected. Removal of an infected hosts reduces the worm's capability to spread, yet removal of a susceptible but uninfected host ensures the safety of that host. When the objective is to minimize the number of hosts that become infected, a defense based on saving susceptible hosts is direct while a defense based on suppressing infections is indirect. We address the question that asks which approach is more effective, and in doing so observe how fast the removal process must be relative to the infection process to be effective.

II. Model Description

Our model represents a subnetwork comprising a /16 (i.e., 2^{16} IP addresses), with a small subset of hosts that are vulnerable to a worm. The subnetwork is connected to the larger Internet using a gateway router, modeled after a a Linksys-RVS4000 with 800-Mbps maximum bandwidth [9]. There is also a capability to reduce the threat of more infections by removing infected hosts from the network, or protecting hosts that might still become infected. This capability is modeled as a small pool of servers.

We model the effect of the worm spreading on the Internet and the associated incoming traffic to our network from a growth curve with a peak number of 360,000 infections [3]. The model does not assume a particular time-scale for the worm scans; we couple the speed of the response mechanism to the speed of the infection. The difference between the two is an important feature of the model parameters.

In designing our model, our main goal was to capture the influence of network bandwidth, infection rates, and host removal rates, while maintaining a level of simplicity that results in a quickly solvable model. Our model consists of three main stochastic activity networks. SANs are an extension of stochastic Petri nets. Circles represent "places" in which "tokens" (not seen graphically) may reside. The vector of token counts in places defines the system state. The number of tokens in each place can change when "activities' complete. Transitions are represented by lines, either thin ones (so-called instantaneous activities) or fat ones (so-called timed activities). In the SAN formalism an activity may have an "input gate" connected to it, with logic that describes the system state under which the activity is enabled to complete. A timed activity interposes a delay between the instant when the activity becomes enabled through some change in the system state, and when the firing actually occurs. The logic in an input gate is a Boolean function of the system state. A SAN may also use an "output gate" following an activity. Logic within the output gate describes the modification to the system state that occurs as a result of the firing.

These ideas are better understood in the context of our specific SAN model. We have three distinct but inter-related pieces. Figure 1 shows a small model that describes how the rest of the internet evolves. The place labeled GI contains a value (i.e., number of tokens) recording the number of infections currently extant in the global internet outside of the subnetwork of interest. There follows an input gate (*ext_int*) and a timed activity. The input gate's logic specifies that the activity is enabled provided that there remain susceptible-but-uninfected hosts in the subnetwork of interest, and also in the global internet. The rate associated with the activity is

$$GI \times (S_0 - GI) \times \lambda / 2^{32}$$

where GI is the value in the place of the same name, S_0 is the original number of susceptible hosts in the global internet (excluding the subnetwork), λ is the rate at which infection packets are launched into the internet by an attacker. The delay between when the activity is enabled to complete and when it completes is an exponential with this rate; this construction is an SAN expression of the Time-of-Next-Infection (TNI) model discussed in [10]. The output gate logic is invoked after firing, and serves to add one to the the value in GI. This is an accurate description of activities in a worm that select all targets uniformly at random, and an approximate description when the worm scans preferentially, i.e., targets hosts close to it in IP space with higher probability. For simplicity of exposition we use only the simpler model, even though we do consider preferential scanning.

Our SAN contains another simple component that governs when patching may occur. Illustrated in Figure 2, this network has a place "Oblivious" with an initial marking of 1. The input gate to the activity enables an immediate firing once the model's detection criteria are satisfied. Our experiments assume that a worm is detected when 4 of the hosts in the local network are infected. Other criteria can easily be included here, e.g., triggers based on the numbers of infections in the global network.

Figure 3 illustrates the SAN which describes how the local network evolves. At the left we find a placed labeled *Hosts*, marked with an initial value equal to the number of susceptible hosts. Tokens from *Hosts* move to the place entitled "Infected" through a timed activity entitled *Infect*, or move to the place entitled *takedown2*. The latter case occurs when an uninfected host is selected to be removed from the network, the former case when an uninfected and unselected-for-removal host becomes infected. The infection activity is enabled by input gate *enable_infection* so long as there are susceptible hosts in the system; the rate associated with the activity is the sum of the rate at which infections within the subnetwork infect other hosts in the subnetwork, and the rate at which the global network infects the local area. This expression is not especially simple. The set of statements we place in the form describing the activity is

```
double lr = bias*scanRate*Infected->Mark()
 *(Hosts->Mark()+offline2->Mark())/subnetsize;
double gr = (1.0-bias)*GI->Mark()*scanRate*
    (subnetsize/4.294967e9);
return (lr + (gr<Bandwidth?gr:Bandwidth/8000)
     *Hosts->Mark()/subnetsize);
```

The Möbius tool takes the set of statements provided and uses it to create the body of a C++ procedure that is called to compute the activity's rate. The value of places in a Möbius model are obtained through calls to the *Mark()* method, where the place label is used as a name and the member function *Mark()* refers to the value contained. Möbius allows one to

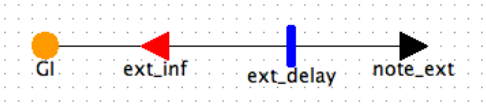

Fig. 1. Global Internet SAN Model

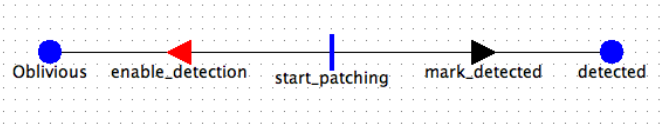

Fig. 2. Infection Detection SAN Model

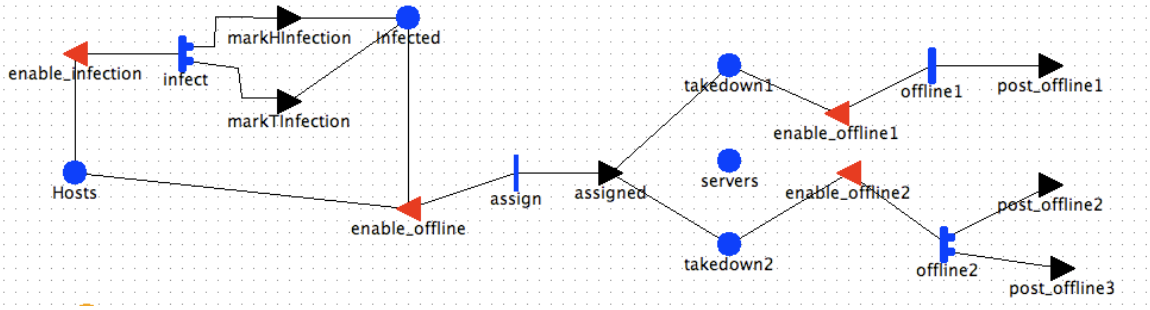

Fig. 3. Subnetwork SAN Model

define global variables and initialize them. This code fragment contains three of these, *scanRate*, *subnetsize*, and *Bandwidth*, denoting (respectively) the rate at which an infected host injects scans into the network, the number of hosts in the local network, and the rating of the gateway router to push bits from the global Internet into the subnetwork. The value computed for the locally induced infection rate (*lr*) takes a fraction (*bias*) of scans generated and targets them entirely within the subnetwork. The value for *gr* takes the fraction of all scan traffic generated by the infected hosts in the global internet directed to the subnetwork (of size 2^{16}) , caps it by the bandwidth limitation if that is too large, and computes the probability that a scan from the global internet reaches one of the subnetwork's susceptible hosts.

Transition *Infect* has two branches, to different output gates. At the end of the activity completing one of these is chosen randomly, using probabilities encoded in the activity structure. The logic here comes from our wish to model the possibility that an infection overtakes a host for which a take-down server has been dispatched. We need to distinguish between hosts scheduled for takedown and those that have not, and so move a token from the *Hosts* place to the *takedown2* place when it is selected to be removed. At the end of *Infect* firing, gate *markHInfection* is chosen with probability equal to the number of tokens in *Hosts* divided by the number of tokens in *Hosts* and *takedown2*. Gate *markTInfection* is chosen with complimentary probability. Both update a high-water mark for tokens in place *Infected*; *markHInfection* moves a token from *Hosts* to *Infected*, while *markTInfection* moves a token from *takedown2* to *Infected*, and adds a token to *servers* (to reflect release of the server assignment.) This aspect of the model may seem counter-intuitive—why release a server just because the targeted host becomes infected? Our model implements this mechanism because it allows for the removal service time for an infected host to differ from that of a merely susceptible host. The *remaining* service time of an interrupted exponential service time is itself exponential, so the mechanism of releasing the server does not in any way "forget" some important aspect of service time. It just allows the decision policy to allocate the released server, rather than preemptively force the otherwise server to now deal with an infected host.

Input place *enable_offline* governs whether infected hosts or susceptible hosts are subject to being taken offline to protect them (and the subnetwork). The enabling conditions are that the worm is detected, removal servers are available, and that tokens exist either in *Infected* or *Hosts*. The *assign* activity is immediate, no time elapses between its enabling condition and the execution of the logic in output gate *assigned*. That logic binds a removal server to take down either an infected machine, or a susceptible machine. Our experiments exercise different ways of making the assignment, as follows.

A global variable holds a threshold against which we compare the number of infected hosts that have not yet started the network disconnection. If the experiment uses a "Infections Below Threshold" (IBT) policy, a server is allocated to an

infected host if the number of infections is (strictly) below the threshold, otherwise a server is assigned to a host. In the limit of increasing threshold value, the policy simply allocates servers to infections, so long as there are infections remaining to bring down. If there are no instances of the type of host selected by the policy, then the other type is selected. The logic is reversed in experiments using the "Hosts Below Threshold" (HBT). The IBT policy encodes a desire to attack infections if the system is not overwhelmed by them, but if the infection count is too high then just save hosts as fast as possible. The HBT policy encodes a desire to save hosts so long the infections aren't yet too dangerous, but attack them once they have reached critical mass. Here again the power of the SAN formalism is that this policy can be written in C++, linked into the analysis engine, and be invoked at this point in the model analysis. This is an important point, because it is not immediately clear what the best policy ought to be—what is the effect on the future high water infection mark or total number of infections? To take down an infected host reduces the infection rate on the remaining susceptibles, but they may become infected anyway. To take down a susceptible host (rather than an infected one) may save that host, but allow for more to be infected than would occur if an infected host were taken down. Our experiments show that a tool like the one we have developed can be used to investigate optimization options.

When *assigned* maps a server to take down an infected host it removes a token from the *servers* place and adds one to the *takedown1* place. Likewise, mapping a *servers* member to take down a susceptible host removes a token from the servers pool and adds one to *takedown2*. In our model both the *offline1* and *offline2* activities are timed, with exponential distributions. While the precise form of a "real" distribution is unlikely to be rigorously exponential, it does give us a way to include variance (the geographic area served by a /16 is probably large enough to force some travel or communication between sites, hence some variation in the time between a server is assigned to a task and when that task completes). An exponential is also useful because it allows us to just describe the service rate of the activity as the product of the number of servers engaged in taking down infected hosts (alt., susceptible hosts), and a service time required by one server. When *offline1* completes, output gate *post_offline1* removes a token from *takedown1*, adds a token to *servers*, and removes a token from *Infected*. The firing of *offline2* is slightly more complex. We model the possibility that a host removed from the network is not actually susceptible. The *offline3* logic describes a probabilistic branch on firing. With a specified probability the logic encapsulated in *post_offline3* is executed, which restores the server, decrements the *offline2* count, With complementary probability the logic of *post_offline2* is executed. This does what *post_offline3* gate does, and in addition decrements the number of susceptible hosts.

The system updates the activity rate for the activity that just completed because of this dependency (as it does when the associated *post_offline* places receives another token.)

III. Model Parameters

Our experiments fix a number of experimental parameters for all our studies.

- the subnetwork size is 2^{16} IP addresses;
- there are initially 255 susceptible hosts in the subnetwork, and 360000 in the whole internet;
- every experiment begins with one infected host in the global Internet, and none in the subnetwork;
- the single gateway router carries 800-Mbps between the global Internet and the subnetwork;
- the local bias probability in scanning is 0.25;
- there are 8 servers available to remove hosts from the network,
- the reaction mechanism sets in when four hosts in the local network have become infected.

We learn about the effectiveness of removing infections versus hosts by varying the threshold value governing whether to use a free server to remove an infection, or a potentially susceptible host. We learn about the impact that server speed has on response by varying it; in our experiments we link this speed to a "base rate", defined to be the rate at which the first infection within the subnetwork infects another host in the subnetwork. That is, base rate

$$\lambda_b = \lambda_s bias(S-1)/n$$

where λ_s is an infection's scanning rate, S is the number of initial susceptibles in the subnetwork, and n is the number of hosts in the subnetwork. Our experiments use values of $\lambda_b/8$, $\lambda_b/4$, $\lambda_b/2$, and λ_b. The aggregate rate at which infections or hosts can be removed depends on the number of servers, 8 in our experiments. If removal server's rate is λ_r, the aggregate removal rate is $8\lambda_r$; if p_s is the probability that a removed host is susceptible, then the aggregate rate of removing susceptible hosts is $8p_s\lambda_r$.

All 8 servers spring into action once the local subnetwork has four infected hosts. At this instant the infection rate due to intra-subnetwork scanning is slightly less than $4\lambda_b$; the infection rate grows with every successive infection. The worm can be contained so long as its infection rate is no greater than the rate at which infections are removed. Thus, at the instant that the servers are released, the worm has already "won" in a sense, when $8\lambda_r < 4\lambda_b$. Writing $\lambda_r = \alpha\lambda_b$ to reflect our parametrization of λ_r, the system cannot stop growth in the infection rate when $\alpha < 0.5$. In our experiments this occurs when $\lambda_r = \lambda_b/8$ and when $\lambda_r = \lambda_b/4$. There is clearly strong motivation for using servers to remove infected hosts so long as by doing so we can decrease the number of infected hosts. In the case that we cannot reduce the number of infections by removal, it is an open question whether it is better to use removal servers to save susceptible hosts, or continue to attack infections. Our experiments shed some light on this question.

IV. Experimental Results

Figures 4 and 5 illustrate the results of our experiments. Each data plot is the result of 100 independent replications

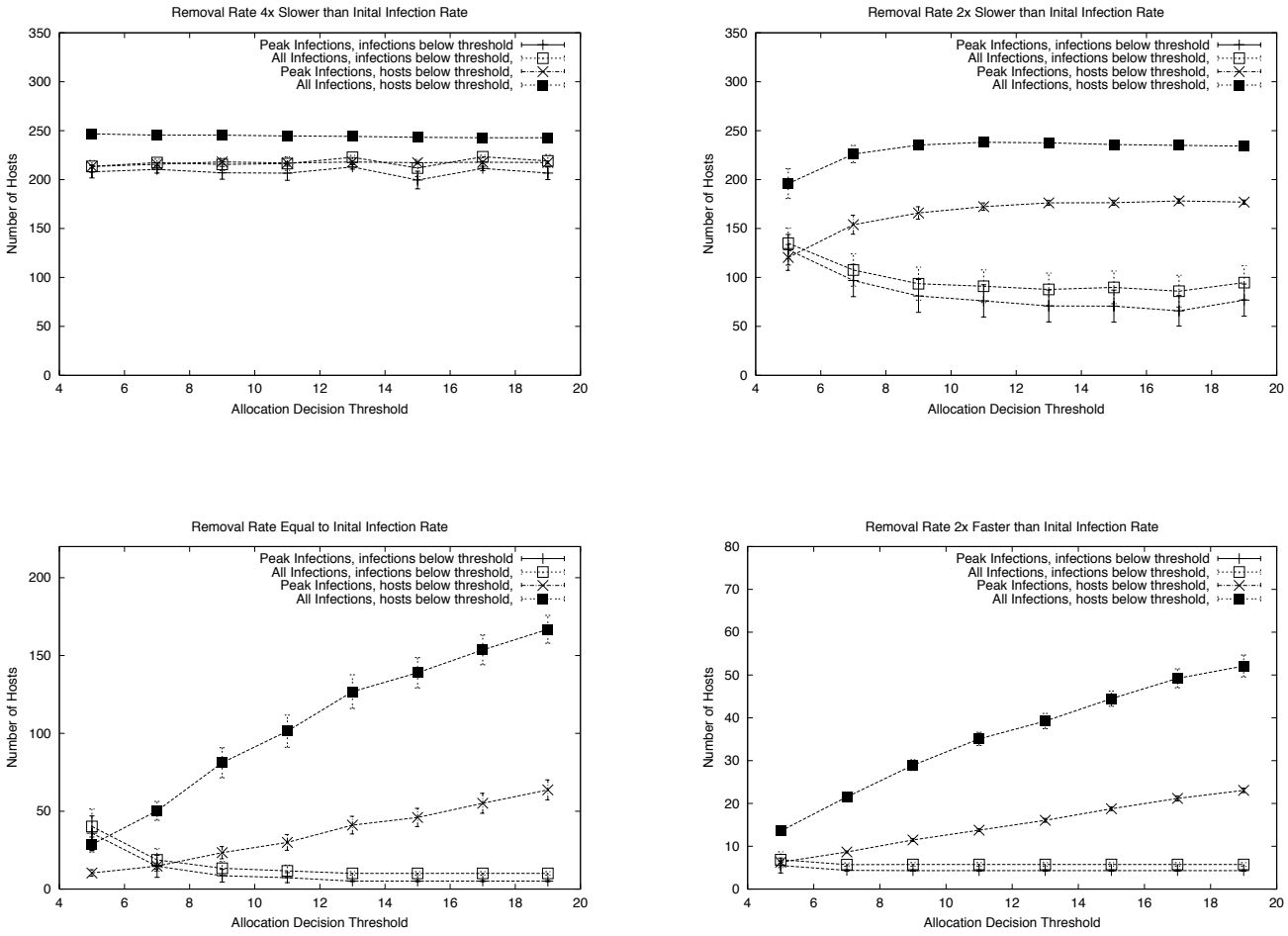

Fig. 4. Infected hosts when removal speed for infections and susceptible hosts are identical

of a discrete-event simulation, and each point has error bars showing the confidence interval at a 95% confidence level. A simulation run terminates when there are no remaining susceptible hosts on-line in the local network. Figure 4 reports experiments where the removal rate for infected hosts is identical to the removal rate for susceptible hosts; Figure 5 revisits the same experiments but when removing susceptibles is 10 times faster than removing infections.

Each graph plots the peak (e.g., maximum) number of hosts infected in the local network and the sum of all infections that occur in the local network; the peak rate gives a measure of the policy's ability to keep the infection rate in check. These results are shown for the both the IBT (infections below threshold) and HBT (host below threshold) policies.

The upper left graph in Figure 4 corresponds to the aggregate removal rate being 4 times slower than the aggregate infection rate at the time the worm is detected. The servers cannot possibly keep up, and they don't. The upper right graph describes when the aggregate removal rate is 2 times slower. Here we see the beginning of two interesting trends in this graph and the two in the lower row: IBT metrics tend to decrease with increasing threshold while HBT metrics tend to increase, and IBT metrics tend to be significantly smaller for a given threshold. The implication is that regardless of whether the removals can keep up with the worm or not, the strategy to follow is to allocate servers to infections. Even when the removal rate is half the infection rate, many hosts are rescued in the early phase of the defense when there are more servers available than infections (and so the available servers take susceptible hosts offline). The lower right and lower left graphs show the impact of having removal rates that keep up with the infection—the IBT policy (with threshold so large it is never crossed) holds the infection count to the value it has when the worm is first detected.

Intuition suggests that preference observed for infected hosts depends on the rate of infection removal in relation to the rate of removing susceptible hosts—if removing infections takes longer (e.g., involves finding them or performing some maintenance by hand whereas removing susceptibles is automated) perhaps it is better to use servers to save

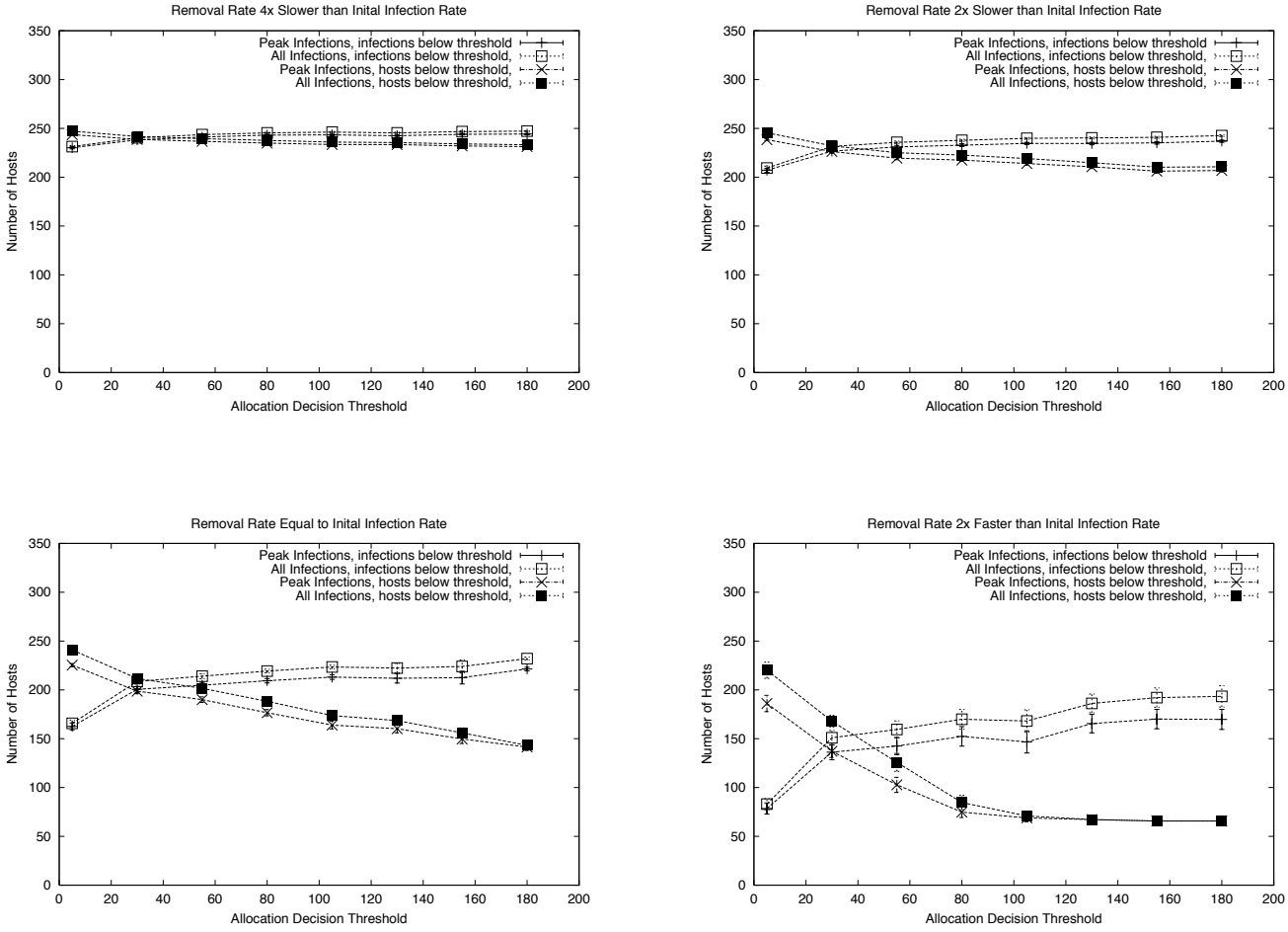

Fig. 5. Infected hosts when removal speed for infections is 10x slower than for susceptible hosts

uninfected hosts. Figure 5 illustrates results from the same set of experiments, but with the infection removal rate being one tenth the value it had before. To see the trends here we show metrics measured over a wider range of decision thresholds. Now we see that the roles of IBT and HBT are reversed—HBT decreases with increasing threshold while IBT increases, showing a preference for using servers to rescue hosts. However, the values of the metrics are considerably larger than for the earlier set of experiments. For example, comparing the lower right graphs of Figures 4 and 5, we see that when the aggregate removal rate is twice the initial infection rate, the number of infections suffered in the best case is ten times larger in the second set of experiments than in the first.

The experiments presented suggest the use of a server allocation policy that prefers to take down a given host type when an instance of such exists (and allocates to the other when there are no instances of the preferred type). Whether the preference should be for infected hosts or for susceptible hosts depends, at a minimum, on the relative speed of removing infections compared with the speed of removing susceptibles. As our experiments leave a number of model parameters fixed, it is premature to assume that this relative speed is the only parameter of interest. A promising aspect of the results so far is that the parameter of interest is one that can be evaluated prior to a worm attack, and so a certain level of planning can be usefully conducted against the possibility of attack.

V. Conclusion

We used Möbius to model the spread of Internet worms, and defenses based on removing infected and susceptible machines from the network. The tool is able to capture complex dependencies and server allocation decision policies. We use the tool to consider defense policies that allocate servers to remove infected hosts, or susceptible hosts, the allocation decision depending on the comparison of the number of infected hosts present against a policy threshold. By varying that threshold we find the best policy (at least on the experiments we conducted) is to allocate servers preferentially always to one type of host, or the other. Whether preference should be given

to infected hosts or to susceptible hosts is seen to depend on the comparative speeds at which one host or the other can be removed. We also see that the defense is effective provided there is an aggregate removal rate that is close to the aggregate infection rate at the point of detection. A removal rate that is as little as 1/2 the aggregate infection rate at detection can still rescue a significant number of susceptible hosts.

In addition to providing insight into worm behavior, the results show that a tool like Möbius can be a powerful and useful component for an analyst interested in best choosing how to spend dollars and deploy policy to protect a large IT infrastructure.

ACKNOWLEDGMENT

The help of Tod Courtney is gratefully acknowledged. This research was supported in part by DARPA Contract N66001-96-C-8530, and NSF Grants CCR-0209144 and CNS-0524695. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. In addition, this research is a part of the Institute for Security Technology Studies, supported under Award number 2000-CT-CX-K001 from the U.S. Department of Homeland Security, Science, and Technology Directorate. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security or the Science and Technology Directorate.

REFERENCES

[1] CERT Coordination Center. CERT Advisory CA-2001-19 'Code Red' Worm Exploiting Buffer Overflow In IIS Indexing Service DLL, July 2001; http://www.cert.org/advisories/CA-2001-19.html .

[2] Cisco Security Advisories. 'Code Red'Worm - Customer Impact, July 2001; http://www.cisco.com/warp/public/707/cisco-code-red-worm-pub.shtml .

[3] C. Shannon, D. Moore, and J. Brown. "Code-Red: A Case Study on the Spread and Victims of an Internet Worm," Proc. Internet Measurement Workshop (IMW), ACM Press, 2002, pp. 273284.

[4] H. Berghel. The Code Red Worm, Communications of the ACM, 2001. 44(12):p. 15-19.

[5] Mobius. http://www.mobius.uiuc.edu/ .

[6] S. Friedl. Analysis of the new 'Code Red II' Variant, Aug. 2001; http://www.unixwiz.net/techtips/CodeRedII.html .

[7] N. Weaver. The Spread of the Sapphire/Slammer Worm, http://www.caida.org/publications/papers/2003/sapphire/sapphire.html

[8] D. Moore and C. Shannon. The Spread of the Witty Worm, http://www.caida.org/analysis/security/witty/.

[9] Linksys. Linksys Data Sheet, `http://www.linksys.com/servlet/Satellite?c=L_Product_C2&childpagename=US%2FLayout&cid=1150490915278&pagename=Linksys%2FCommon%2FVisitorWrapper`

[10] D. M. Nicol. The Impact of Stochastic Variability on Worm Detection, In *Proceedings of the 2006 ACM Workshop on Rapid Malmare (WORM'06)*, Alexandria, VA, pp. 57–64.