Increasing Communications Security through Protocol Parameter Diversity*

Elena G. Barrantes
Universidad de Costa Rica, Escuela de Ciencias de la Computación e Informática
San Pedro, San José, Costa Rica, 2060
gbarrantes@ecci.ucr.ac.cr

and

Stephanie Forrest
University of New Mexico, Department of Computer Science
Albuquerque, New Mexico, 87131
forrest@cs.unm.edu

Abstract

Pulsing attacks use carefully synchronized high-rate low-duration bursts of traffic that are injected into a network to induce denial-of-service. These attacks are effective because the bursts force protocols into low-performance states. The synchronization relies on the fact that most computers run protocols with identical parameter values. The use of diverse parameter value could make the attacks much less likely to succeed. This paper describes how parameters in TCP protocol implementations can be automatically diversified, introduces several evaluation metrics, and assesses the efficacy of this defense using Kuzmanovic's *shrew* pulsing attack. The experiments show that in a diversified environment under attack, some hosts can obtain near normal throughput, while average network throughput is improved for most (but not all) attack scenarios. Heterogeneity of parameter values among hosts is key to the defense.

Keywords: Security, networks, pulsing attacks, diversity defenses.

*The authors gratefully acknowledge the partial support of the National Science Foundation (grants ANIR-9986555, CCR-0219587, CR-0085792, CCR-0311686, EIA-0218262, and EIA-0238027), Intel Corporation, and the Santa Fe Institute.

1 Introduction

Many kinds of attack take advantage of well-known regularities in our computing infrastructure. For example, code injection attacks exploit common knowledge about memory layouts (e.g., stack or heap), email viruses rely on the fact that email address books are stored in standard locations and formats, and file virus use knowledge about the file system structure and naming conventions. Recently, several projects have addressed the problem of widely replicated attacks through a technique known as automated diversity, (see [15] for a survey) In most of the diversity work, the emphasis has been on diversity of interfaces, for example, instruction set numbers (instruction set diversity) [3], or names of library functions [5]. In these examples, the programs execute identically, and the diversity changes the conventions by which the programs communicate with one another.

A second, and more ambitious, source of diversity is known as *implementation diversity* [6]. Here, the idea is to change the way in which a given function is implemented, that is, potentially changing the behavior of the program. Implementation diversity for security purposes can be thought of as n-version programming scaled up so that n is large, potentially as large as the number of computers being protected, and the versions are generated automatically instead of by teams of programmers. Such an approach is inherently risky, as it is difficult to guarantee both that the multiple implementations correctly implement the target functionality and that they are likely to foil an attack.

In this paper, we study a modest form of implementation diversity, in which we modify certain parameters of the Transmission Control Protocol (TCP) protocol to make timing properties of the protocol more difficult to predict. This form of diversity is designed to mitigate the *shrew* attack [16] that slows down TCP by simulating congestion conditions. One of the problems with protocol diversification is the difficulty of modifying or generating code that respects the abstract protocol definition. We simplify the process by identifying numeric parameters of the protocol, determine their acceptable ranges of values, and randomize them within that range.

1.1 Threat Model

A communication protocol defines the set of rules that two or more entities use to interact. In practice, most protocols are explicitly or implicitly defined as extended finite-state machines (FSMs) [13], with state-by-state rules defining: how to interpret received messages, what to do in their absence after some time has elapsed, what messages should be sent, how to update state variables, and under what conditions the FSM transitions to another state.

Ideally, the conditions would refer to observable states of the real environment, but in practice, indirect indicators are used that rely on imperfect measurements and require artificial mechanisms to recover from failures. For example, TCP uses the time between the transmission of new (non-retransmitted) packets and the reception of their acknowledgements (ACKs) to construct a weighted average which estimates the round trip time (RTT) between two endpoints at any given moment in time. The diversity of conditions that causes delays makes the parameters used in the construction of the weighted average an approximation of the true RTT. The estimated RTT is used to set the retransmission timeout, so its accuracy is essential for TCP to decide correctly when to retransmit without waiting too long, but also to avoid retransmitting too eagerly, thereby flooding the network with unnecessary retransmissions. Because of observed fluctuations of the mean, TCP designers added additional measures (minimum waiting times) to ensure that retransmissions would not be premature in the case of congestion [2].

The protocol is vulnerable to attacks that deceive the estimators, the *shrew* attack [16] being a concrete example. The vulnerability arises because the true state of the network is not observed directly and because the estimates rely on parameters that are somewhat arbitrary and often have hardcoded default values. These parameters can create exploitable regularities in the estimators. The fact that the parameters are fixed makes it all that much easier to fine-tune attacks by adjusting timing or content to deceive the estimators.

There are several ways to exploit fixed parameters. For example, communication protocols typically include emergency states to recover from problematic situations in the environment. If an attacker can perturb the environment in such a way that the estimators signal a nonexistent emergency situation, the the protocol will incorrectly enter one of the slower, recovery-mode states. In a slight variation, the protocol can be manipulated so that it continuously switches between states or restarts any one of them. This type of attack is a form of Denial of Service (DoS). Although it does not gain control of the hosts, it is potentially a serious problem because it denies legitimate users access to resources.

There are many arbitrary parameters in the implementation of any protocol, and many of them could profitably be diversified. In this paper, however, we focus on the parameters relevant to the *shrew* exploit presented by Kuzmanovic and Knightly [16]. This attack uses carefully synchronized traffic surges to force TCP to enter the *slow-start* mode repeatedly. The parameter being exploited in this case is the minRTO parameter, which is part of the TCP timeout (RTO) calculation and was originally designed as part of the congestion control mechanism.

Section 2 reviews the TCP congestion control algorithm and the how retransmission timeout is calculated, to illustrate how the fixed *minRTO* parameter is exploited by the *shrew* attack. As we mentioned earlier *shrew* is just one example of a general class of possible attacks on TCP parameters. The attack itself is described in Section 3. In Section 4 the diversification defense is discussed, focusing on how, when and what parameter values to randomize. Our data collection strategy is described in Section 5, Section 6 discusses various evaluation criteria, and Section 7 reports experimental results. Related work is discussed in Section 9, ideas for future work in Section 10, and a summary of our findings is given in Section 11.

2 Background: Congestion Control in TCP

The Transmission Control Protocol (TCP) [2] uses adjustable timeouts to determine when data have been lost and when to retransmit.There are two main variables that any windowing retransmission protocol must control: the size of the receiver (and/or transmitter) window and the window timeouts [4]. The management of these factors in TCP is briefly described below. In terms of the part of the FSM we are concerned with, after a connection has been established, and as long as a connection is not considered lost or finished, it sends and receives packets, transitioning between the following four implicit states: *slow start* (SS), *congestion avoidance* (CA), *fast retransmit* (FRT) and *fast recovery* (FRE). We are only to be concerned with the SS and CA states here.

Once a valid round trip time (RTT) measurement is made, the state variable of interest becomes the Congestion Window (CW), initialized to 1 segment. During SS, TCP increases the congestion window exponentially until data are lost or the limit is reached. In the absence of errors, the exponential growth stops when the transmitting window reaches either (1) the threshold between slow start and congestion avoidance (SSTRESH) or (2) the receiver-advertised window (RWND). Most implementations start SSTRESH at the same value of RWND. At this point, TCP transitions to the congestion avoidance (CA) state. During CA, the transmission window is incremented by one each RTT (not at each ACK), thus increasing the window size only linearly. CA stops increasing the transmission window once it reaches RWND.

At any point during this process a timeout cuts the value of SSTRESH in half (roughly). More importantly, CW is set to LW (the Loss Window), which is never to be larger than one segment. After the unacknowledged packets are retransmitted, TCP returns to the SS state.

Not only the window is shortened (reducing the number of packets sent during a retransmission), but the waiting time is expanded. After a timeout, RTO (the retransmission timer) is doubled, and it is reset from a measured RTT only after an ACK is received for new data. The recorded measurement (RTT_{sample}) maintains an up-to-date estimate of the round trip time using the smoothed round trip time (SRTT) and the round trip variance (RTTVAR), as follows:

$$SRTT \leftarrow (1-\alpha)SRTT + \alpha RTT_{sample} \quad (1)$$

$$RTTVAR \leftarrow (1-\beta)RTTVAR + \beta|SRTT - RTT_{sample}| \quad (2)$$

$$RTO \leftarrow max(minRTO, SRTT + max(G, k * RTTVAR)) \quad (3)$$

where $\alpha = \frac{1}{8}$, $\beta = \frac{1}{4}$, $k = 4$, and $minRTO = 1s$ as recommended in the relevant RFCs [1] [2, 19]. The parameter G refers to the clock granularity in the host.

In summary, a timeout forces a return to SS and a dramatic decrease in window size and increase in waiting time. If it were possible to guess when the lost packets would be retransmitted and cause them to be dropped, then the affected data flow could be kept from sending any new data forever. This is the basis of the *shrew* attack described in the next section.

[1] All Internet protocols are described in RFCs *Request for Comments* documents, which are generated and maintained by the Internet community.

3 The *shrew* attack

The critical observation made by Kuzmanovic and Knightly is that because most round trip times on the Internet are below 1 second, Equation 3 returns minRTO most of the time. By inducing losses with a period of minRTO, it is possible to force most flows to reset continuously to SS. This can reduce throughput to nearly zero as the flow will continue resending packets that were in transit during the first loss, and it will not move on to new data [16]. In a typical environment with several flows on different timescales, a burst once per minRTO is necessary and sufficient to implement the attack.

The attack is based on synchronized high-rate low-duration packet bursts. The rate of the burst must be close to the wire rate (e.g. 100 Mbps in a Fast Ethernet network) in order to induce loss. Even if the bursts have very high bandwidth, their duration and frequency can be chosen so that the *average* attack bandwidth is relatively low. The length of the burst (b) must be larger than most real RTTs in the target system to ensure that at least one packet of most flows will be dropped. Finally, the burst is sent each T seconds ($T >> b$), known as the *period of the attack*. In a system with homogeneous minRTO, if RTT_{real} is the real round trip time of a flow, then all flows with $RTT_{real} < b$ will suffer, as long as T is synchronized with minRTO.

This DoS attack can be characterized by its effect on throughput. Let throughput be defined as the amount of data delivered, in a period of time [4]. Normalized throughput (denoted ρ) is throughput as a proportion of the maximum available bandwidth. The reduced normalized cumulative throughput for several flows using a given minRTO under attack was derived in [16] for the standard TCP RTO calculation (Equations 1, 2, 3) and is given by:

$$\rho(T) = \frac{\lceil \frac{minRTO}{T} \rceil T - minRTO}{\lceil \frac{minRTO}{T} \rceil T} \tag{4}$$

where T is the period of the attack. Equation 4 holds for all flows where:

- $b \geq RTT_i$; and
- $minRTO > SRTT_i + 4RTTVAR_i$

The normalized throughput given by Equation 4 will be our basis for comparing different diversity implementations, and we refer to it as the *standard TCP (model)*.

Kuzmanovic and Knightly [16] present stronger evidence that the attack works and that it does so under a wide variety of environments. The *shrew* paper also explored the idea of diversifying minRTO values, but under conditions that left the authors unconvinced of its effectiveness. In the next section we show that, under the correct conditions, a randomization of some of the TCP parameters used for congestion control mitigates the effects of this attack.

4 Randomizing TCP congestion control parameters

The original *shrew* paper studied the aggregate throughput of a diversity defense where each flow chose a uniformly distributed minRTO over an interval $[a, b]$. With this approach, the randomization shifted the most vulnerable period and the maximum throughput during the period, but the *aggregate* throughput remained susceptible to the attack. In the following, we extend their work by exploring different randomization strategies and using alternative metrics for evaluating success.

Rather than using a flow aggregate, we treated each host as a unit, with all flows in a given host sharing the same minRTO, but with each host in the same LAN choosing a different minRTOs. We then measured the throughput of each host.

The experiments were done using two non-standard variants of the RTO calculation, and compared to the theoretical estimate given in Equation 4. The first was the Linux (kernels 2.4.2x) implementation, with Equation 3 changed to Equation 5, below:

$$RTO \leftarrow SRTT + max(minRTO, k * RTTVAR) \tag{5}$$

The reason for using this implementation was to allow the calculation of the "real" RTT to have an effect in the computed timeout given that most round trip times on the Internet are below the recommended value

of minRTO (1 s) [14]. This variant guarantees that, at any moment, the RTO will always be larger than minRTO, varying by the estimated average RTT time, which makes it more difficult for the attacker to find effective attack periods, even if the randomization scheme for the parameters were known. We will use the label `Kernel 1` to refer to a standard Linux kernel, using randomized minRTOs.

We modified further the Linux implementation of Equation 3 to maximize the spreading of calculated timeout values but at the same time keep a reasonably accurate estimate of the RTT. The modification is described by Equation 6.

$$RTO \leftarrow SRTT + k * RTTVAR + minRTO \tag{6}$$

We refer to the configuration with Equation 6 and the standard values of α, β and k as `kernel 2`. A third configuration, using Equation 6 but containing one sample set of modified values of α, β and k, is called `kernel 3`. Table 1 summarizes the RTO calculations to be evaluated, and the labels for the kernels which incorporate them.

Label	**RTO calculation**	α	β	k	**Source of data**
Standard	$max(minRTO, SRTT + max(G, k * RTTVAR))$	$\frac{1}{8}$	$\frac{1}{4}$	4	Model (Equation 4)
Kernel 1	$SRTT + max(minRTO, k * RTTVAR)$	$\frac{1}{8}$	$\frac{1}{4}$	4	Standard Linux Kernel 2.4.2x.
Kernel 2	$SRTT + k * RTTVAR + minRTO$	$\frac{1}{8}$	$\frac{1}{4}$	4	Modified Linux Kernel 2.4.2x.
Kernel 3	$SRTT + k * RTTVAR + minRTO$	Varies			Modified Linux Kernel 2.4.2x.

Table 1: Explored RTO variants: Column 1 indicates the name of each variant; column 2 gives the formula; columns 3 through 5 give the values of α, β and k, and column 6 explains the source of the data. For the three kernels, `iperf` was used to collect the throughput data

4.1 Parameter ranges for the randomization

Now that we have identified which parameters to diversify using randomization and modified the RTO calculations accordingly, we need to identify the range of acceptable values for each parameter.

For minRTO, we can choose integer values between 0 and 2000 ms, thus making the network minRTO average close to a compliant 1 s. Although minRTO is used differently in each of the four kernels, the result of the calculations can never be smaller than minRTO. The main difference among Kernels 1, 2, and 3 with respect to the standard model is that the final RTO is no longer trivially predictable if the RTT is below minRTO. RTO will always have value minRTO plus some γ, where γ depends on previous measurements. Requiring the average minRTO to comply with the 1 s average ensures that the system as a whole will still respect the congestion-control requirements. It is important to note that standard Linux kernels use a minRTO of 20 ms, thus being potentially bad neighbors in the presence of congesting.

The parameters α, β and k control the RTT estimate. Allman and Paxson examined the feasible ranges of these parameters [1], and we adopt their ranges, even though some of them are outside the current TCP specification. Although the range for minRTOs is chosen so that the average is the required 1 second, the other three parameters are chosen without such constraint. Table 2 summarizes the values used for each parameter.

5 Methodology

The network setup used to conduct the experiments was a four-router pipeline between two networks. The attacker induces temporary outages on one side of the pipeline. All an attacker needs to do is clog **at least one** of the (finite) queues of the routers and/or switches along the path for sufficient time to induce the loss of all packets for a given RTT.

The 2.4.27 Linux kernel sender TCP code was modified, so as to make each of the four parameters accessible and modifiable, for each of the kernel versions. The original minRTO was the easiest as it was already a constant in the code. However, α and β were hardcoded *in the implementation*, so it was necessary first to recover the original numerical form, before setting them as modifiable variables. Finally, k was used

Parameter name	Recommended	Min	Max
RTO_{min} (s)	1	0	2
α (powers of 2)	$\frac{1}{8}$	0	1
β (powers of 2)	$\frac{1}{4}$	0	1
k (integer)	4	2	16

Table 2: Parameter ranges used to generate diversity: Column 1 gives the parameter name; column 2 gives the recommended value, and columns 3 and 4 define the minimum and maximum limits we use in our experiments.

as a direct constant in the formulas, so we just found all its occurrences and replaced them with a variable. All four parameter variables were made public, and a kernel module was created to modify them as needed, using the `/proc` filesystem. A randomization is achieved by retrieving four bytes from `/dev/random`, through the internal kernel interface. Additionally, it is possible to manually modify each of the four parameters.

The attack tools were provided by Alexander Kuzmanovic, and he has made them available at his web site `http://www.ece.rice.edu/networks/shrew`. Each attack is defined by: (a) the inter-burst period, (b) the burst duration, and (c) the burst rate. The Internet experiments described in [16] were done at 10 Mbps, while these experiments were done at 100 Mbps. Scaling was performed to create an equivalent set-up. Bursts were 20 ms in length with attacks ranging from 0.2 to 1.5 s inter-burst periods. The *average* bandwidth of an attack is defined by $\frac{bR}{T}$ where b is the burst rate, R is the wire rate (100 Mbps in Fast Ethernet), and T is the attack period.

The effects of the attack on legitimate flows are measured using `iperf` between endpoints to send two different types of traffic. The first type is long transmissions of 200 s with throughput measurements scheduled each 30 s. The second type is a mix of the following simultaneous flows: a 'long' transmission of 200 s, 'medium' length serial transmissions of 10 s each, and 'short' serial transmissions of 1 s per-transmission. Each TCP flow attempts to transmit at the maximum wire rate and the TCP congestion control balances the bandwidth among all simultaneous transmissions. The reported throughput figures correspond to averages over five iterations for each attack period and type of traffic mix. The throughputs are reported in units of 100 Mbps, hence they range between 0 and 1. To conduct the attacks, a set of tools were written to synchronize senders, receivers and the attacker using an ad-hoc communications protocol.

The attacks were executed using each of the three kernel versions, with several arbitrary values for α, β and k in the case of `kernel 3`. For each kernel version, the following minRTOs were tested: $\{0.2, 0.3, \ldots, 1.5\}$, with the attack periods as described before and both traffic types were sent. For each (kernel, traffic mix, minRTO, period) the throughput obtained by each data flow was recorded. This data is analyzed using the criteria for success defined in the next section.

The next section outlines a variety of performance metrics which will be used to analyze the results in Section 7.

6 Metrics

One way to evaluate performance is to measure average throughput on the network. However, even if the average throughput is degraded during an attack, a diversity defense is effective if some individuals escape harm. That is, we are interested in measuring the survivability of the network in terms of individual throughput rather than average throughput.

This section discusses how to evaluate the performance of the randomizations described in Section 4. Because we are interested in attacks that throttle legitimate throughput, our metrics are expressed in terms of throughput, both when the network is behaving normally and when the network is under attack. Evaluating performance when the network is not under attack but running the diversity defense allows us to assess the cost of the defense itself.

If S_D denotes a system using a given defense and S_O defines the same system in its original, undefended condition, then we expect that **under attack** $\rho(S_D) > \rho(S_O)$ and that under normal conditions $\rho(S_D) \sim \rho(S_O)$, where ρ is the standard notation for normalized throughput, the amount of useful data being delivered by the system divided by the available bandwidth. In this work, we assume that S is a group of hosts in

a given network. Under normal conditions, for TCP, it holds that each node obtains an equal share of the available bandwidth because of dynamic load-balancing; hence for n hosts with similar workloads, each host obtains about $\frac{1}{n}$th of the total bandwidth. The total throughput of the network can be considered a theoretical $n\frac{1}{n} = 1$.

Under attack in a homogeneous environment, assuming that all hosts respond identically, the throughput at each host is reduced by some proportion q. Therefore, the network throughput is reduced to $n(q\frac{1}{n}) = q$. However, if each node responds differently to the attack, as in the case of the diversity defense, it is less obvious what the network bandwidth would be. We use the following approximation: If each host reduces its throughput by some individual q_i, then we assume that the *network throughput estimate* is:

$$\rho(S_D) = \sum_i q_i \frac{1}{n} \tag{7}$$

where S_D is the network. $\rho(S_D)$ can be less than, equal to, or larger than q, depending on the distribution of q_i.

Equation 7 gives a conservative estimate of network throughput under attack. When some hosts underutilize available bandwidth, the fraction available to others becomes larger. The real throughput would therefore be larger than this estimate. Unfortunately these interactions are difficult to measure or estimate, so when studying the throughput gain we will use the conservative estimate.

Another evaluation measure is based on the throughput of an unrandomized host. For the *reference throughput* (q) defined as the throughput a host would have had under attack in the absence of the randomization, the *percentage of survivors* is defined as the percentage of hosts in which the throughput under attack is larger than the throughput in the absence of a randomization. A host h_i is denoted a *survivor* if in the presence of an attack with period T:

$$\rho(h_{iD}) > \rho(h_{iO}) \tag{8}$$

Based on Equations 7, and 8, for a given attack, as defined by its period T, we define the following metrics for evaluating a randomization:

- **Average increase in throughput (gain):** $\frac{\rho(S_D)-q}{q}$;
- **Percentage of survivors:** $\frac{m}{n}$ where m is the number of hosts that comply with Equation 8 and n is the number of hosts in the network

7 Results

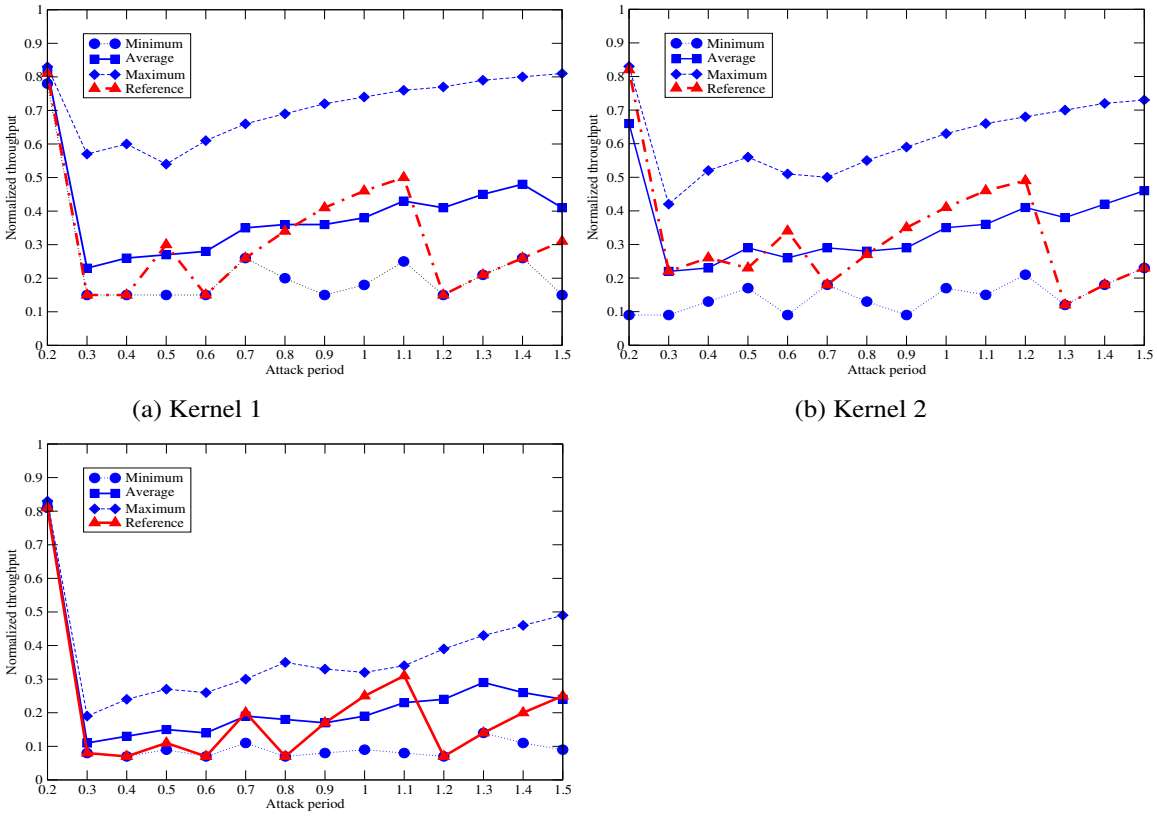
In this section we evaluate the effect of the diversity defense using randomized parameter values. We focus on minRTO randomizations, and then briefly discuss the effect of modifying the α, β and k parameters. All results shown for `Kernel 3` were obtained using $\alpha = \frac{1}{2}$, $\beta = \frac{1}{16}$, $k = 8$.

7.1 Cost

Efficiency was evaluated for all kernel variations and traffic mixes described in Section 5 (data not shown). We compared throughput with the defense in place (without attack) to throughput without the defense (also without attack). The difference in throughput was less than 1% (not shown). We conclude that the cost of the randomization is negligible in a non-congested environment.

7.2 Effectiveness at defeating attacks

Throughputs for different values of the kernel parameters were collected by running attacks against hosts in the testbed, for each of the three kernels. To compare to the standard TCP recommendation, we also show the normalized throughput as given in Equation 4. We collected data on the three kernels for nine minRTOs: 200, 250, 400, 500, 750, 800, 1000, 1250 and 1500 ms. We also used the model to show that a larger-range (0.1 to 2.1 s) randomization is also feasible. The measured throughput averages are counted towards the average throughput in proportion to their relative presence in the population. For the standard TCP model, for which we have a theoretical derivation, we plot the throughput for minRTO between 0.1 and 2.1 s.



(a) Kernel 1

(b) Kernel 2

(c) Kernel 3

Figure 1: Throughput under attack. Minimum, average and maximum throughput for the nine RTO values, per attack period, for `Kernel 1`, `Kernel 2`, and `Kernel 3`. The line with triangles shows the reference: the throughput under attack with the unrandomized standard version with minRTO=1 s.

Figure 1 shows the average, minimum and maximum normalized throughput measured for the three kernels. The reference normalized throughput (q) is also shown for comparison. The full table summarizing the percentage of survivors per kernel, per attack period is not shown due to space constraints. The randomization obtains a survival percentage larger than zero for all attack periods in all kernels. This means that with high probability there will always be at least one host able to use some percentage of the bandwidth Further, there are only four instances in which the percentage of survivors is below 20% (attack periods 0.4, 1.1 and 1.2 in `Kernel 2` and attack period 1.1 in `Kernel 3`), and no instances in which the percentage of survivors is below 10%. The network average throughput with the defense is larger (that is, better) than the non-randomized (reference) throughput for 10 out of 14 attack periods in the case of `Kernel 1`, and `Kernel 2`, and 6 out of 14 for `Kernel 3`.

Improvements are modest in terms of average throughput (under most attack periods, the kernels obtain an average throughput below 0.5), although this is still better than the average obtained using the TCP-recommended minRTO setup. However, if we consider the maximum and minimum values, a different picture emerges. As Figure 1 shows, for `Kernel 1` all maximum throughputs are above 0.5, in `Kernel 2` they are all above 0.4 and only in `Kernel 3` are they low (oscillating between 0.2 and 0.5). This means that some of the hosts are obtaining near-normal throughputs, which is precisely the goal of a diversity defense: Some members of the population survive almost unaffected. Maximum values exceed the reference throughput in all kernels at all attack periods. Even the network averages are larger than the reference in a considerable number of cases.

We plot the minimum, maximum, and average throughput that can be obtained using the standard TCP model (Equation 4) with randomized minRTO in Figure 2. The curves confirm the tendency noted in the data for the kernels, namely, that the maximum throughput that some host can obtain under attack is much

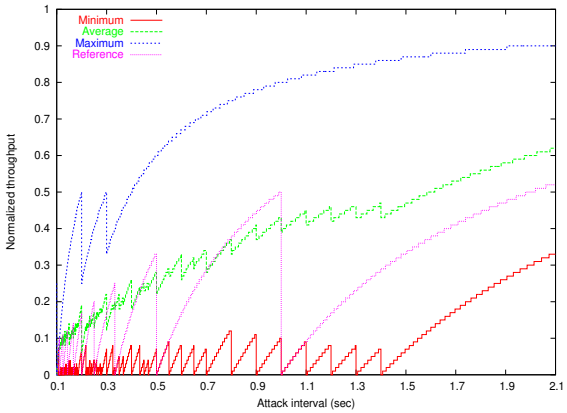


Figure 2: Standard TCP (Equation 4), discrete, normalized throughput for minRTOs ranging from 100 to 2100 ms. in increments of 100 ms. Shown are minimum, average, and maximum throughputs. The reference (nonrandomized) throughput of the network is plotted for comparison.

larger than the the nonrandomized network throughput (reference). Even the average network throughput exceeds the reference throughput for most of the attack periods. The maximum throughput is consistently larger than the throughput obtained with no randomization (reference), and the average throughput is larger than the reference throughput for most of the attack periods.

From the experimental data for the three kernels and the model for the standard TCP calculation, we conclude that randomization noticeably improves the average network throughput under attack for most attack periods, and all attack periods have at least one member which can continue transmitting with a high throughput.

At this point, we have reported data comparing the performance of a diversified system to a non-diversified one. Two issues remain: (1) Looking for an optimal minRTO value that would be more effective than the population of randomized values, and (2) Evaluating the role of the randomization of the α, β and k parameters.

7.3 Is there an optimal minRTO value?

The idea of a network populated with diverse parameter values only makes sense if there is no single parameter value that can minimize the vulnerability. If there were such a value, then it could be deployed on every computer in the network, leading to a homogeneous solution. We can identify the value 200 ms (for minRTO) as a promising candidate. However, if it were adopted as a homogeneous minRTO value for the entire network it could cause flooding in times of real congestion [1]. The same reasoning applies to using minRTO=250, 400 or 500 ms. After that there are no clear candidates, as the throughput patterns for larger minRTOs keep changing with the attack periods. Therefore, we conclude that there is unlikely to be a single minRTO optimum that could be adopted by the entire network.

8 Discussion

In Section 7, we concluded that the direct cost of randomizing parameters on throughput were negligible. However, if (for example) the entire Internet were to adopt our approach it is almost certain the modified implementations would have an impact on global Internet throughput when the network is congested. However, what the impact might be cannot be determined without extensive testing and is an important area of future work. The Allman and Paxson paper [1] observes that the congestion control algorithm is relatively insensitive to different values of the three extra parameters. And, we conjecture that this would also be true for the different kernel implementations.

Returning to the various the kernel modifications that we tested, our data show that the standard Linux implementation `kernel 1` gives the best overall performance under minRTO diversity. However, it's performance is very close to that of `kernel 2`. The difference is accounted for by the consistently larger

RTOs in `kernel 2` damping the amount of throughput delivered, as the flows have to wait longer before retransmitting. There are periods, however, where `kernel 2` and `kernel 3` slightly outperform `kernel 1`. With the existing data, it is difficult to choose between `Kernel 1` and `Kernel 2`.

On the other hand, `kernel 3` (shown for parameters $\alpha = \frac{1}{2}$,$\beta = \frac{1}{16}$ and $k = 8$ that gave the *best* performance of the attempted combinations) performed consistently worse than either `kernel 1`, or `kernel 2`. More research is necessary on this issue, as several variables were changed simultaneously. As it is to be expected, any modification that artificially increases the RTO over the real values of the measured RTTs will create a decrease in the throughput, by creating unnecessary periods of non-transmission.

A result that is not shown but worth noting is the effect of the attacks on filtered flows. As observed in [16], short-lived flows are sometimes able to use the leftover bandwidth because they either (a) start and finish the transmission in between attack bursts; or (b) having fewer total packets to send are able to finish before the long flows by smuggling an occasional packet into the network. As we observed in the mixed transmissions, this translates into actual gains in bandwidth for small transmissions, with respect to their bandwidth in the absence of attacks and when they are competing fairly with the other TCP flows. However, even these transmissions are affected by the attack, as they never manage to use all the available bandwidth. The random survival of these small transmissions argues for research into randomly restarting the congestion status of longer transmissions. This is left as future work.

9 Related work

The goal of diversity is to reduce the risk of widely replicated attacks, by forcing the attacker to redesign the attack each time it is applied. Typically, the number of different diverse solutions is very high, potentially equal to the total number of program copies for any given program. Manual methods are thus infeasible, and the diversity must be produced automatically.

Diversity methods as security mechanisms start in the 90's with Cohen's proposal to "mutate" binary code and program layout to make it more difficult for viruses to find a target [6]. Forrest et al. argued for a more general view of the possibilities of diversity for security [10], based on biological diversity.

The automated introduction of diversity in a system can be classified based on what is being adapted [8]: either the *interface* or the *implementation*. Interface adaptations work on the surface of the code being protected. They modify the layout or access controls to interfaces, without changing the implementation of the core code that the interfaces are giving access to. Implementation adaptations do not alter the interface. Instead they modify implementation of portions of the system to make them resistant to attacks.

The concept of diversity for security is currently an area of interest in public security forums [11], and in academia, where most diversification projects have explored interface randomizations ([20, 18, 3, 7, 5]). Implementation diversifications have not been researched as much, and this paper is our contribution to this class of diversity in the restricted area of communication protocol parameters.

9.1 Attacks on communication protocol parameters

DDoS attacks where the attacker sends a periodic short, high-rate burst have been mentioned since at least 2001 [9]. The *shrew* paper [16] explained one possible theoretical underpinnings of such attacks, and explored some endpoint defenses, from the point of view of the TCP specification. Luo and Chang [17] extended the theoretical research on this class of attacks ("pulsing DDoS"), and proved that a *shrew* attack is just one instance of several possible timeout attacks on the TCP congestion algorithm. Guirguis et al.[12] present a more general model for pulsing attacks to destabilize systems by pushing them into controller lock-ups using temporary overloads.

We have already discussed extensively the Kuzmanovic and Knightly paper [16]. Their article explores minRTO randomization as well, with the assumption that each flow chooses a different minRTO. The authors' conclusion, based on a derivation of the approximate average aggregate flow, is that randomization only minimally shifts and smoothes the attack throughput pattern while leaving the network still open to the attack. As this research demonstrates, by changing the perspective to individual hosts using different but fixed minRTOs, at any attack period, some of the hosts are able to successfully survive the attack and even occupy the bandwidth left open by affected computers. We also believe that a limitation of their analysis in [16] is that they examined the randomization over a fairly small range of minRTOs (1 to 1.2 s).

A more recent take on attacks inherent to the TCP state machine was proposed by Luo and Chang [17]. They identify two classes of attacks, depending on the type of false congestion signal: those that force the

victim TCP to enter the timeout state (*timeout-based*, such as the *shrew*), and the ones that make it enter the Fast Recovery state (*AIMD-based*, AIMD means Additive Increase and Multiplicative Decrease – the standard TCP strategy for window-size management), which exploits the predictability of the increase and decrease parameters in AIMD to force TCP into a slower state.

An even more general framework of attacks with "just-enough" bandwidth to cause problems but not be detected was presented by Guirguis et al. in [12]. The authors model pulsing attacks as a dynamical system. The attack depends on being able to predict when the system is about to stabilize and perturb it at just that point. Systems with many fixed parameters are possibly easier to model and thus predict.

This class of attacks do not rely on a protocol weakness as such, but on a necessary mechanism (distributed congestion control over a large, heterogeneous network) with unnecessary rigidity (fixed parameters). Kuzmanovic and Knightly [16] study some existing router-assisted mechanisms to detect and curb malicious high-rate flows, and conclude that they only mitigate but do not eliminate the problem. A detection algorithms, based on wavelets, is studied in [17].

10 Future work

A strategy of randomly resetting the congestion control state should be explored as it might confer additional benefits to the overall minRTO randomization scheme.

Although the effects of randomizing α, β and k were disappointing, they could still be useful in the context of a `kernel 1` setup for flows with large RTTs. This must be explored further.

TCP contains many parameters that are fixed arbitrarily. However, whether this is a general characteristic of protocols remains an open question. It is to be expected that simpler, lower layer, or non-reliable protocols are not as vulnerable to exploitation of fixed parameters. The identification of implicit, explicit but unnecessary, or dangerous parameters in others protocols must be undertaken, as well as global stability analyses with varying parameters, such as the one done for TCP congestion control parameters done by Allman and Paxson [1].

11 Conclusions

We used three real and one theoretical implementation of the TCP RTO calculations to argue that randomizing parameters in a protocol can have protective effects against attacks targeting protocol flaws. As long as the parameters' uniform distribution average equals the recommended or required values for the parameters, a sub-network will be protected and as a whole will still behave as a good neighbor. To demonstrate this point, the *shrew* attack [16] was used. By moving the perspective from averages to individuals, and considering the differences in throughput for each of them, it is possible to build a network that is more resilient to this attack. More specifically, we presented evidence that the randomization: (1) has a negligible cost in terms of direct execution at the host implementing the randomization; (2) improves the *average* network throughput under attack in most cases; (3) under all shrew attack periods, guarantees that there is at least one host that achieves near normal throughput; and (4) allows hosts to have a response heterogeneous enough so they can achieve larger than predicted throughputs.

We also showed that there are no **usable** (larger than or equal to 1 s) minRTO optimum for the defender, so the randomization maximizes the benefits, while (on average) preserving the minRTO IETF requirement. Slowly changing the random minRTO in the hosts will allow all of them to share the benefits of the smaller minRTOs.

Finally, we could not find support for the hypothesis that randomizing the α, β and k parameters could have a beneficial effect, (at least in the context of 'close' networks) as the modifications to make them visible to the final RTO setup made the sender response times extremely slow.

References

[1] Mark Allman and Vern Paxson. On estimating end-to-end network path properties. *ACM SIGCOMM Computer Communication Review*, 29(4):263–274, October 1999.

[2] Mark Allman, Vernor Paxson, and W Stevens. Tcp congestion control. Technical Report RFC 2581, Network Working Group, April 1999.

[3] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):3–40, 2005.

[4] Dimitri Bertsekas and Robert Gallagher. *Data Networks*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1991.

[5] Sandeep Bhatkar, Daniel DuVarney, and Ravi Sekar. Address obfuscation: An approach to combat buffer overflows, format-string attacks and more. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, D.C., U.S.A., August 4-8 2003.

[6] Fred Cohen. Operating System Protection through Program Evolution. *Computers and Security*, 12(6):565–584, October 1993.

[7] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, D.C., U.S.A., August 4-8 2003.

[8] Crispin Cowan, Heather Hinton, Calton Pu, and Jonathan Walpole. A Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *National Information Systems Security Conference (NISSC)*, Baltimore MD, October 16-19 2000.

[9] Michelle Delio. New breed of attack zombies lurk. *Wired*, May 2001.

[10] Stephanie Forrest, Anil Somayaji, and David Ackley. Building Diverse Computer Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[11] D. Geer, R. Bace, P. Butmann, P. Metzger, C. Pfleeger, J. S. Quarterman, and B. Schneier. Cyber insecurity: The cost of monopoly, 2003.

[12] Mina Guirguis, Azer Bestavros, Ibrahim Matta, and Yitong Zhang. Reduction of quality (roq) attacks in internet end-systems. In *Proceedings of the the 24th IEEE INFOCOM (INFOCOM'05)*, Miami, Florida, USA, March 2005.

[13] G J Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[14] Hao Jiang and Constantinos Dovrolis. Passive estimation of tcp round-trip times. *ACM SIGCOMM Computer Communication Review*, 32(3):75–88, July 2002.

[15] James E. Just and Mark Cornwell. Review and analysis of synthetic diversity for breaking monocultures. In *WORM 04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 23–32, New York, NY, USA, 2004. ACM Press.

[16] Aleksandar Kuzmanovic and Edward W Knightly. Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants. In *ACM SIGCOMM Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, Karlsruhe, Germany, August 25-29 2003.

[17] Xiapu Luo and Rocky K.C. Chang. On a new class of pulsing denial-of-service attacks and the defense. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'05)*, February 2005.

[18] PaX team. Non executable data pages. In *http://pageexec.virtualave.net/ pageexec.txt*, 2002.

[19] Vernor Paxson. Computing tcp's retransmission timer. RFC 2988, Network Working Group, November 2000.

[20] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *Proceeding of the 22nd international symposium on reliable distributed systems (SRDS'03)*, pages 26–272, Florence, Italy, October 06-08 2003.