Detecting Worms through Cryptographic Hashes

University of California, Davis
Department of Computer Science
Security Laboratory (seclab)

Banipal Shahbaz, Raymond Centeno
{shahbaz, centeno}@cs.ucdavis.edu
ECS 235
June 6, 2003

Abstract

Internet worms have been a problem since Robert Morris introduced the Morris worm in 1988. Since then we have seen the destructive power of which worms are capable. Code Red I and II, and Nimda are just a few examples of how worms exploit vulnerabilities not only in original applications but also those left behind by other worms. Recently, we have seen the Slammer worm spread at an alarming rate. Had the worm carried a more malicious payload, the amount of damage it could have caused is catastrophic. For these reasons, it is important to look into different methods to prevent worms from spreading. Furthermore, preventative measures need to be automated to keep up with the speed and complexity that flash and Warhol worms maintain.

Once a worm has been detected other machines on the internet need to be made aware of its presence so that they too will not be exploited. Theoretically, internet packets containing worm code can be blocked based on many things such as ports targeted, system commands executed, source or destination IP addresses, etc. However, it is not always reasonable (or possible) to block traffic on a given service/port or source IP. Furthermore, scanning each packet for machine code that exploits buffer overflows can be expensive. This paper focuses on the idea of using a system called WormDePaCk, (Worm Detection through Packet Checksums), overlapping checksums of packet information as profiles (or signatures, used interchangeably) for different worm attacks. These profiles will be broadcast to machines on the network and will be used by these machines to compare against checksums calculated for incoming packets. Packets will then be dropped based on the probability that they contain worm code.

1. Introduction (to worms and threats)

The digital age we are living in has enabled us to connect to resources that were once difficult (if not impossible) to get. For instance with a few clicks of the mouse, we can check the current weather forecast for Paris, see Mount Fuji live on a web cam, or buy rare movie memorabilia from an auction site halfway across the world. In many ways the internet has become an essential part of our lives; its existence has allowed us to use our time more efficiently.

What if the internet was unavailable for a few minutes? We would be inconvenienced a little but we could move on. However what if the internet was gone for a few hours? Or days? Imagine the chaos that event would ensue. ATMs could not dispense cash to people, businesses would lose millions in revenue, university research would grind to a halt, and this is only the tip of the iceberg.

All of this is possible if an internet worm is unleashed that causes so much damage that the network is rendered useless for a brief moment of time. The first such worm attack* occurred in 1988

*The first worm actually came from Xerox PARC in 1978. It was a program that search for idle computers on the network to install and run a program that analyzed network traffic patterns.

when the Morris Worm disabled the internet for five days [18]. Similar worms, Code Red; Nimda; and more recently, Slammer and Fizzer, have not entirely caused the internet to be offline but instead have caused people to experience a slowdown in the connection or have portions to be completely unreachable. This may only be the beginning of attacks to come. So called flash worms and Warhol worms can be capable of spreading across the internet in a matter of minutes, infecting hosts along the way and consuming massive amounts of network bandwidth [20]. Table 1 below shows the number and percentage of machines infected by different worms, as of May 2002, organized by continent.

Countries	Worm.Klez.H (since April 17, 2002)	Nimda-AO (since Sept 22, 2001)	Code Red (since Aug 6, 2001)	VBS_Love Lttr.Be (since Aug 28, 2000)	Melissa. A (since Dec 6, 1999	Total	%
North America	90,504	304,666	38	36,816	15,742	447,766	40.32
Europe	44,018	266,955	11	6,465	4,527	321,976	28.99
Asia	45,967	61,981	48	4,485	4,423	116,904	10.52
South America	14,152	1,226	0	539	3,910	19,827	1.79
Africa	7,099	110,592	0	1,666	1,791	121,148	10.91
Australia/New Zealand	6,435	71,420	3	3,955	284	82,097	7.39
Unknown	253	393	1	185	74	906	0.08
Total	208,428	817,233	101	54,111	30,751	1,110,624	100

Table 1: Source: Trend Micro, Incorporated [15]

2. Anatomy of Worms

Portions of this section from [20].

Computer worms are defined to be programs that self-propagate across the Internet by exploiting security flaws in widely used services. A worm may or may not have a malicious payload. Typically, the size of the worm may affect how quickly it can spread, thus a large payload can limit a worm's propagation. Worms with or without malicious payloads are dangerous for several reasons. First, one a large number of hosts have been subverted by a worm, they can be used to launch huge distributed denial of service attacks which can render systems and networks useless. Worms, like FIZZER, to be discussed shortly, can also be used to steal or corrupt immense amounts of sensitive data. Furthermore, worms, like Slammer, which subvert most of their victims in a matter of minutes, can be used to disrupt the network and reek havoc in general.

Before continuing further, several recent and powerful worms should be discussed.
Code Red I was a worm that was first seen on July 13th, 2001. Ryan Permeh and Marc Maiffret of Eeye Digital Security disassembled the worm to see how it works. They discovered that the worm spread by exploiting the Microsoft IIS web servers using the .ida vulnerability. Once this type of server was infected, the worm launched 99 threads which generated random IP addresses of other hosts and it attempted to compromise them using the same vulnerability. On some of the infected machines a 100th thread was used to simply deface the web server.

The first version of Code Red I had a major flaw. The random number generator used to determine which machines were to be targeted by a particular thread of the worm used the same seed for all instantiations of the worm. In other words, all copies of the Code Red worm had the same set of target IP addresses. For example if thread number 32 used x as the seed for the random number generator on

machine *Y* then thread number 32 on machine *Z* also used *x* as the seed for its random number generator. So, in fact, there were 100 different sequences of randomly generated IP addressed that were explored by the worm. This was a significant limitation on its possible spread.

On July 19th a second version of Code Red I was released which no longer had the random number seeding problem. Also, this version no longer attempted to deface the webserver on the compromised host. Instead, there was a payload added that caused a DDOS on www.whitehouse.gov. This version spread rapidly until almost all Microsoft IIS servers were compromised. It then stopped suddenly at midnight UTC due to an internal constraint that caused it to terminate. The worm then reactivated on August 1st, then shut down after a few days, then reactivated, etc. According to Lawrence Berkeley National Labs, Each time Code Red I v2 reactivated it grew in strength however; this feature of the worm is still not understood.

To estimate the number of infected machines at a given time, the "Random Constraint Spread Model" (RCS) is used. This model assumes that the worm in question has a good random number generator and a good seed value. It also assumes that the targets are chosen randomly and that once a target has been compromised it cannot be compromised again. For this model let:

N = total # of vulnerable servers which can be potentially compromised on net

K = initial compromise rate, i.e. number of vulnerable hosts that an infected host can find and compromise per hour at the start of the incident.

NOTE: ***K*** is a global constant and does not depend on processor speed, network connection, or location of the infected machine.

a = proportion of vulnerable machines which have been compromised

t = the time in hours.

To set up a differential equation note that at a particular time t a proportion of the machines has been compromised. The number of machines that will be compromised in a period of time ***dt*** is equivalent to ***Nda.***

$$Nda = (Na)K(1-a)dt$$

Thus, the number of machines to be compromised is dependent on the number of machines currently compromised multiplied by the number of machines each of those machines can compromise per unit time. Solving the equation gives us the logistic equation known to track the growth of epidemics in finite systems:

$$a = \frac{e^{K(t-T)}}{1+e^{K(t-T)}}$$

Note that as *t* approaches infinity, the limit of *a* goes to one. This makes sense because after an infinite amount of time one would expect that the proportion of all vulnerable machines that are infected reaches 100%. Also, when *t* is small, *a* grows exponentially. This represents the exponential growth experienced when a worm is spreading during its initial infection phase.

Figure 1 below illustrates an expected growth for Code Red I v2 compared to its actual spread rate for the first day of infection. The actual growth of a worm can be approximated by counting the

number of unique machines that scan a potential target in order to attempt infection. Although it turned itself off at midnight UTC, some machines with inaccurate clocks kept it alive. At its reawakening on August first, 60% of the vulnerable systems had been patched within the 11 days that it had laid dormant.

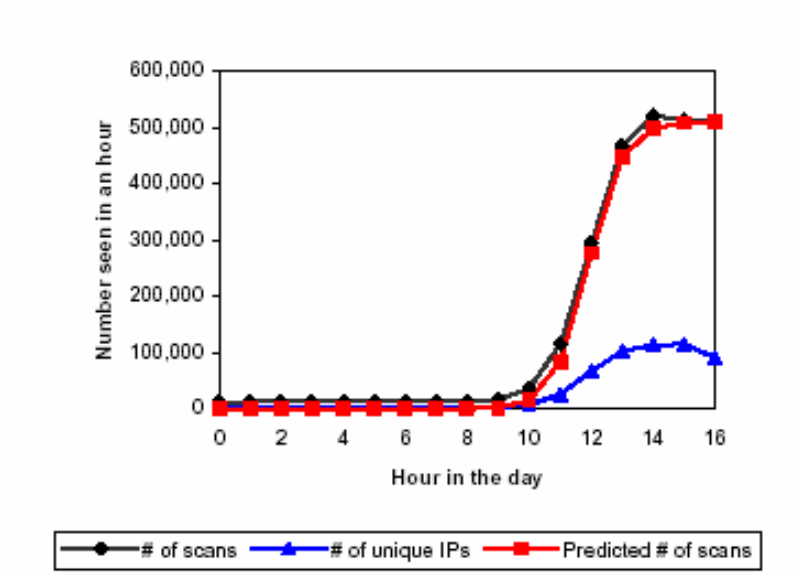

Figure 1: Hourly probe rate of Code Red I on July 21, 2001 [20]

For Code Red I v2 we see that for K=1.8 infections per hour and T=11.9 the logistic equation maps almost perfectly to the observed number of scans targeted to LBNL. Note that he incoming scan rate that a site sees is proportional to the total number of compromised servers but it is not representative of the actual amount of infected machines. The true number of infected hosts is more than that observed because it takes time for all infected machines to target a particular machine.

Shortly after Code Red I was released, Code Red II was introduced onto the Internet on August 4th 2001. Despite its name it had nothing to do with Code Red I and unlike the first Code Red, Code Red II was designed to terminate on October 1st. Code Red II exploited the same buffer overflow vulnerability in Microsoft's IIS web server. Unlike Code Red I, this worm had a payload that attempted to install a root backdoor on the victim machine. However, the worm was only successful when IIS was run on a Windows 2000 platform but on Windows NT it simply caused a system crash.

Code Red II had a much smarter propagation tactic. It used a localized attack technique. Specifically, the worm was more likely to attack machines that were physically close to it. To do so, it chose the next IP to scan using the following probabilities:

Assume current infected host IP is A.B.C.D
3/8: random IP from within the current machine's B address space
1/2: random IP from current machines A address space.
1/8: random IP from entire internet.

This technique allowed the worm to infect clusters of machines at a time and at a much quicker rate since machines with similar IP's are usually closer together topologically due to common ISPs etc. This also allowed the worm to spread quickly into private networks once it got passed the initial firewall.

The Nimda worm was also introduced around the same time. It was first encountered on September 18th, 2001. It spread rapidly and managed to maintain itself on the Internet for months. What made Nimda powerful was its ability to spread behind firewalls and also its many forms of propagation. Specifically, Nimda used five methods to spread itself:

1) Infecting web servers from infected client machines via active probing of MS IIS vulnerability.
2) Bulk emailing itself as an attachment based on email addresses gotten from the infected machine.
3) Copying itself across network shares.
4) Adding exploit code to web pages on compromised servers thus spreading to users who browse the pages
5) Scanning backdoors left by Code Red II and "Sadmind" worms.

These five methods of propagation enabled Nimda to spread so quickly that at one point in time it attempted 140 HTTP connections per second with LBNL; and in just 30 minutes after release it had a steady state of 100 HTTP connections per second. It was also able to cross firewalls through email since most firewalls rely on web servers to remove pathogens. Like Code Red I, the full functionality of Nimda is still not known.

Most recently, the Fizzer worm was discovered in May of 2003. Fizzer is an arbitrary-protocol worm which uses the KaZaA peer-to-peer network to self-propagate. There have been mixed opinions on how serious of a threat Fizzer actually is. On the one hand, the worm spreads more like a virus than a worm because of the fact that it needs user interaction to execute, thus, making its propagation very slow. On the other hand, other security analysts have claimed that it is a severe threat because of its extremely malicious payload.

Specifically, Fizzer arrives at target machines as an executable and creates copies of itself under arbitrary names in the file-sharing folder. Thus, other users on the network can download the worm and run it. Fizzer also spreads through Microsoft Outlook by scanning the addresses in the victim's address book and sends infected messages to the recipients. It also attacks random email addresses at common mail systems such as Hotmail and Yahoo.

The Fizzer worm contains a dangerous payload which performs a myriad of harmful covert functions. First, the worm installs a key logging program which records all keystrokes to a log file. It can then transmit this information to the author of the program through a backdoor utility which allows users to control the machine over IRC channels. Also, the worm frequently checks a Geocities website which contains revised versions of executable modules and performs a self update. In addition, the worm attempts to avoid detection by trying to shut down common anti-virus programs on the victim machine [13].

F-Secure, on the other hand, has labeled Fizzer as a virus rather than a worm. This is probably due to the fact that by definition a worm can propagate independent of any human interaction and Fizzer requires users to open an executable file before it can spread. F-Secure also adds that among the above mentioned threats, Fizzer contains a Denial of Service attack and an AOL backdoor which creates a bot that can be used to control the victim machine [5]. Fizzer can be removed from an infected machine and stopping propagation is simply a matter of not opening the infected files. However, Fizzer illustrates a low level of polymorphism by changing its executables, its file names, title messages in emails, etc. Fizzer also illustrates the potential power of internet worms.

This paper only covers a few of the most recent worms. As the "good guys", it is more important to know what factors make a worm propagate efficiently and quickly in order to create mechanisms to stop them. In general, the goal of a worm author is to create a worm that can spread so quickly that human intervention cannot stop it from completing its tasks. There are several methods to help speed up the

spreading of the worm. These include hit list scanning, permutation scanning, creating topologically aware worms, and internet scale hit-lists.

It is the scanning portion of the worm that can bottleneck its growth. Scanning should be very fast and should be capable of utilizing the entire available network bandwidth. A TCP SYN packet is only 40bytes and an exploit can be only a few hundred bytes, thus with modern day communication capabilities the number of scans per-second can easily reach 100. Also, worm authors need to spend a lot of time researching the vulnerabilities that they are trying to exploit. If the vulnerability is not widespread then scanning will only yield a small number of targets which leads to a small *K* value in the logistic equation. So, to be successful, the author must spend time finding vulnerabilities that will lead to a fast spread of the worm.

After launch onto a network it takes a while for a worm's spread rate to get up to speed. This is because the target address space that the worm must deal with is large and since there is only one copy of the worm on the network, it must attempt to scan all of these addresses by itself in order to find a vulnerable host. Even though a worm spreads exponentially at first, the time it takes to infect the first 10,000 hosts dominates the infection time of the entire worm. Hit list scanning is a divide and conquer method that remedies this problem. Hit list scanning requires the author to first find ten to fifty thousand potential vulnerable targets ahead of time and include them in a "hit list". The parent worm is given this hit list and attempts each target sequentially. Once a target has been found and successfully infected, the hit list is then split between the original parent and the new child worm on the compromised system. Even if the hit list starts as a large overhead payload, it quickly reduces itself to a negligible size since it is divided in half each time a new host is infected. Furthermore, hit lists do not have to be perfect; not all machines on the list need to be vulnerable machines at the time of launch. The hit list simply increases the probability of finding a vulnerable machine during the initial phases of a scan.

There are several methods to obtain hit lists. Even if a worm author performed a port scan on the entire internet, the odds that he/she raises suspicion are very small. However, if the author wants assurance that he/she remains hidden, a stealth scan is the solution. A stealth scan is a random port scan over a period of a few months. Thus, spreading the port scans out over a period of time makes the author's activity a little less suspicious. One problem with stealth scans is that once the worm is ready to be released some of the information on the hit list will be out of date due to hosts performing system updates and vulnerability patches.

Distributed scanning is a data collection method which is similar to a distributed denial of service attack. To perform a distributed scan an attacker uses a few thousand already compromised machines as "zombie" machines to perform the scans and report the information. The information from the zombie machines is then coalesced into one large hit list. Other methods of determining the location of vulnerable machines include DNS searches, spiders or web crawling techniques, and public surveys on vulnerable applications. Finally, one of the most effective methods of finding vulnerable hosts is to simply listen for information on the network. For example, Code Red I gave away over 300,000 potential targets because any machine that performed a scan for the vulnerability must itself have already been compromised meaning it too suffers from the IIS vulnerability. So, in order to determine which machines should be targeted one simply needs to log the IP addresses of the machines performing the scans.

Once the hit list phase of the worm infection terminates, the worm continues to spread using random scanning. However, scanning random IP addresses is not very efficient thus permutation scanning can be used. The motivation behind permutation scanning is that random scanning can be inefficient because many addresses can be scanned more than once. Permutation scanning assumes that a worm can detect whether or not a target has already been scanned and if so move on to scanning a new target. In a permutation scan all worms share the same random permutation of IP address space to attack. This permutation is created before the worm is launched.

Once the hit list phase spreads the worm, each spawned worm starts scanning according to a randomly assigned location in the permutation scan. It sequentially scans the locations in its section of the address space and when it sees that a machine is already infected, it then chooses another random point in the permuted list and starts from there. A partitioned permutation scan is a variation of the standard permutation scan in which each version of the worm has its own unique initial range on the permuted list to cover. Once it is done covering that range it then switches over to a regular permutation scan and chooses a new random place in the list to start scanning.

The permutation scan technique has one small weakness which slows it down. If a machine responds to a scan as if it has already been scanned then it protects *n* machines which follow it in the permuted list. This is because once a worm sees that the machine is infected it jumps to a new address to scan. The *n* machines are protected for a short period of time but can be attacked later when a strain of the worm jumps to their particular location in the list. This problem can be remedied by the attacker if the condition to jump to a random location is dependent on more than one encounter of a previously infected host. Combining permutation scanning and hit lists creates Warhol worms which are capable of infecting all of their potential targets within 15 minutes to an hour. By the time the entire network of vulnerable machines is compromised only a few worm strains remain active.

Topological scanning is an alternative to hit list scanning. This method uses information found on the already compromised host to determine who the next targets will be. Email worms are examples of topological scanning worms since they get their targets from the addresses in the user's emails. These worms sometimes need outside human intervention to spread and are occasionally classified as viruses.

Another variant to hit-list worms are known as flash worms. These worms would be able to infect a majority of susceptible machines in seconds. Rather than attempting to blindly scan the internet for vulnerable machines, flash worms attack addresses on a predetermined list. The first worm divides the list in n blocks, attempt to infect the first machine of each block, and then hand the list to a child worm to infect further. An optimized worm would begin to attack machines on the list as the rest of the list is being downloaded. A variation on this method would be to store the list on a high-bandwidth network server and give child worms the address to download its subset list, rather than sending it.

The spread rate for flash worms would be limited by one of two factors. First, worm code tends to be small (Code Red I was about 4KB in size, Slammer was 376 bytes), so the list would be significantly larger than the worm. So it propagation time would be constrained by the bandwidth of the network; in particular, the time it would take to transmit the list to child worms. A second limiting factor is the latency to further infect machines as the levels of child worms increases. With the possibility for exponential growth, the bandwidth again may limit a worm's growth.

Stealth worms follow the notion that "easy does it." Instead of infecting millions of computers in minutes (or seconds), stealth worms attempt to spread as slowly as possible. These worms are difficult to detect because they do not trigger alarms for unusual network communication patterns. One example is a worm that has attacks a pair of exploits, E_s (exploit on a web server) and E_c (exploit on a web client, or browser). Suppose the attacker plants the exploit on the server. As people browse the webpage the attacker checks to see if the user's browser is vulnerable to E_c. If so, the attacker infects the browser and sends E_c and E_s. When the user visits another website, the E_s exploit looks for vulnerable servers to send E_s and E_c.

A second type of stealth worms are typically found in P2P networks, such as KaZaA or Grokster. These worms attach themselves to files being transferred and so piggyback to other machines for possible infection or use them as a launching pad for a future attack. Since the worms would be small in size, the user would not be able to distinguish between a 4.1MB music file and a 4.13 infected music file. The file

size discrepancy is less evident with larger files (like movies). Considering that there have been over 230,309,616 downloads of the program, making it the most downloaded program in history [11], there are many machines sharing gigabytes of files for stealth worms to utilize.

The Slammer worm (also known as Sapphire) infected the internet on January 25, 2003. According to researchers, it infected nearly 90% of all vulnerable machines within 10 minutes and achieved its full scanning rate of 55 million scans per second after three minutes [16]. Slammer exploited a buffer overflow problem in Microsoft SQL Server or MSDE 2000 on port 1434. This vulnerability was discovered nearly six months prior to the outbreak and a patch was released by Microsoft. However many systems did not update their systems and were susceptible to the exploit.

Slammer did not have a malicious payload; in fact, it was only 376 bytes large (404 including the packet headers). Since it utilized UDP, rather than TCP, the worm was able to send packets as fast as the bandwidth permitted. The worm did not have to initiate the required handshake in TCP. Its method of propagation was through random scanning. As seen in figure 2, initially it conformed to RCS trends; but as machines were being infected and were attempting to infect the same machines, saturating the network, the number of probes per second quickly leveled off.

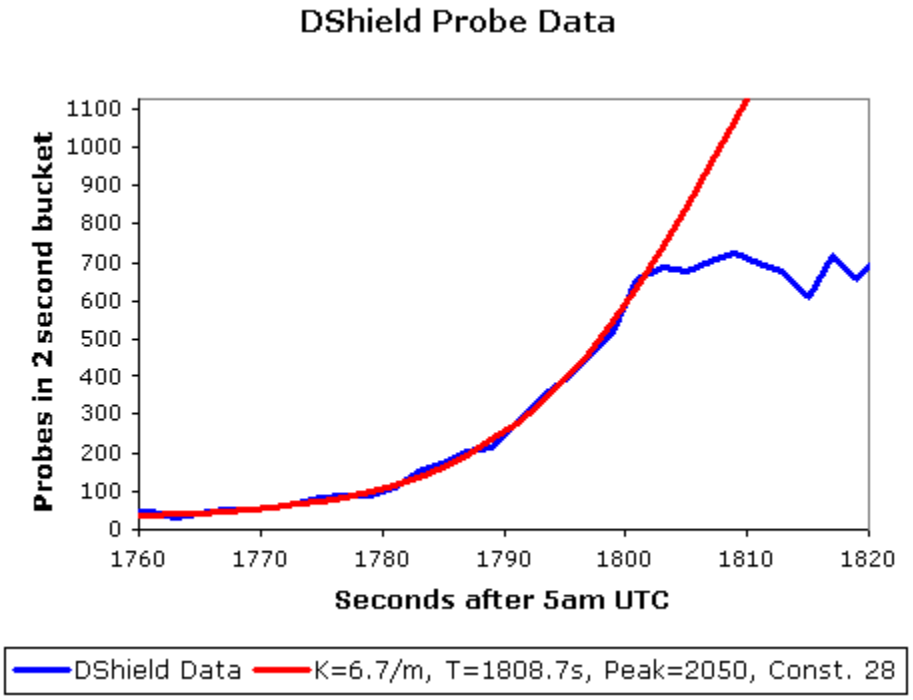

Figure 2: Source: DShield dataset of Slammer probes [16]

It is important to group similar worms together when they are being researched. To help do so, a significant amount of work has been done to classify worms. Worms can be classified according to two different characteristics. The first is the transport mechanism used by the worm to propagate, and the second is the method in which the worm is actually launched onto a system.

There are two known methods of worm transport. The first is through electronic mail. There are two types of e-mail worms: Native and Parasitic. A native email worm is one which is created using the native scripting language of the e-mail system being used. A worm of this type is carried in a proprietary form when a message is sent. An email worm of this type can exist only on the e-mail system on which it was designed to operate and cannot function outside of the designated system. At the time of Nachenberg's publication, there had been no known public native e-mail worms.

The second type of e-mail worm is one which "leverages the transport capabilities of an e-mail system to spread itself" [12]. This is known as a parasitic e-mail worm. A worm of this type can exist outside of the email platform and simply use the e-mail platform as only one of its methods of spreading. Nimda (described earlier) is an example of a parasitic e-mail worm which used five different methods to spread itself.

An arbitrary protocol worm can spread itself using one or more non-email based protocols. For example, a worm that uses FTP to transfer itself from host to host would be an arbitrary protocol worm. Peer-to-peer worms are also considered arbitrary protocol worms. Fizzer was in arbitrary protocol worm which spread itself using the KaZaA peer-to-peer network. Fizzer can also be categorized as a parasitic e-mail worm since it uses Microsoft Outlook as another method of propagation.

Aside from worm transport classification, worms are also classified according to the method used for their launch. Self-launching worms are capable of spreading to other systems without any human interaction. These worms exploit a vulnerability on the victim machine to be launched. For example, worms which use a back door to execute are typically self-launching worms.

User launched worms require human intervention in order to propagate. For example, the Fizzer worm required a user to actively open an infected file for the worm to then scan an address book and self replicate in the KaZaA shared folder. A combination of user launched and self launching worms is called a hybrid launch worm.

According to Nachenberg, there are three technological trends which are mainly responsible for simplicity and feasibility of computer worms: infrastructural homogeneity, ubiquitous programmability, and increasing connectedness via a homogenous communication mechanism

It is the homogeneity of operating systems and communication protocols that enables worms to spread easily. More than 90% of the world uses some flavor of a windows operating system. Also, most of the world uses common communication protocols such as SMTP and FTP. Furthermore, there are common applications used that run these protocols. Thus, a worm author needs to simply find vulnerabilities in a commonly used operating system (i.e. Windows 9X, NT, XP, 2000 etc), a common flaw in a protocol, or vulnerability in a common user application such as Microsoft's Outlook email client. In fact, over 99% of all computer viruses are designed for some Windows platform [12].

Another enabler of simple worm programming is that of ubiquitous programmability. For example, the macro feature of the Microsoft Office suite allows malware authors to create worms that spread from one office document to another in less than 50 lines of code. This is because macros can have the capability to copy themselves from one document to another. The danger lies in the fact that macros have access to other components of the Windows operating system. Thus, allowing the worm to utilize, say, an email client such as Outlook in order to spread itself. Furthermore, Microsoft's Component Object Model (COM) interface makes it easy for worm authors to interface with other Windows programs and the rest of the system. Combining macros with COM enables worm authors to create highly powerful worms that can manipulate the entire host system.

Finally, an increased connectedness using homogeneous communications allows worms to spread over the internet at an extremely fast rate. Cable connections and Digital Subscriber Lines aid in this interconnectivity of the internet. Also, by having a constant connection to the internet, users are able to have constantly connected applications running. These applications make great targets for worm authors and make worm propagation easier and quicker.

It is important to note that even computer viruses were rudimentary and simplistic at first. However, over the years, virus authors have incorporated many tricks into their programs. The same is true for computer worms. In other words, six to ten years ago worms were fairly simplistic in nature. It is a safe bet to assume that they, too, will follow the evolution of computer viruses and incorporate new tricks to defeat security. Nachenberg comments on "second generation worms". These are worms that are difficult to remove, have polymorphic capability, attack antivirus or anti-worm software, or spread on wireless systems. Of these second generation characteristics, this paper is mostly concerned with polymorphism.

There are several types of polymorphic changes that can be made within a packet containing worm code in order to confuse a worm detecting system. The first uses polymorphic engines within a data packet to encrypt worm code with a new and random key before each time it is sent to infect new hosts. The second type of polymorphism occurs when padding is used within a packet to hide worm code. If the padding is changed between infections then the packet will appear to be different by some percentage. This also includes random data insertion to change the overall length of a packet. This random data could be extra characters in the padding section or NOPs in the code itself.

A polymorphic worm is similar to a polymorphic virus. A polymorphic virus is an encrypted virus which contains a decrypting routine and a mutation engine. In other words, each time a polymorphic virus is loaded, the decrypting routine first decrypts the virus, the virus then executes, and at some point in time, the mutation engine creates a new encrypted version of the virus along with a new decrypting routine to pass on to other files that are to be infected. Thus, the resulting virus is not only encrypted, but also different from the original virus. This makes scanning for the virus difficult since no single signature can match the continuously changing sequence of bytes of which the virus is composed.

The problem is further inflamed by the widely distributed mutation engines. Once an attacker creates an efficient engine, it is common for him/her to pass it on to other would-be attackers. There are two popular methods for finding polymorphic worms: using Generic Decryption, and using CPU heuristics. Both of these methods are useful but have a few shortcomings. Both rely on actually executing the virus.

Generic Decryption uses a virtual machine to test all suspected viruses. It tests to see if the executable modifies itself. If it does, it is likely that the self modifications are attributed to the decryption stage of the virus. If a virus is found, since it is running in a virtual machine, it cannot cause any damage to the system. The problem is that all executables need to be tested on a virtual machine first. This is not practical and is not fool proof.

The heuristic method is similar to Generic Decryption except instead of monitoring the data in RAM it monitors the calculations and results that the CPU performs and uses. For example, a polymorphic worm may take part in performing many useless calculations from which it uses none of the results. It would do this in order to make itself appear as a clean program. In other words, by adding useless computation instructions it is trying to add information to its signature that has nothing to do with the virus itself. Unfortunately, it is difficult to keep track of the rule set that monitor virus activity according to heuristic values. For example, adding x new rules may result in losing the capability to detect n previously detectable viruses [22].

The worm polymorphism problem is even more difficult to solve because WormDePaCk attempts to detect worms at a packet level. Thus, a virtual test environment approach would not work as it did for polymorphic viruses.

3. Proposal

We want to build a system that attempts to analyze every packet for malicious content. If we had the computing power, we could open each packet and look for data that we know is malicious. However as networks become increasingly fast reaching gigabit speed, it is difficult to examine packets completely. The way we attempt to solve this program is to use checksums which represent the content as the means to packet inspection. WormDePaCk, Worm Detection through Packet Checksums, uses a series of checksums to identify and find similar packets.

WormDePaCk maintains a database of checksums of packets known to be worms or contain worm code. All future incoming packets are checksummed and compared to the database. If the ratio of checksum matches is greater than the threshold allowed by the system, an alert is raised and neighboring hosts are notified. The system informs them of possible checksums to look for, as well as the suggested threshold value. If more packets arrived matching a known worm, the threshold level can increase (with increasing confidence) and more checksum values can be sent as "wanted posters."

One problem mentioned earlier is worm variants. These will probably create checksums that differ from the original version of the worm. WormDePaCk attempts to overcome this problem by utilizing overlaps in the checksums. If variants exist, we hope that a majority of the packets will remain constant (it could be the actual propagating code). The overlaps ensure that certain parts of the data section are processed more than once using different blocks. This is also the reason behind the threshold level.

Related to the threshold is the paranoia level. The paranoia level is inversely related to the threshold level. The higher the paranoia level, the more a system believes packets are malicious; therefore, the threshold level should be low to alert on minor matches. A benefit of having a paranoia level is a possible new use for honeypots. These machines would have their paranoia level set at a high number so that it will "detect" more worms and so will scrutinize the packets more carefully.

WormDePaCk implements some safeguards against attacks. It is possible that a misconfigured system wanting to send worms could send bad data to its neighbors. It could set the threshold level at 100% meaning that the entire packet would have to match to trigger an alert. Then the attacker would modify the packets slightly so as to not lose any functionality, but cause the checksums to be slightly different. Therefore WormDePaCk does not allow rules that invalidate previously issued signatures or cause neighbors to ignore alerting hosts.

4. Related Works

4.1 IDS

Network intrusion detection systems, or NIDS, monitor packets on the network wire and attempts to discover if a hacker/cracker is attempting to break into a system (or cause a denial of service attack) [4]. An alternative is a host-based IDS which attempts to discover attacks on a particular system, or host. There are two main types of IDSes: anomaly-based IDSes, looking for unusual patterns or behaviors; and signature-based (or rule-based) IDSes, finding activities which violate a predefined set of rules.

4.2 Symantec-Mantrap

There is a belief that "knowing is half the battle." In other words, understanding what attackers do will allow us to protect the systems better. Using honeypots, Internet-attached server that act as a decoy, luring in potential hackers in order to study their activities and monitor how they are able to break into a

system [8], is one method that is used to gather information about attacks. An alternative approach used by Symantec in its ManTrap product is to use cages, which provide attackers with systems that could exist in a production system, but does not affect the real network if they are compromised [23].

When ManTrap is installed in a system, the system administrator is able to include real-world information such as employee names, web pages, etc. Further the information and data files are organized on the system as on a production system. In fact, an attacker would not be able to distinguish between a ManTrap cage and a real system. From the system administrator's point of view each cage is a subdirectory of the host machine, whose root directory has been changed using the chroot command. From an attacker's point of view each cage is a separate machine with its own IP address and subdirectories that he/she is able to browse.

As the attacker peruses the system, ManTrap is logging all the activity. ManTrap attempts to log all keystrokes made by the attacker, as well as all processes invoked directly on a shell or called by binaries installed by the attacker. The logs are used to try to identify the attacker's motive and to fingerprint an attack to a person. These logs are protected from modification by the attacker. In fact, the attacker does not know they exist.

When an alert is issued ManTrap uses two protocols, SMTP and SNMP. There are four levels of alerts:

Level 1: Messages sent only if an attacker sends a packet to the network from a cage, opens a new session, or causes a process to be run on the host as root

Level 2: Messages sent in situations that indicate a possible security compromise, such as a cage receiving a packet, a process is executed, or keystrokes are recorded

Level 3: Messages sent when the cage experiences any traffic

Level 4: User customizable alerts

4.3 Thumbprinting

Since the goal of this project is to detect worms on a packet by packet level, designing an algorithm to create worm signatures is non trivial. The task of creating signatures for worms is complicated with the introduction of worm polymorphism and packet fragmentation across links. Polymorphism prevents the creation of a single signature to identify a specific worm strain. Similarly, data fragmentation introduces difficulty in matching of a signature for a specific packet to the same data which has possibly been fragmented at some router.

After some time, research unveiled a similar task, tracing remote logins, which faces some of the same problems that occur in signature generation. "Thumbprinting" [19, 21] is a method used to trace the origin of intruders and was designed by Stuart Staniford at UC Davis in the early 1990s. Thumbprinting relies on the fact that the content over an extended connection is the same over all of the links. Its relation to worm signature generation will be addressed shortly, but first, a quick overview of how thumbprinting works is required.

The idea behind thumbprinting is to monitor every link of a network by taking time spliced thumbprints of the information traveling across the link. Thus, once an intrusion has been detected, the thumbprint on the immediate incoming link is then compared with the thumbprints taken on all neighboring links from the adjacent machine. The thumbprint which matches the suspicious one will then be followed, and again, the neighboring links of the machine examined will have their thumbprints compared to the one in question, and so on until the originating host is found.

To create a thumbprint, a substrate, a vector, and a thumbprinting function are required. For example, if one wanted to determine the number of times "slammer" occurs in a given sequence, one would need to define the substrate as [0,1,2,3,4,5,6,7] and the vector as {'s','l','a','m','m','e','r'}. The thumbprinting function would then take a value of 1 on the sequence given by the vector and zero otherwise. Similarly, the number of times the character 'x' is followed by seven arbitrary characters and then a '?' would be represented by the substrate [0,8] and the vector {'x','?'}. The sum of the thumbprinting function over a given interval of time would then be considered its thumbprint. It is important to note that the results of the thumbprints for consecutive time intervals is additive. This helps resolve problems relating to clock skew across networks, packet fragmentation, and bottlenecks in the network.

Determining what function to use is one of the more difficult questions that needs to be resolved. Staniford uses principle component analysis to answer this question but is out of the scope of this paper. It is suitable for our purposes to simply accept the thumbprinting function as a pattern recognizer using the substrate and vector as input.

Thumbprinting is related to worm signature generation because it addressed many of the same problems faced when dealing with worm signature generation and comparison. First, just as WormDePaCk has the capability to create several different checksum digests using different algorithms; Staniford proposed having several thumbprinting functions to alleviate any discrepancies encountered when deciding which link to trace. This type of redundancy in worm signature comparison would help in the case where a polymorphic worm was to fool one of the checksum functions. After further study, it was determined that this was highly unlikely thus only the CRC32 checksum function was used, as described later.

The main type of intrusion examined by Staniford is that of a remote connection. Thus, all the packets in the TCP session contain only one byte of information representing a keystroke. Thumbprinting examined the problem of padding packet data with useless information. Unfortunately, Staniford did not address a solution to this problem. Similarly, he did not address the problem of encryption of data at any point along the connection chain. WormDePaCk cannot ignore this possibility since polymorphic worms will most likely use encryption engines in the near future. Staniford also attempted to use the checksum idea but explained why such a solution would fail due to lack of the additive feature and robustness. This made creating a thumbprint function for his algorithm a little less complicated.

4.4 Cryptographic Hash Functions

Cryptographic hash functions ($h : A \rightarrow B$) are defined to be functions that have the following properties [2]:

1. For any $x \in A$, $h(x)$ is easy to compute.
2. For any $y \in B$, it is computationally infeasible to find $x \in A$ such that $h(x) = y$.
3. It is computationally infeasible to find $x, x' \in A$, such that $x \neq x'$ and $h(x) \neq h(x')$.

Hash functions map messages and data in A to hashes, or checksums, in B. In general the range of B is considerably less than the range in A. This means that multiple items in A will map to the same item in B. The best cryptographic function tries to evenly distribute the hashes. There are many different cryptographic functions that are currently being used. The following are three cryptographic functions used to create checksums or message digests: MD5, SHA-1, and CRC32.

4.4.1 MD5

MD5 was developed by Professor Ronald L. Rivest of MIT. What it does, to quote the executive summary of rfc1321, is:

[The MD5 algorithm] takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target message digest. The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA [1, 14].

The message digest is calculated as follows [14]:
Suppose we have a b-bit message as input written: $m_0, m_1, \ldots, m_{b-1}$

1) Pad the message with a single 1 followed by 0s until the length is 64 bits less than a multiple of 512.
2) A 64-bit representation of b is appended to the padded message. Now the message is an exact multiple of 512, and is an exact multiple of 16 (32-bit) words. Let M[0, 1, …, N-1] be the words, where N is a multiple of 16.
3) A four-word buffer (A, B, C, D) is used to compute the message digest. The buffer is initialized as follows: (in little-endian)
   A: 01 23 45 67
   B: 89 AB CD EF
   C: FE DC BA 98
   D: 76 54 32 10
4) There are four functions defined that take three 32-bit words as inputs to produce one 32-bit word
   F(X, Y, Z) = XY v not(X) Z
   G(X, Y, Z) = XZ v Y not(Z)
   H(X, Y, Z) = XY xor Y xor Z
   I(X, Y, Z) = Y xor (X v not(Z))

   Follow the algorithm given in [14] to produce A, B, C, D
5) Output A, B, C, D (in little-endian)

4.4.2 SHA-1

SHA is short for Secure Hash Algorithm and is described in the Federal Information Processing Standards (FIPS) Publication 180-1 [6]. It is mainly used to create a compressed version of a message or data. SHA produces a 160-bit message digest. This digest is then used as the input into the DSA (or similar algorithm) to produce a signature. Using the digest is more efficient than using the original message because the digest is usually shorter in length and thus is easier to compute the signature. In order to verify a message, the recipient uses SHA to compute the digest of the received message, runs it through DSA (must be the same algorithm used by the transmitter), and compares the signatures.

The algorithm is defined in [6] and follows similar steps as in MD5. The message is padded in a unique way to be a multiple of 512. SHA-1 processes the message in 512-bit blocks. Similar functions and constants are defined that are used in the algorithm to compute the digest.

4.4.3 CRC32

CRC is short for cyclic redundancy checking and is a method of checking for errors in data that has been transmitted on a communications link [3]. It appends a few bits to the end of a message and sends the extended string. In CRC32, the length of the appended bits is 32. The receiver of the string can then perform a calculation which if not zero would indicate that one or more bits of the message is corrupted [9]. Matloff cites Peterson and Davie that have said that 32 bits are enought to give a certain level of protection against errors in messages that are 12,000 bits in length.

The following is how the CRC is computed. The following is taken from [9].

Let M be the message we wish to send, m bits long. Let C be a divisor string, c bits long. C will be fixed, while M (and m) will vary. We must have that:

- m > c-1
- c > 1
- The first and last bits in C are 1s

The CRC field will consist of a string R, c-1 bits long. Here is how to generate R, and send the message:

1) Append c-1 0s to the right end of M. They are placeholders for the CRC. Call this extended string M'.
2) Divide M' by C, using mod-2 arithmetic. Call the remainder R. Since we are dividing by a c-bit quantity, R will be c-1 bits long.
3) Replace the c-1 appended 0s in M' by R. Call this new string W.
4) Send W to the receiver.

5. Implementation

WormDePaCk is implemented as a host-based system for this prototype. Future versions may be a combination of host-based and network-based systems, but further research is needed to see its feasibility. The current implementation of this project includes a signature generator and a comparator. Network connectivity of the scheme will be discussed in future work.

The signature generator is created using C and C++ and uses the mhash library [10] to produce the CRC32 checksums of blocks of data. The files *checksum.h* and *checksum.cc* can be found in Appendices A and B, respectively. The packets are represented as files on the system. The "packet" is opened and a CRC32 checksum is computed for each block of data by using the *getData()* and *compHash()* functions in *checksum.cc* and objects and methods in the mhash library. The blocksize and overlap can either be predefined by a system administrator or can be dynamically modified as circumstances change. The checksums are collected in an array to be processed later by the comparator.

The comparator is created using C++ and uses the files in the signature generator and *compare.cc* (Appendix C), *QuadratricProbing.h* (Appendix D), and *QuadraticProbing.cpp* (Appendix E). The QuadraticProbing files come from Weiss [24] and provide the hash table implementation used as a database for the checksums. In addition, the file *CPUTimer.h* (written by Ted Krovitz for Data Structures class and found in Appendix F) is used to compute the CPU time for the comparator program. It will be used to estimate the time needed to compute and/or compare the checksums to the database. When the comparator module begins, the threshold level is set by the paranoia level. In this implementation **threshold** = 100 – **paranoia**, and is clamped by 0 and 100. There may be a better relationship between **threshold** and **paranoia**. The original array of checksums is entered into the database by using the hashtable.*insert()* function of the QuadraticProbing hash class. Then CPU time is reset and the array of checksums of the packet to be compared is searched in the database using the hashtable.*find()* function. If a match is found, a variable is incremented. At the end of the array, the ratio of checksum matches to original number of packets. If the ratio is bigger than **threshold**, then a rule is generated and would be sent to other systems. At the same time, the computation time is displayed.

6. Results, limitations, and future work

In general, the idea of using overlapping checksums to determine variations in packet data seems sound but it relies on a few assumptions. First, the algorithms and data structures used are rather rudimentary. Hashing, storing, and comparing large amounts of signatures require a lot overhead. It is assumed that this implementation of WormDePaCk can be easily adapted to run efficiently and without much overhead on standard modern day desktops, servers, etc. At this point in time, the algorithm is being designed to run on machines which contain roughly one thousand times the computing power of regular desktops. However, if this scheme is to be successful it is important that it be able to run on the average personal computer. To do so, future versions of the program should utilize better storage and comparison algorithms. Also, in order to help alleviate some of the computational overhead required, it is interesting to look at the possibility of having smart network interface cards (NICs) which could perform many of these calculations ahead of time.

Second, it is important to realize that this is not a tool used to discover new worms WormDePaCk is designed to work in conjunction with some other method of discovering worm activity. Thus, the effectiveness of this scheme is highly reliant on having an initial notification of worm activity. This notification can simply come from a machine that has already been compromised. Backtracking activity could uncover what caused the compromise and then use that information to generate a signature. Again, there is a lot of "wishful thinking" and generalization of hypothetical situations but if preventing worms from spreading was a simple and easily implemented task, this paper would simply be a survey on the methods of doing so.

The results obtained from the current version of WormDePaCk simply reflect the performance capabilities of the checksumming scheme and comparator as described above. The first step toward developing an efficient signature system was to decide on a signature generating function. As stated earlier, CRC32 was chosen because of its small digest size and its accuracy. Second, tests were run to determine a block and overlap size such that the total size of the signature is not too large yet the signature still yields a somewhat fine granularity when determining the modified bytes of the data packet. To perform the test, several files were created of arbitrary sizes. Then the data of each of the files was slightly manipulated to represent a different packet which contains mostly similar data to that of the suspicious file.

The files were then analyzed using the code written in *chks.c* (source found in Appendix G) and the PERL script *bufftest.pl* (source found in Appendix H). The output was a table representing possible block sizes of five to forty in increments of five. This data can be found in Appendix I. In addition, each block size was tested with an overlap which ranged from zero to just under one half of the current block size used. These tables were then analyzed to determine which block size and overlap combination yielded a good indication of data being changed yet at the same time did not require an extremely large signature. Of course using a block size of five with zero overlap would give the best indication of which bytes of data differ from the original file but the signature size of this combination is simply too large to be practical in future use. A scan of the table for a percentage change that is small combined with an overall signature size which is not too large reveals that using a block size of thirty with an overlap of ten is a good choice. Of course each file tested is different, but on average this was a good choice.

Tests were also run to determine the amount of computation time required to perform signature database checks. However, the amount of test data used is far less than that what would be necessary to form a complete worm signature database. This prototype utilized files that were 2724 bytes in size. The blocksize is set to 30 bytes with an overlap of 10 bytes, as explain earlier. With full debugging enabled which displays all work along the way, the time to compute is 0.02 seconds. Future work on this project includes creating a true signature database, possibly using actual worm packets. Also, this very basic

implementation of our protocol does not employ host communication. It is planned that future versions of the project will be capable of exchanging new signature entries and performing self database updates.

One main limitation faced when using checksums in this scheme is that of discrepancies found when data is inserted into a packet. In other words, if a signature was formed for the sentence:

1) A boy took his dog for a walk in the park on a sunny afternoon.

Our scheme would be able to detect changes such as character replacement as follows:

2) A b0y t093&%dog for a walk in the ~8m on a sunny afternoon.

But would require more computational complexity if the changes included the addition of random padding such as:

3) 1234A boy took his dog for a walk in the park on a sunny afternoon.

This results from starting the checksum blocks at the first character in the initial unaltered sequence. For example, if the block size was ten then "A boy took" would be in the first block of the signature. But the first block for the signature of 3) would include "1234A boy ". This throws off the entire array of checksum digests resulting from the signature generation for 3). Thus, to fix this for a signature scheme using a block size of n, there would need to be n different signatures generated. The first signature would discard zero characters from the front of the input before calculation, the second would discard one character, the third would discard two, etc. This complicates the entire algorithm and needs to be remedied in the future.

Other future work includes utilizing special locality of matched signature blocks along with the total percentage of matched blocks to determine whether or not a packet is malicious. In other words, certain blocks of the signature, such as those pertaining to a buffer overflow, would be flagged as critical. Thus a matching of a critical block influences the total likelihood of witnessing a malicious packet. Work is also being done to determine the presence of machine code within the data portion of packets. This has also proven to be a difficult problem to solve.

7. Conclusion

This paper would not be complete without mentioning the recent announcement of a malware writing course offered for fall of 2003 at the University of Calgary which, among other types of malware, teaches students how to make worms. Of course, the class is for educational purposes regarding computer security but the fact that this course exists is a testament to the prevalence of worms and viruses.

Although the problem of worm propagation will not be solved easily, it is important to explore different and creative methods to stop worms from infecting other machines. It seems simple to design a worm intrusion detection system when computational power is great such as that provided by the Cloud Shield machines. However, designing a scheme that will run efficiently without excessive overhead on modern day desktops complicates the matter. To make WormDePaCk work effectively, it is important to find the right combination of checksum algorithms, communication protocol, hardware integration, etc.

8. References

[1] Abzug, Mordechai. MD5 Homepage (unofficial). http://userpages.umbc.edu/~mabzug1/cs/md5/md5.html.

[2] Bishop, Matthew. Computer Security: Art and Science. Addison Wesley. September 2002.

[3] CRC-a whatis definition. http://whatis.techtarget.com/definition/0,,sid9_gci213868,00.html

[4] FAQ: Network Intrusion Detection Systems. http://www.robertgraham.com/pubs/network-intrusion-detection.html

[5] F-Secure. Fizzer. http://www.f-secure.com/v-descs/fizzer.shtml

[6] FIPS PUB 180-1. Secure Hash Standard. http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[7] GameDev.net-Cyclic Redundancy Checking. http://www.gamedev.net/reference/articles/article1941.asp

[8] Honeypot-Webopedia. http://www.webopedia.com/TERM/H/honeypot.html

[9] Matloff, Norm. Cyclic Redundancy Checking. University of California, Davis. September 7, 2001. http://heather.cs.ucdavis.edu/~matloff/Networks/CRC/CRC.pdf

[10] Mhash. http://mhash.sourceforge.net/

[11] Kazaa. Kazaa Media Desktop sets most downloaded software record. http://www.kazaa.com/us/news/most_downloaded.htm. May 26, 2003.

[12] Carey Nachenberg. Computer Parasitology. Symantec AntiVirus Research Center

[13] John Leyden. Fizzer Stealth Worm Spreads Via KaZaA. Security Focus News. May 12, 2003. http://www.securityfocus.com/news/4660

[14] Rivest, Robert. The MD5 Message-Digest Algorithm. IETF RFC 1321. April 1992. http://www.ietf.org/rfc/rfc1321.txt?number=1321

[15] Seng, Li Peng. Computer Worms Costing Billions Globally. eSecurityPlanet.com. May 7, 2002. http://www.esecurityplanet.com/views/article.php/1038861

[16] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, N. Weaver. The Spread of the Sapphire/Slammer Worm. February 2003. http://www.caida.org/outreach/papers/2003/sapphire/

[17] Don Seeley. A Tour of the Worm. Dept of Computer Science, University of Utah.

[18] Eugene Spafford. The Internet Worm Program: An Analysis. Purdue Technical Report CSD-TR-823. December 8, 1988.

[19] Staniford, Stuart. Distributed Tracing of Intruders. University of California, Davis. March 1995.

[20] Stuart Staniford, Vern Paxson, Nicholas Weaver. How to 0wn the Internet in Your Spare Time. In Proceedings of the 11th USENIX Security Symposium. August 5-9 2002.

[21] Stuart Staniford-Chen, L. Todd Heberlein. Holding Intruders Accountable on the Internet. In Proceedings of the 1995 IEEE Symposium on Security and Privacy, pages 39-49, Oakland, CA, May 1995.

[22] The Symantec Enterprise Papers, Understanding and Managing Polymorphic Viruses. Volume XXX, 1996.

[23] Symantec Enterprise Security. Intrusion Detection Systems: Symantec ManTrap, Technology Brief. 2003. http://enterprisesecurity.symantec.com/content/displaypdf.cfm?pdfid=343&EID=0

[24] Weiss, Mark A. Source Code for Data Structures and Algorithm Analysis in C++ (Second Edition). Addison-Wesley, 1999. http://www.cs.fiu.edu/~weiss/dsaa_c++/code/

Appendix A: checksum.h

```
/*
 * checksum.h
 *
 * Programmed by: Bani Shahbaz, Raymond Centeno
 *
 * Contains function declarations and constants used by hashing functions
 *
*/
#ifndef CHECKSUM_H
#define CHECKSUM_H

#include <stdio.h>
#include <string.h>
#include <mhash.h>
#include <stdlib.h>


#define NUM_HASHES_USED 3

/*BUFFER SIZE AND OVERLAP CAN BE CHANGED DEPENDING ON GRANULARITY DESIRED*/
#define BUFFSIZE 20
#define OVERLAP 4

enum H_TYPE { MD5, SHA1, CRC32 };  /* THIS ENUMERATION IS USED FOR INDEXING
THE HASH_ENABLE ARRAY */


int getData(FILE *fp, char buffer[BUFFSIZE], MHASH hash1, unsigned char
hashDigest[][128], short hash_enable[NUM_HASHES_USED], int overlap);
void callHash(short hash_enable[], unsigned char hash[][128], MHASH hash_1,
char buffer[], int index);
void printHash(unsigned char hash[], hashid hashType);
void printShortHash(unsigned char hash[], hashid hashType);
void compHash(unsigned char hash[], MHASH hash_1, char buffer[], hashid
hashtype);
int checksumToInt(unsigned char hash[], hashid hashType);

#endif
```

Appendix B: checksum.cc

```
/*
 * checksum.cc
 *
 * Programmed by: Bani Shahbaz, Raymond Centeno
 *
 * Contains functions used by hashing functions
 *
*/

#include "checksum.h"
#include <stdio.h>
#include <string.h>
#include <mhash.h>
#include <stdlib.h>


/*
 * getData processes the "packet" by dividing the data in BUFFSIZE segments,
calls callHash to compute
 * the CRC32 checksum, and clears the buffer variable. Rewinds the file
pointer if overlap is greater
 * than 0.
 *
 * Inputs: file pointer to "packet", buffer, hashing object, var to store
hashes, var that says which
 *         algorithm is used, integer amount of overlap
 * Output: number of hashes computed
*/
int getData(FILE *fp, char buffer[BUFFSIZE], MHASH hash1, unsigned char
hashDigest[][128], short hash_enable[NUM_HASHES_USED], int overlap)
{
      int count=BUFFSIZE;
      int index=0;

      while(count == BUFFSIZE){
      count = fread(buffer, sizeof(char), BUFFSIZE, fp);

      // use overlaps to find packets that are offset
      fseek(fp, (-1)*overlap, SEEK_CUR);

      callHash(hash_enable, hashDigest, hash1, buffer, index);
      bzero(buffer, BUFFSIZE*sizeof(char));
      index++;
      }

      return index;
}


/*
 * getData processes the "packet" by dividing the data in BUFFSIZE segments,
calls callHash to compute
 * the CRC32 checksum, and clears the buffer variable. Rewinds the file
pointer if overlap is greater
 * than 0.
```

```
 *
 * Inputs: file pointer to "packet", buffer, hashing object, var to store
hashes, var that says which
 *         algorithm is used, integer amount of overlap
 * Output: none
*/
void callHash(short hash_enable[], unsigned char hash[][128], MHASH hash_1,
char buffer[], int index)
{
      if(hash_enable[MD5]){
      bzero(hash[index], sizeof(unsigned char)*128);
      compHash(hash[index], hash_1, buffer, MHASH_MD5);
      printHash(hash[index], MHASH_MD5);
      }

      if(hash_enable[SHA1]){
      bzero(hash[index], sizeof(unsigned char)*128);
      compHash(hash[index], hash_1, buffer, MHASH_SHA1);
      printHash(hash[index], MHASH_SHA1);
      }

      if(hash_enable[CRC32]){
      bzero(hash[index], sizeof(unsigned char)*128);
      compHash(hash[index], hash_1, buffer, MHASH_CRC32);
      //printHash(hash[index], MHASH_CRC32);
      }

      return;
}



/*
 * getData processes the "packet" by dividing the data in BUFFSIZE segments,
calls callHash to compute
 * the CRC32 checksum, and clears the buffer variable. Rewinds the file
pointer if overlap is greater
 * than 0.
 *
 * precondition: hash contains the digests from the hash performed in the
most recent compHash.
 * postcondition: hash digest will be printed to the screen according to the
number in a digest of type
 *    hashType.
 *
 * Inputs: file pointer to "packet", buffer, hashing object, var to store
hashes, var that says which
 *         algorithm is used, integer amount of overlap
 * Output: none
*/
void printHash(unsigned char hash[], hashid hashType)
{
      int i;

      switch (hashType){
      case MHASH_MD5:
            printf("\nMD5: ");
            break;
      case MHASH_SHA1:
```

```
            printf("\nSHA1: ");
            break;
      case MHASH_CRC32:
            printf("\nCRC32: ");
            break;
      default:
            break;
      }

      printf("\nHASH BLOCK SIZE: %d\n", mhash_get_block_size(hashType));

      for (i = 0; i < mhash_get_block_size(hashType); i++)
      {
            printf("%x", hash[i]);
        }
      printf("\n");


}


/*
 * getData processes the "packet" by dividing the data in BUFFSIZE segments,
calls callHash to compute
 * the CRC32 checksum, and clears the buffer variable. Rewinds the file
pointer if overlap is greater
 * than 0.
 *
 * Inputs: file pointer to "packet", buffer, hashing object, var to store
hashes, var that says which
 *         algorithm is used, integer amount of overlap
 * Output: none
*/
void printShortHash(unsigned char hash[], hashid hashType)
{
      int i;

      for (i = 0; i < mhash_get_block_size(hashType); i++)
      {
            printf("%x", hash[i]);
      }
}


/*
 * getData processes the "packet" by dividing the data in BUFFSIZE segments,
calls callHash to compute
 * the CRC32 checksum, and clears the buffer variable. Rewinds the file
pointer if overlap is greater
 * than 0.
 *
 * precondition: buffer has data to be hashed, hashType is a valid type of
hashid, hash_1 has been defined,
 *    and hash contains enough space for all the possible digests.
 * postcondition: hash will contain the digests for the hash of type hashType
on the data in buffer.
 *
 * Inputs: file pointer to "packet", buffer, hashing object, var to store
hashes, var that says which
```

```
 *         algorithm is used, integer amount of overlap
 * Output: none
*/
void compHash(unsigned char hash[], MHASH hash_1, char buffer[], hashid
hashType)
{
      hash_1 = mhash_init(hashType);
      if(hash_1 == MHASH_FAILED)
      {
            printf("HASH FAILED\n");
            exit(1);
      }

      mhash(hash_1, buffer, BUFFSIZE*sizeof(char));
      mhash_deinit(hash_1, hash);
      return;
}


/*
 * getData processes the "packet" by dividing the data in BUFFSIZE segments,
calls callHash to compute
 * the CRC32 checksum, and clears the buffer variable. Rewinds the file
pointer if overlap is greater
 * than 0.
 *
 * Inputs: file pointer to "packet", buffer, hashing object, var to store
hashes, var that says which
 *         algorithm is used, integer amount of overlap
 * Output: number of hashes computed
*/
int checksumToInt(unsigned char hash[], hashid hashType)
{
      int newValue=0;

      for (int i=0; i<mhash_get_block_size(hashType); i++)
      {
            newValue= (37*newValue) + (int)hash[i];
      }

      printf("hash:%d\n", newValue);
      return newValue;
}
```

Appendix C: compare.cc

```
/*
 * compare.cc
 *
 * Programmed by: Bani Shahbaz, Raymond Centeno
 *
 * Contains function declarations and constants used by hashing functions
 *
*/

#include <iostream>
#include <mhash.h>
#include <math.h>
#include "QuadraticProbing.h"
#include "NetPacket.h"
#include "checksum.h"
#include "CPUTimer.h"

#define PARANOIA 50           // paranoia level, integer between 0 and 100
#define HASHSIZE 50000        // size of initial hash table

using namespace std;

int main(int argc, char *argv[]){
      // VARIABLE DECLARATIONS

      // Used by hashing digest
      MHASH hash1;                        // hashing object
      char charBuffer[BUFFSIZE];          // BUFFSIZE defined in checksum.h
      unsigned char hashDigest[5000][128];      // collection of hashing
digests
      short hash_enable[NUM_HASHES_USED]={0}; // NUM_HASHES_USED defined in
checksum.h
      FILE *fp_original, *fp_modified;    // FILE pointers

      // Used by CPU timer
      CPUTimer ct;                        // variable used to compute CPU
time

      // Used by database hash
      QuadraticHashTable<int> checksumHash(-100, HASHSIZE);
      int debug=0;                        // If debug==1, display debugging
output
                                    // If debug==0, don't display output
      int paranoia, threshold;            // Parameters used by comparator to
decide % similarity
      int foundChecksum=0;                // total # of checksums found in DB
      int totalChecksum=0;                // total # of checksums in original
packet
      int totalChecksumNewPacket=0;       // total # of checksums in the new
packet
      int ratio=0;                        // foundChecksum/totalChecksum

      // Use CRC32 for the checksums
      hash_enable[CRC32]=1;
```

```
	// Zero out the buffer used for the CRC32 hashes
	// BUFFSIZE defined in checksum.h
	bzero(charBuffer, BUFFSIZE*sizeof(char));

	// Process the original packet, overlap is 0
		// open the packet
	fp_original = fopen("approx.cc","r");
	if(fp_original==NULL){
		printf("\nERROR OPENING INPUT FILE\n");
		exit(1);
	}
		// calculate the checksums
	totalChecksum=getData(fp_original, charBuffer, hash1, hashDigest,
hash_enable, 4);

		// insert into hash table
	for (int i=0; i<totalChecksum; i++){
		checksumHash.insert(checksumToInt(hashDigest[i],MHASH_CRC32));
		//printHash(hashDigest[i], MHASH_CRC32);
	}

		// close the packet
	fclose(fp_original);


	// Process the incoming packet to compare with the first packet
		// open the packet
	fp_modified = fopen("approx_modified.cc","r");
	if(fp_modified==NULL){
		printf("\nERROR OPENING INPUT FILE\n");
		exit(1);
	}

		// calculate the checksums
	totalChecksumNewPacket=getData(fp_modified, charBuffer, hash1,
hashDigest, hash_enable, 19);

		// close the packet
	fclose(fp_modified);


	// Begin the comparator
	if(debug)
		cout << ">> Beginning the comparator module" << endl;

	// Prep work for comparator
	if (argc==2 && argv[1][0]=='d')
		debug=1;

	// Set the paranoia level
	paranoia=PARANOIA;
	if (paranoia < 0) paranoia=0;
	if (paranoia > 100) paranoia = 100;
	threshold=(100-paranoia);

	// Reset the CPU timer
	ct.reset();

	if (debug){
```

```
            cout << ">> Using threshold of " << threshold << "%" << endl;
      }

      // Calculate the checksums and store
      foundChecksum=0;
      for (int i=0; i<totalChecksumNewPacket; i++){
            if (debug){
                  cout << ">> " << i << ": Calculated checksum: ";
                  printShortHash(hashDigest[i], MHASH_CRC32);
                  cout << endl;
            }

            // For each checksum, check the database for a match
            if (debug){
                  cout << ">> Checking database for ";
                  printShortHash(hashDigest[i], MHASH_CRC32);
                  cout << endl;
            }
            // Match found, inc found var
            if
(checksumHash.find(checksumToInt(hashDigest[i],MHASH_CRC32)) != -100){
                  foundChecksum++;
                  if (debug){
                        cout << ">> Checksum ";
                        printShortHash(hashDigest[i], MHASH_CRC32);
                        cout << " found\n" << endl;
                  }
            }
            // No match found
            else{
                  if (debug){
                        cout << ">> Adding Checksum ";
                        printShortHash(hashDigest[i], MHASH_CRC32);
                        cout << " to database\n" << endl;
                  }
            }
      }

      // If percentage > threshold (defined 100-paranoia_level)
      // output "Packet found"
      cout << foundChecksum << ":" << totalChecksum << endl;
      ratio = (int)(((float)foundChecksum/(float)totalChecksum)*100.0);
      if(debug){
            cout << ">> Using threshold of " << threshold << "%" << endl;
            cout << ">> Ratio calculated: " << ratio << "%" << endl;
      }

      // Develop rule
      if (ratio>threshold){
            if (debug){
                  cout << ">> Possible attack packet found. Generating rule."
<< endl;
            }
            cout << ">> RULE: Block on port " << "YYYY" << " for " << "1 min"
<< endl;
      }
      else{
            if (debug){
```

```
                  cout << ">> Threshold level not met. Packet believed to be
safe." << endl;
            }
      }

      // Display the time used by the CPU in computing the checksums
      cout << ">> CPU time: " << ct.cur_CPUTime() << endl;

      if (debug){
            cout << ">> Ending the comparator module" << endl;
      }

      return 1;
}
```

Appendix D: QuadraticProbing.h [24]

```
        #ifndef _QUADRATIC_PROBING_H_
        #define _QUADRATIC_PROBING_H_


        #include "vector.h"
        #include "mystring.h"



        // QuadraticProbing Hash table class
        //
        // CONSTRUCTION: an initialization for ITEM_NOT_FOUND
        //               and an approximate initial size or default of 101
        //
        // ******************PUBLIC OPERATIONS*********************
        // void insert( x )       --> Insert x
        // void remove( x )       --> Remove x
        // Hashable find( x )     --> Return item that matches x
        // void makeEmpty( )      --> Remove all items
        // int hash( String str, int tableSize )
        //                        --> Static method to hash strings

        template <class HashedObj>
        class QuadraticHashTable
        {
          public:
            explicit QuadraticHashTable( const HashedObj & notFound, int size
= 101 );
            QuadraticHashTable( const QuadraticHashTable & rhs )
              : ITEM_NOT_FOUND( rhs.ITEM_NOT_FOUND ),
                array( rhs.array ), currentSize( rhs.currentSize ) { }

            const HashedObj & find( const HashedObj & x ) const;

            void makeEmpty( );
            void insert( const HashedObj & x );
            void remove( const HashedObj & x );

            const QuadraticHashTable & operator=( const QuadraticHashTable &
rhs );

            enum EntryType { ACTIVE, EMPTY, DELETED };
          private:
            struct HashEntry
            {
                HashedObj element;
                EntryType info;

                HashEntry( const HashedObj & e = HashedObj( ), EntryType i =
EMPTY )
                  : element( e ), info( i ) { }
            };

            vector<HashEntry> array;
            int currentSize;
```

```
    const HashedObj ITEM_NOT_FOUND;
    bool isPrime( int n ) const;
    int nextPrime( int n ) const;
    bool isActive( int currentPos ) const;
    int findPos( const HashedObj & x ) const;
    int hash( const string & key, int tableSize ) const;
    int hash( int key, int tableSize ) const;
  int hash( unsigned char* key, int tableSize ) const;
    void rehash( );
};

#include "QuadraticProbing.cpp"
#endif
```

Appendix E: QuadraticProbing.cpp [24]

```
        #include "QuadraticProbing.h"
        #include <iostream.h>

        /**
         * Internal method to test if a positive number is prime.
         * Not an efficient algorithm.
         */
        template <class HashedObj>
        bool QuadraticHashTable<HashedObj>::isPrime( int n ) const
        {
            if( n == 2 || n == 3 )
                return true;

            if( n == 1 || n % 2 == 0 )
                return false;

            for( int i = 3; i * i <= n; i += 2 )
                if( n % i == 0 )
                    return false;

            return true;
        }

        /**
         * Internal method to return a prime number at least as large as n.
         * Assumes n > 0.
         */
        template <class HashedObj>
        int QuadraticHashTable<HashedObj>::nextPrime( int n ) const
        {
            if( n % 2 == 0 )
                n++;

            for( ; !isPrime( n ); n += 2 )
                ;

            return n;
        }

        /**
         * Construct the hash table.
         */
        template <class HashedObj>
        QuadraticHashTable<HashedObj>::QuadraticHashTable( const HashedObj &
notFound, int size )
          : ITEM_NOT_FOUND( notFound ), array( nextPrime( size ) )
        {
            makeEmpty( );
        }

        /**
         * Insert item x into the hash table. If the item is
         * already present, then do nothing.
         */
        template <class HashedObj>
```

```
        void QuadraticHashTable<HashedObj>::insert( const HashedObj & x )
        {
                // Insert x as active
            int currentPos = findPos( x );
            if( isActive( currentPos ) )
                return;
            array[ currentPos ] = HashEntry( x, ACTIVE );

                // Rehash; see Section 5.5
            if( ++currentSize > array.size( ) / 2 )
                rehash( );
        }

        /**
         * Expand the hash table.
         */
        template <class HashedObj>
        void QuadraticHashTable<HashedObj>::rehash( )
        {
            vector<HashEntry> oldArray = array;

                // Create new double-sized, empty table
            array.resize( nextPrime( 2 * oldArray.size( ) ) );
            for( int j = 0; j < array.size( ); j++ )
                array[ j ].info = EMPTY;

                // Copy table over
            currentSize = 0;
            for( int i = 0; i < oldArray.size( ); i++ )
                if( oldArray[ i ].info == ACTIVE )
                    insert( oldArray[ i ].element );
        }

        /**
         * Method that performs quadratic probing resolution.
         * Return the position where the search for x terminates.
         */
        template <class HashedObj>
        int QuadraticHashTable<HashedObj>::findPos( const HashedObj & x )
const
        {
/* 1*/      int collisionNum = 0;
/* 2*/      int currentPos = hash( x, array.size( ) );

/* 3*/      while( array[ currentPos ].info != EMPTY &&
                   array[ currentPos ].element != x )
            {
/* 4*/          currentPos += 2 * ++collisionNum - 1;  // Compute ith probe
/* 5*/          if( currentPos >= array.size( ) )
/* 6*/              currentPos -= array.size( );
            }

/* 7*/      return currentPos;
        }


        /**
         * Remove item x from the hash table.
         */
```

```
        template <class HashedObj>
        void QuadraticHashTable<HashedObj>::remove( const HashedObj & x )
        {
            int currentPos = findPos( x );
            if( isActive( currentPos ) )
                array[ currentPos ].info = DELETED;
        }

        /**
         * Find item x in the hash table.
         * Return the matching item, or ITEM_NOT_FOUND, if not found.
         */
        template <class HashedObj>
        const HashedObj & QuadraticHashTable<HashedObj>::find( const
HashedObj & x ) const
        {
            int currentPos = findPos( x );
            return isActive( currentPos ) ? array[ currentPos ].element :
ITEM_NOT_FOUND;
        }

        /**
         * Make the hash table logically empty.
         */
        template <class HashedObj>
        void QuadraticHashTable<HashedObj>::makeEmpty( )
        {
            currentSize = 0;
            for( int i = 0; i < array.size( ); i++ )
                array[ i ].info = EMPTY;
        }

        /**
         * Deep copy.
         */
        template <class HashedObj>
        const QuadraticHashTable<HashedObj> & QuadraticHashTable<HashedObj>::
        operator=( const QuadraticHashTable<HashedObj> & rhs )
        {
            if( this != &rhs )
            {
                array = rhs.array;
                currentSize = rhs.currentSize;
            }
            return *this;
        }


        /**
         * Return true if currentPos exists and is active.
         */
        template <class HashedObj>
        bool QuadraticHashTable<HashedObj>::isActive( int currentPos ) const
        {
            return array[ currentPos ].info == ACTIVE;
        }

        /**
         * A hash routine for string objects.
```

```
         */
        template <class HashedObj>
        int QuadraticHashTable<HashedObj>::hash( const string & key, int
tableSize ) const
        {
            int hashVal = 0;

            for( int i = 0; i < key.length( ); i++ )
                hashVal = 37 * hashVal + key[ i ];

            hashVal %= tableSize;
            if( hashVal < 0 )
               hashVal += tableSize;
            return hashVal;
        }


        /**
         * A hash routine for ints.
         */
         template <class HashedObj>
        int QuadraticHashTable<HashedObj>::hash( int key, int tableSize )
const
        {
            if( key < 0 ) key = -key;
            return key % tableSize;
        }

      /**
       * A hash routine for unsigned char*
       */
       template <class HashedObj>
      int QuadraticHashTable<HashedObj>::hash( unsigned char* key, int
tableSize ) const
      {
          int hashVal = 0;

          for( int i = 0; i < 128; i++ )
            hashVal = 37 * hashVal + key[i];

          hashVal %= tableSize;
          if( hashVal < 0 )
            hashVal += tableSize;

          return hashVal;
      }
```

Appendix F: CPUTimer.h

```
#ifndef CPUTIMER_H
#define CPUTIMER_H

#include <time.h>
#include <iostream.h>

// CPUTimer
//
// For ECS 110 - UC Davis - By Ted Krovetz
//
// This class is a convenient way to count CPU time.
// The ANSI C function clock() is used to get the current
// user + system time that the current application
// (and it's child processes) has expended.
// At creation, a variable of CPUTimer type records
// the current value of clock().
//
// reset() - sets the stored variable to the current
//           value of clock(). In essence, resetting the clock.
//
// cur_CPUTime() - returns the difference between the current
//                 clock() value and the previously stored value.
//                 In essence, returning how much CPU time has passed.
//
// Example: to time a function (possibly main())
//
//   #include "CPUTimer"
//   void foo(void) {
//     CPUTimer ct;
//     ... whatever foo does ...
//     cerr << ct.cur_CPUTime() << endl;
//   }

class CPUTimer {
private:
  clock_t tick_count;

public:
  CPUTimer(void);
  void reset(void);
  double cur_CPUTime(void);
};

// AutoCPUTimer
//
// AutoCPUTimer is derived through C++ inheritance. It
// inherits all the public members of CPUTimer, but
// includes a destructor which will automatically
// output the CPU time used to cerr (stderr).
// Example: to time a function (possibly main())
//
//   #include "CPUTimer"
//   void foo(void) {
//     AutoCPUTimer at;
//     ... whatever foo does ...
```

```
//   }
//
// This example will have identical output to the
// previous example, however the output to cerr is
// done automatically,.

class AutoCPUTimer : public CPUTimer {
public:
  ~AutoCPUTimer(void);
};

// Implementation --
// It is generally not good to expose the mechanics of your ADT
// In the public interface (i.e. the header file). It is here
// however, to make program timing as simple as possible.
// There is _NO_ .cpp file for these classes. #include'ing
// is sufficient for their use.

CPUTimer::CPUTimer(void)
{
  tick_count = clock();
}

void CPUTimer::reset(void)
{
  tick_count = clock();
}

double CPUTimer::cur_CPUTime(void)
{
  return double(clock() - tick_count) / CLOCKS_PER_SEC;
}

AutoCPUTimer::~AutoCPUTimer(void)
{
  cerr << cur_CPUTime() << endl;
}

#endif
```

Appendix G: chks.c

```
/*    USAGE OF THIS FILE
 *    REQUIRES mhash.h
 *
 *    executable is called checksum
 *
 *    checksum inputfile [m] [c] [s] [o integer] [b integer] [d] [p]
 *
 *    m - enable md5
 *    c - enable crc32
 *    s - enable sha1
 *    o - set over lap to integer
 *    b - set buffsize to integer
 *    d - debug option
 *    p - enable polymorphic digests
 */



#include <stdio.h>
#include <string.h>
#include <mhash.h>
#include <stdlib.h>


#define NUM_HASHES_USED 3

/*BUFFER SIZE AND OVERLAP CAN BE CHANGED DEPENDING ON GRANULARITY DESIRED*/
//#define BUFFSIZE 20
//#define OVERLAP 6

int BUFFSIZE = 20;
int OVERLAP = 6;

void checkArgs(int argc, char *argv[], short hash_enable[NUM_HASHES_USED],
FILE **fp);
void compHash(unsigned char hash[], MHASH hash_1, char buffer[], hashid
hashtype);
void printHash(unsigned char hash[], hashid hashType);
void callHash(short hash_enable[], unsigned char hash[], MHASH hash_1, char
buffer[]);
int getData(FILE *fp, char buffer[BUFFSIZE], MHASH hash1, unsigned char
hashDigest[128], short hash_enable[NUM_HASHES_USED]);

enum H_TYPE { MD5, SHA1, CRC32 };  /* THIS ENUMERATION IS USED FOR INDEXING
THE HASH_ENABLE ARRAY */


int DEBUG = 0; /* DEBUG VARIABLE */
int POLY_TEST =0; /* CREATE DIGESTS FOR SHIFTED POLYMORPHISM */

int main(int argc, char *argv[])
{
      MHASH hash1;
      char x[BUFFSIZE];
      unsigned char hashDigest[128];
      short hash_enable[NUM_HASHES_USED]={0};
```

```
	FILE *fp;

	checkArgs(argc, argv, hash_enable, &fp);

	bzero(x, BUFFSIZE*sizeof(char));

	getData(fp, x,hash1, hashDigest,hash_enable);

	printf("\n");

	return 0;
}

/*	get data gets BUFFSIZE data from file pointer fp, calls callHash to
determine what hashes it will perform
 */
int getData(FILE *fp, char buffer[BUFFSIZE], MHASH hash1, unsigned char
hashDigest[128], short hash_enable[NUM_HASHES_USED])
{
	int count=BUFFSIZE;
	int i=0;
	int flag=1;


	/*The outer for loop along with the first fread and the rewind are used
in the case where
	we want to check for data insertion polymorphism.  More digests are
made, one for
	each possible shift, there are BUFFSIZE possible shifts*/

	for(i=0; i<BUFFSIZE && flag == 1; i++)
	{

	if(POLY_TEST != 1) flag = 0;
	if(POLY_TEST == 1) fread(buffer,sizeof(char), i, fp);

	while(count == BUFFSIZE){
	count = fread(buffer, sizeof(char), BUFFSIZE, fp);

	fseek(fp, (-1)*OVERLAP, SEEK_CUR);
	callHash(hash_enable, hashDigest, hash1, buffer);
	bzero(buffer, BUFFSIZE*sizeof(char));
	}

	printf("********\n");
	count=BUFFSIZE;
	if(POLY_TEST == 1) rewind(fp);
	}

	return count;
}



/*NOT VERY ROBUST BUT GETS THE JOB DONE*/
void checkArgs(int argc, char *argv[], short hash_enable[NUM_HASHES_USED],
FILE **fp)
{
	int i;
```

```
	*fp = fopen(argv[1], "r");
	if(fp==NULL){
		printf("\nERROR OPENING INPUT FILE\n");
		exit(1);
	}
	if(argc > 2){
		for(i=2; i<argc; i++)
		{
		if(strcmp(argv[i], "m")==0) hash_enable[MD5]=1;
	//enable md5
		if(strcmp(argv[i], "s")==0) hash_enable[SHA1]=1;
	//enable SHA1
		if(strcmp(argv[i], "c")==0) hash_enable[CRC32]=1;
	//enable crc32
		if(strcmp(argv[i], "d")==0) DEBUG = 1;
	//enable debug
		if(strcmp(argv[i], "o")==0) OVERLAP = atoi(argv[i+1]);
	//change overlap size
		if(strcmp(argv[i], "b")==0) BUFFSIZE = atoi(argv[i+1]);
	//change buffsize
		if(strcmp(argv[i], "p")==0) POLY_TEST = 1;
	//enable polymorphic digest
		}
	}
}


/*	determines which hashes to compute according to the hash_enable array
which was
 *	filled out during the argument check
 */
void callHash(short hash_enable[], unsigned char hash[], MHASH hash_1, char
buffer[])
{
	if(hash_enable[MD5]){
	bzero(hash, sizeof(unsigned char)*128);
	compHash(hash, hash_1, buffer, MHASH_MD5);
	printHash(hash, MHASH_MD5);
	}

	if(hash_enable[SHA1]){
	bzero(hash, sizeof(unsigned char)*128);
	compHash(hash, hash_1, buffer, MHASH_SHA1);
	printHash(hash, MHASH_SHA1);
	}

	if(hash_enable[CRC32]){
	bzero(hash, sizeof(unsigned char)*128);
	compHash(hash, hash_1, buffer, MHASH_CRC32);
	printHash(hash, MHASH_CRC32);
	}

	return;
}
```

```
/*
      precondition: hash contains the digests from the hash performed in the
most recent compHash.
      postcondition: hash digest will be printed to the screen according to
the number in a digest of type hashType.
*/
void printHash(unsigned char hash[], hashid hashType)
{
      int i;

      if(DEBUG){
      switch (hashType){
      case MHASH_MD5:
            printf("\nMD5: ");
            break;
      case MHASH_SHA1:
            printf("\nSHA1: ");
            break;
      case MHASH_CRC32:
            printf("\nCRC32: ");
            break;
      default:
            break;
      }
      }

      if(DEBUG) {printf("\nHASH BLOCK SIZE: %d\n",
mhash_get_block_size(hashType));}


      for (i = 0; i < mhash_get_block_size(hashType); i++)
      {
            printf("%.2x", hash[i]);
        }
      printf("\n");


}

/*
      precondition: buffer has data to be hashed, hashType is a valid type of
hashid, hash_1 has been defined,
            and hash contains enough space for all the possible digests.
      postcondition: hash will contain the digests for the hash of type
hashType on the data in buffer.
*/
void compHash(unsigned char hash[], MHASH hash_1, char buffer[], hashid
hashType)
{
      hash_1 = mhash_init(hashType);
      if(hash_1 == MHASH_FAILED)
      {
            printf("HASH FAILED\n");
            exit(1);
      }
      mhash(hash_1, buffer, BUFFSIZE*sizeof(char));
      mhash_deinit(hash_1, hash);
```

```
        return;
    }
```

Appendix H: bufftest.pl

```
#!/usr/bin/perl -w

# this script is used to compare two data files using a number of different
block sizes
# and overlaps in order to determine the optimal combination of the two.
# the two files are named infile and infile2.
# first their digests are calculated and placed in output1 and output2
# then their digests are compared to determine the % difference between the
two
# The results of the computation are printed to the file called "results"
# table1.xls was created using data gotten from this script


system("rm -f results");
open(RESULTS, ">>results") || die "could not open results: $!";


for($i=5; $i < 55; $i = $i+5)
{
      for($j=0; $j<$i/2; $j=$j+1)
      {

            $count = 0;
            $numlines =0;
            $percent =0;
            system("./checksum infile c o $j b $i > output1");
            system("./checksum infile2 c o $j b $i > output2");

            open(IN1, "output1") || die "could not open output1: $!";
            open(IN2, "output2") || die "could not open output2: $!";

            while(defined ($check = <IN1>))
            {
                  $check2 = <IN2>;
                  chomp ($check);
                  chomp ($check2);

                  if($check ne $check2)
                  {
                        $count = $count+1;
                  }
                  $numlines = $numlines +1;
            }

            if($numlines ne 0){$percent = $count/$numlines;}

            print RESULTS "$i $j $count $numlines $percent\n";

            close(IN1);
            close(IN2);
      }
}
```

Appendix I: table1.xls

SAMPLE 1				
Block Size	Overlap	# of diff. Bits	Total # blocks	Difference Ratio
5	0	23	278	0.0827338
5	1	27	347	0.0778098
5	2	37	461	0.0802603
10	0	15	140	0.1071429
10	1	16	156	0.1025641
10	2	18	175	0.1028571
10	3	21	199	0.1055276
10	4	22	232	0.0948276
15	0	11	94	0.1170213
15	1	11	101	0.1089109
15	2	13	108	0.1203704
15	3	12	117	0.1025641
15	4	15	127	0.1181102
15	5	17	140	0.1214286
15	6	18	155	0.1161290
15	7	20	174	0.1149425
20	0	10	71	0.1408451
20	1	10	75	0.1333333
20	2	10	79	0.1265823
20	3	9	83	0.1084337
20	4	12	88	0.1363636
20	5	14	94	0.1489362
20	6	13	101	0.1287129
20	7	13	108	0.1203704
20	8	15	117	0.1282051
20	9	17	127	0.1338583
25	0	8	58	0.1379310
25	1	8	60	0.1333333
25	2	9	62	0.1451613
25	3	9	65	0.1384615
25	4	9	68	0.1323529
25	5	10	71	0.1408451
25	6	10	75	0.1333333
25	7	10	79	0.1265823
25	8	11	83	0.1325301
25	9	12	88	0.1363636
25	10	14	94	0.1489362
25	11	14	100	0.1400000
25	12	17	108	0.1574074
30	0	7	48	0.1458333
30	1	8	50	0.1600000
30	2	9	52	0.1730769
30	3	8	53	0.1509434
30	4	8	55	0.1454545
30	5	8	57	0.1403509
30	6	8	60	0.1333333
30	7	9	62	0.1451613
30	8	11	65	0.1692308
30	9	11	68	0.1617647
30	**10**	**11**	**71**	**0.1549296**
30	11	11	74	0.1486486
30	12	11	78	0.1410256
30	13	12	83	0.1445783
30	14	13	88	0.1477273
35	0	8	42	0.1904762
35	1	6	43	0.1395349
35	2	7	44	0.1590909
35	3	7	45	0.1555556
35	4	9	47	0.1914894
35	5	9	48	0.1875000
35	6	9	50	0.1800000
35	7	9	51	0.1764706
35	8	9	53	0.1698113
35	9	9	55	0.1636364
35	10	10	57	0.1754386
35	11	9	59	0.1525424
35	12	10	62	0.1612903
35	13	12	65	0.1846154
35	14	12	67	0.1791045
35	15	13	71	0.1830986
35	16	12	74	0.1621622
35	17	13	78	0.1666667
40	0	6	37	0.1621622
40	1	7	38	0.1842105
40	2	7	39	0.1794872
40	3	6	40	0.1500000
40	4	7	41	0.1707317
40	5	8	42	0.1904762
40	6	7	43	0.1627907
40	7	7	44	0.1590909
40	8	8	45	0.1777778
40	9	9	47	0.1914894
40	10	9	48	0.1875000
40	11	9	50	0.1800000
40	12	10	51	0.1960784
40	13	9	53	0.1698113
40	14	11	55	0.2000000
40	15	11	57	0.1929825
40	16	9	59	0.1525424
40	17	12	62	0.1935484
40	18	12	64	0.1875000
40	19	13	67	0.1940299

SAMPLE 2				
Block Size	Overlap	# of diff. Bits	Total # blocks	Difference Ratio
5	0	5	127	0.0393701
5	1	7	158	0.0443038
5	2	8	209	0.0382775
10	0	4	65	0.0615385
10	1	6	71	0.0845070
10	2	5	80	0.0625000
10	3	7	91	0.0769231
10	4	9	105	0.0857143
15	0	4	44	0.0909091
15	1	5	47	0.1063830
15	2	4	50	0.0800000
15	3	5	54	0.0925926
15	4	7	59	0.1186441
15	5	6	64	0.0937500
15	6	8	71	0.1126761
15	7	9	79	0.1139241
20	0	4	34	0.1176471
20	1	5	35	0.1428571
20	2	4	37	0.1081081
20	3	5	39	0.1282051
20	4	5	41	0.1219512
20	5	6	44	0.1363636
20	6	8	46	0.1739130
20	7	5	50	0.1000000
20	8	7	54	0.1296296
20	9	7	58	0.1206897
25	0	4	27	0.1481481
25	1	4	28	0.1428571
25	2	5	29	0.1724138
25	3	4	31	0.1290323
25	4	5	32	0.1562500
25	5	5	33	0.1515152
25	6	6	35	0.1714286
25	7	6	37	0.1621622
25	8	6	39	0.1538462
25	9	5	41	0.1219512
25	10	6	43	0.1395349
25	11	8	46	0.1739130
25	12	8	49	0.1632653
30	0	4	23	0.1739130
30	1	4	24	0.1666667
30	2	5	25	0.2000000
30	3	4	25	0.1600000
30	4	4	26	0.1538462
30	5	8	27	0.2962963
30	6	6	28	0.2142857
30	7	7	29	0.2413793
30	8	4	30	0.1333333
30	9	5	32	0.1562500
30	**10**	**5**	**33**	**0.1515152**
30	11	7	35	0.2000000
30	12	8	36	0.2222222
30	13	7	38	0.1842105
30	14	7	40	0.1750000
35	0	4	20	0.2000000
35	1	5	21	0.2380952
35	2	4	21	0.1904762
35	3	5	22	0.2272727
35	4	4	22	0.1818182
35	5	5	23	0.2173913
35	6	4	24	0.1666667
35	7	6	24	0.2500000
35	8	4	25	0.1600000
35	9	6	26	0.2307692
35	10	8	27	0.2962963
35	11	6	28	0.2142857
35	12	7	29	0.2413793
35	13	7	30	0.2333333
35	14	6	31	0.1935484
35	15	6	33	0.1818182
35	16	7	34	0.2058824
35	17	8	36	0.2222222
40	0	4	18	0.2222222
40	1	4	18	0.2222222
40	2	5	19	0.2631579
40	3	5	19	0.2631579
40	4	6	20	0.3000000
40	5	4	20	0.2000000
40	6	5	21	0.2380952
40	7	4	21	0.1904762
40	8	5	22	0.2272727
40	9	5	22	0.2272727
40	10	5	23	0.2173913
40	11	4	24	0.1666667
40	12	6	24	0.2500000
40	13	6	25	0.2400000
40	14	6	26	0.2307692
40	15	8	27	0.2962963
40	16	7	28	0.2500000
40	17	8	29	0.2758621
40	18	8	30	0.2666667
40	19	8	31	0.2580645

SAMPLE 3				
Block Size	Overlap	# of diff. Bits	Total # blocks	Difference Ratio
5	0	41	13277	0.0030880
5	1	57	16595	0.0034348
5	2	74	22126	0.0033445
10	0	35	6640	0.0052711
10	1	39	7377	0.0052867
10	2	46	8299	0.0055428
10	3	51	9484	0.0053775
10	4	63	11064	0.0056941
15	0	33	4427	0.0074543
15	1	33	4743	0.0069576
15	2	39	5108	0.0076351
15	3	41	5533	0.0074101
15	4	41	6036	0.0067926
15	5	47	6639	0.0070794
15	6	52	7376	0.0070499
15	7	61	8298	0.0073512
20	0	29	3321	0.0087323
20	1	34	3496	0.0097254
20	2	37	3690	0.0100271
20	3	33	3907	0.0084464
20	4	37	4150	0.0089157
20	5	42	4427	0.0094872
20	6	43	4743	0.0090660
20	7	51	5107	0.0099863
20	8	53	5533	0.0095789
20	9	56	6035	0.0092792
25	0	28	2657	0.0105382
25	1	33	2768	0.0119220
25	2	34	2888	0.0117729
25	3	29	3019	0.0096058
25	4	37	3163	0.0116978
25	5	36	3321	0.0108401
25	6	42	3495	0.0120172
25	7	43	3689	0.0116563
25	8	40	3906	0.0102407
25	9	47	4150	0.0113253
25	10	47	4427	0.0106167
25	11	54	4743	0.0113852
25	12	54	5107	0.0105737
30	0	31	2215	0.0139955
30	1	29	2291	0.0126582
30	2	30	2373	0.0126422
30	3	32	2461	0.0130028
30	4	34	2555	0.0133072
30	5	34	2657	0.0127964
30	6	38	2768	0.0137283
30	7	37	2888	0.0128116
30	8	37	3019	0.0122557
30	9	42	3163	0.0132785
30	**10**	**44**	**3321**	**0.0132490**
30	11	47	3495	0.0134478
30	12	50	3689	0.0135538
30	13	49	3906	0.0125448
30	14	54	4150	0.0130120
35	0	30	1899	0.0157978
35	1	29	1955	0.0148338
35	2	26	2014	0.0129096
35	3	29	2077	0.0139624
35	4	36	2143	0.0167989
35	5	35	2215	0.0158014
35	6	34	2291	0.0148407
35	7	34	2373	0.0143279
35	8	37	2460	0.0150407
35	9	42	2555	0.0164384
35	10	38	2657	0.0143018
35	11	41	2768	0.0148121
35	12	41	2888	0.0141967
35	13	42	3019	0.0139119
35	14	49	3162	0.0154965
35	15	48	3320	0.0144578
35	16	55	3495	0.0157368
35	17	56	3689	0.0151803
40	0	26	1662	0.0156438
40	1	30	1704	0.0176056
40	2	29	1749	0.0165809
40	3	35	1796	0.0194878
40	4	32	1846	0.0173348
40	5	34	1899	0.0179042
40	6	32	1954	0.0163767
40	7	31	2014	0.0153923
40	8	33	2076	0.0158960
40	9	36	2143	0.0167989
40	10	37	2215	0.0167043
40	11	40	2291	0.0174596
40	12	38	2372	0.0160202
40	13	44	2460	0.0178862
40	14	44	2555	0.0172211
40	15	46	2657	0.0173128
40	16	48	2767	0.0173473
40	17	48	2887	0.0166263
40	18	48	3019	0.0158993
40	19	53	3162	0.0167615