# NOTE TO USERS

# Runtime Detection

# of

# Active Scanning Worms

by

Andrew Preston

A thesis submitted to

The Faculty of Graduate Studies and Research

in partial fulfilment

of the requirements for the degree of

Master of Applied Science in Electrical Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

January 13, 2008

*Your file Votre référence*
*ISBN: 978-0-494-36831-2*
*Our file Notre référence*
*ISBN: 978-0-494-36831-2*

# Abstract

Intrusion Detection Systems for active scanning network worms is an active research area. A wide range of approaches from packet content monitoring to statistical measurements have been proposed to address the problems of fast spreading network worms and the economic havoc that they wreak. This thesis reviews the existing state of the art in active worm detection and proposes a new method based on locality. By tracking the mean number of destinations each source in a monitored network contacts and monitoring for abrupt upward changes, worms are detected early in their propagation stage. This low-overhead method requires only minimal inspection of the packet header. A novel means of storing the required statistics for each source using a variant of the Bloom Filter is presented. The algorithm is evaluated through simulation against the Code Red and SQL Slammer worms.

# Acknowledgements

I wish to thank my supervisor, Changchang Huang, for his encouragement and guidance throughout my time at Carleton University. I must also recognize the support and enjoyable distractions of my friends and classmates, including Megan Holtzman, Laurence Smith, Rosalyn Seeton, Matt MacLeod and Mahdi Javer. Whenever I wavered from the thesis path, my family, consisting of parents Jon and Caroline and sister Gwen, and especially my girlfriend, Kristina Schuring, never let me forget it. As I enter the new world of "after the thesis", I'm not entirely sure what we're all going to talk about now!

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This thesis explores a method for runtime detection of active scanning worms during their propagation phase. Active scanning worms are the class of computer virus that infects computers by sending probe packets to random addresses. The probes attempt to exploit the worm's targeted vulnerability and infect the destination computer, spreading the worm and increasing the network's load. The intention is that the proposed method could be used as the input to automated or manual worm response systems that quarantine infected systems, thereby mitigating the network impact and resulting economic damage of an active scanning worm event.

Most networks, and the Internet in particular, have a wide range of host types connected to them spanning PCs to printers and everything in between. The hosts vary by hardware configuration, operating system and running applications. The software stack, from drivers through the operating system and to the application level, is generally a combination of software from many sources and vendors. It only takes a lax security effort by one contributor to the overall system to introduce a vulnerability, like a buffer overflow, that a worm can exploit. If that particular component is widely

distributed across the Internet, then it becomes a target for worm developers. Not surprisingly, the most targeted platforms for worms are PCs running Microsoft Windows, particularly older versions like Windows 95 or XP. Worms have exploited both fundamental services provided by the operating system and applications, including server applications, running on top of the OS.

The most successful and virulent worms infect all susceptible hosts connected to the Internet within hours. Each infected host generates a substantial volume of outgoing probe traffic; the combined traffic volume of all infected hosts consumes all available link capacity, causing routers to fail and Internet communications to grind to a halt. Preventing a worm's spread is often as simple as closing a port on a gateway router or firewall. However, these routers are themselves controlled over the network and may be unreachable by human administrators during a worm event. Further, those administrators need information about the worm in order to defend their network from it yet their attempts to communicate with each other over the Internet are also stymied. Put simply, it is impossible for a human to respond 'quickly' enough to stop a worm's propagation once that worm is already active on their network.

The economic impact of a worm event is substantial. The author worked at a company where twice during the three years of employment the entire company, 1000 employees, was unable to work for four to five hours due to a worm causing havoc on the corporate intranet. These events were both global worm events; a simple extrapolation from the author's experience shows that a single worm can idle hundreds of thousands of workers and slow the world economy. Therefore, there is a need for effective means to prevent the spread of network worms. Many software developers, with Microsoft taking a particularly active role, have made security the focus of their

development processes in an attempt to reduce the number of vulnerabilities in their systems and stop the worm before it starts. Active research is also underway into methods for detecting the worm as it propagates across the network though monitoring of the network and the traffic carried over it. One area of this research focuses on signatures, examining packet payloads for those sent by known worms. The second area focuses on stopping unknown worms by developing statistical indicators for network activity and issuing alerts when these indicators move outside their healthy range.

Methods that identify worms through statistical analysis of network activity are capable of detecting the zero-day vulnerability, new worms that have not been seen before. There is value in systems that detect the known worms, as old worms tend to linger on the Internet for years infecting systems that have not been patched to close the worm's targeted vulnerability. However, only new worms are capable of wreaking the type of chaos that causes substantial economic impact because a new worm attacks a vulnerability that no system is patched for.

The worm detection scheme discussed in this thesis is called Source Flow Counting (SFC). It is intended to detect both known and unknown worms through anomaly detection. The algorithm maintains a record of the number of destinations contacted by each host in the network under observation. When this value has an abrupt upward change of significant magnitude, the system is deemed to be infected. All active scanning worms cause such an abrupt increase in the destination count as a worm that spreads slowly will never cause the type of damage its developer seeks. SFC uses variants of the Bloom Filter to implement its data store. This compressed data structure allows SFC to monitor quite a large network using modest storage and processing resources. Through simulation testing, SFC was able to identify hosts

infected with two representative worms: the Code Red II worm and SQL Slammer.

## 1.1 Objective

Develop a method to identify hosts infected with active scanning worms through passive monitoring for n substantial increase in short period of time in the number of destinations the host contacts. This will be done through;

1. Selecting a method for detecting the abrupt increase in destinations. Identify the required statistics and specify how they are measured.

2. Constructing a data store capable of holding the required statistics for all sources on the monitored network. The data store shall have fast read and write operations.

3. Developing a simulated network environment to exercise the worm detection algorithm. The simulation will generate realistic background traffic so as to replicate a real network as closely as is feasible.

4. Testing the worm detection algorithm against the Code Red II and SQL Slammer worms.

## 1.2 Statement of Contributions

Applied the Shewhart Control Chart to source locality to detect active scanning worms.

Developed the Arbitrary Spectral Bloom Filter (ASBF), a modified Spectral Bloom Filter (SBF), to store arbitrary values assigned to a key value.

Used a SBF to maintain both a record of flows and the number of flows originating from each source during a time period.

Used three modified SBFs to store the variables required to apply the Shewhart Control Chart to each source.

Added sliding-window functionality to the SBF by using two counter vectors within a single SBF, allowing expired data to be dropped from the filter.

Developed a small scale simulation with Hypertext Transport Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), Post Office Protocol (POP) File Transfer Protocol (FTP) and Secure Shell (SSH) background traffic to exercise the worm detection algorithm.

# Chapter 2

# Background Information

This chapter provides background information on two disparate topics that are brought together through the work. The first section covers the history, impact and development of network worms since the first worm was released on a fledgling Internet. The chapter closes with a discussion of the Bloom Filter, a data structure for storing set membership information.

## 2.1 Network Worms

### 2.1.1 History and Impacts

In the context of networks, worms are self-propagating programs that spread by exploiting security holes in widely distributed software. While early worms were used for good purposes such as network or system maintenance, modern worms are most often malicious programs that infect machines without the owner's consent and either damage data or serve as launching platforms for further attacks, such as Distributed Denial of Service (DDoS) attacks. Worms are often grouped together

with computer viruses. The key difference is that viruses generally require some type of user interaction to propagate. For example, many email viruses generally require the user to open an attachment, which appears to be a picture or some similarly innocuous file type but is actually an application or script file.

The label "worm" is derived from a 1975 John Brunner science-fiction story. In "The Shockwave Rider", the hero defeats a totalitarian government by infecting the network it uses to control the lives of its citizens with a "tapeworm" program. This was later shortened to "worm" to describe self-propagating programs. The earliest worms were written by researchers at Xerox's Palo Alto Research Center (PARC) to do useful tasks, such as posting announcements and scheduling jobs to run overnight on unattended machines. PARC also suffered the first worm infection when a coding error in one of these worms caused all the machines on the network to repeatedly crash until the first "vaccine" code could be written [1]. The first worm with malignant effect was the "Morris" worm, released by Robert T. Morris on 2 November 1988, and capable of infecting SUN and Digital Equipment Corporation (DEC) Unix machines. It was intended as a benign proof of concept but a coding error meant that instead the worm overloaded the infected computers and ground what was then the Internet to a halt[2].

Eleven years passed between the Morris worm and the next major worm event. Released on 26 March 1999, the "Melissa" worm was a Microsoft Word macro that took advantage of the almost non-existent security model in Microsoft's Mail API to self-propagate to the first 50 names in the user's Microsoft Outlook address book. This worm clogged internet mail servers with traffic and lead to a 20 month jail sentence for its author. Since Melissa, many worms have been released into the wild with various payloads. Of these, three are considered significant: Code Red, Nimda,

and SQL Slammer. All are classed as active scanning worms, which spread by probing random IP addresses looking for vulnerable systems. The specifics of how an active scanning worm's probing mechanism works depends on the weakness being exploited. Probe packets are sent over Transmission Control Protocol (TCP) or Unigram Data Protocol (UDP) to whichever port the vulnerable service is listening on.

There are three variants of the Code Red (sometimes called CodeRed or Code-Red) worm, which were heavily studied by [2]. The first version, Code Red I v1, was released on 12 July 2001 and targets a buffer overflow vulnerability in Microsoft's Internet Information Server (IIS) web server. The worm propagates during the first 19 days of each month. On each newly infected machine the worm generates a list of random IP addresses and attempts to infect them at a rate of about 11 probes per second. During the next nine days of the month the infected machines launch a DDoS attack against `www.whitehouse.gov`. The worm is dormant during the remainder of the month. Code Red I v1's impact was mitigated by a coding error. All instances of the worm use the same random number seed and as a result generate the same list of target IP address. This significantly limited both the number of infected hosts and the worm's consumption of network resource's. Code Red I v1 did very little damage to infected systems other then defacing some web pages.

At around 10 AM UTC on 19 July 2001 Code Red I v2 was released with a corrected random number generator, changing Code Red I's status from nuisance to major calamity. Figure 2.1 shows the infected host count for 32 hours beginning at 00:00 UTC on 19 July. Within 24 hours of the v2 release, there were over 359 000 infected hosts, representing a substantial portion of all IIS servers online. The figure clearly shows the characteristic growth curve of active scanning worms: a slow start is followed by a rapid increase where the majority of victims are infected. The

growth rate slows as the ratio of vulnerable hosts to infected hosts decreases. The decrease comes about because most of the vulnerable machines have been infected and the many of the remainder have been inoculated through patches or other forms of administrator intervention.

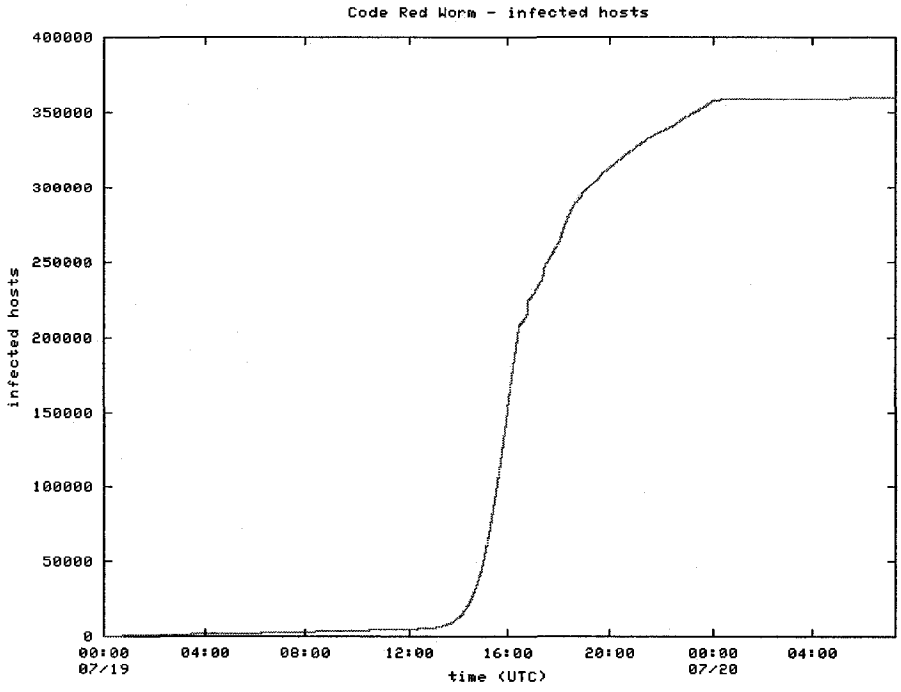


Figure 2.1: Code Red I v2 Infected Hosts: the number of hosts infected with Code Red I v2 from the worm's launch until saturation 32 hours later. [2]

The third version of Code Red, termed Code Red II, is classed with Code Red I because it exploits the same buffer-overflow exploit in IIS. The compiled worm also contained the string "CodeRedII". It was released on 4 August 2001. However, there are no further similarities; Code Red II is an entirely different worm. Rather than leave an obvious calling card through web page defacement, the worm installs a "backdoor" allowing remote control of the system. Code Red II also uses a refined

propagation scheme that targeted local machines more frequently in order to increase the infection rate. Half of the addresses tried are in the same /8 group, $\frac{3}{8}$ths of the time the address is in the same /16 and the remainder are truly random. The rational behind this scheme is that machines with similar IP addresses are more likely to be on the same subnet, resulting in a faster probe and infection. This worm also delays propagation for 24 hours after infection. This act of subterfuge means that reviewing the activity logs of an infected computer around the time propagation begins will not reveal the source of the infection.

The Nimda worm uses multiple vectors of infection to successfully penetrate firewalls and ensure that it remained active for months after its release on 18 September 2001. One vector is active scanning for an IIS vulnerability. The worm also emails itself to addresses in the infected machines' address books, copies itself to open network shares, adds exploit code to web pages served by the infected machine and uses the backdoors installed by Code Red II. This combination of methods allowed the worm to spread very rapidly, with the email vector proving particularly effective in the initial stages. Most firewalls let all mail traffic through and rely on the mail server to detect and quarantine the dangerous traffic. However, the natural delay between the release of a new worm and the subsequent update to the mail server's database with the worm's signature is a period of free travel for the worm.

SQL Slammer, released on 25 January 2003, is a UDP worm targeting a buffer-overflow exploit in Microsoft's SQL Server package. The worm itself had no malicious code; the main impact on networks came from the extreme load of the propagation mechanism. The entire worm is contained in a single 404 byte UDP packet. The Code Red and Nimda worms use TCP connections to propagate, which limits their propagation rate. Each probe requires a proper TCP connection establishment and

tear down and the socket cannot be reused until after its 2MSL timer has expired, typically 60 or 120 seconds. In an attempt to propagate rapidly, Code Red II created up to 600 threads per infected machine yet still averaged only eleven probes per second.[2] In contrast, Slammer is limited only by the available bandwidth because UDP connections are stateless. During the peak propagation period, infected hosts sent out an average of 4000 packets per second, consuming all available bandwidth. This high probe rate meant that Slammer completely infected all vulnerable machines within ten minutes of release.[3]

The combined worldwide economic impact of the three Code Red worms is estimated to be $2.62 billion (U.S.).[4] It is worth noting that the patch to fix the IIS exploit used by Code Red was published four weeks before the release of Code Red I v1.[5] Similarly, the patch for the SQL Server exploit used by Slammer was published on 24 July 2002[6], six months before Slammer was released. Further, while most worms fade out-of-sight and out-of-mind shortly after their initial propagation phase, they do not die. A study of Internet background radiation done in 2004 shows that in the three networks surveyed that Code Red traffic still accounted for 0.5% of all traffic seen.[7] One cause of this is unpatched machines being connected to the Internet and infected before the needed patches are downloaded and installed.

### 2.1.2 Increasing Virulence

The worms discussed in section 2.1.1 did not realize their maximum potential damage because they used inefficient propagation schemes. For every susceptible machine during the spread of a new worm there is a race between the worm, which is trying to find the machine and infect it, and the system administrator taking steps to protect it through firewall rules, mail filters or anti-virus software. A worm that wastes time

trying to infect machines more than once or infect immune systems simply gives the administrator more time to build defenses. A 2002 paper [8], describes some well-known methods for increasing propagation rates: hit-list scanning, permutation scanning, and topological awareness. Hit-list scanning means targeting the initial infection attempts at machines known to be susceptible. The worm developer works to develop a list of a 25 000 to 50 000 addresses through port scans and traffic monitoring, then directs the worm to infect these addresses first. Permutation scanning works to minimize duplication of effort. All instances of the worm contain a common pseudo-random permutation of the address space. When started, a worm instance starts at the address after its own in the sequence. If it determines that a target machine has already been infected it means that another instance of the worm has already worked through this portion of the sequence, so the current instance picks a new random starting point. Finally, topological awareness is the straightforward concept that infecting the machine across the room is a faster process than trying one around the world. The Code Red II worm attempted to do this with the refined propagation approach of selecting addresses in the same sub-net more often. The paper argues that in designing defences against worms it must be assumed that worm developers are aware of these methods and will exploit them.

Simulation results in [8] comparing a Code Red-type worm to a theoretical worm utilizing a hit list and permutation scanning, termed a Warhol worm, show that these methods result in a significant improvements in worm performance. The simulated network contained 300 000 machines scattered through a $2^{32}$ address space. Every machine was susceptible to the worm. The Warhol worm infected 99.99% of machines over 30 times faster, in 15 minutes versus eight hours. At that propagation rate system administrators are simply unable to react in time. A Warhol worm combined with

what is termed a zero-day exploit, one that is exploited on the same day or even before it becomes public knowledge, is capable of infecting nearly every susceptible machine before humans are even aware of it.

## 2.2 Bloom Filters

The linkage between the material covered in this section and the worms discussed previously is that some algorithms for detecting worms, including the Source Flow Counting (SFC) algorithm proposed here, use Bloom Filters in their operation.

### 2.2.1 The Bloom Filter

Proposed in 1970 in [9] by Burton Bloom, the Bloom Filter (BF) is a method of representing a set $S = \{s_1, s_2, \ldots, s_n\}$ of keys from a universe $U$, by using a bit vector $V$ of length $m = O(n)$ bits. The operations to insert an element into a BF and to query it for the presence of an element have low computational demands. The vector $V$ has a predefined length $m$, which simplifies the design of systems incorporating the BF.

All bits in the vector $V$ are initially set to zero. The BF uses $k$ independent hash functions $h_1, h_2, \ldots, h_k$ to map keys in $U$ to the range $\{1, \ldots, m\}$. One possible form for the hash functions is $h_{a,b}(s) = (as + b \mod m)$, where $0 < a < m$ and $0 \leq b < m$. [10] With this family of hash functions, $m$ must be a prime number to ensure an even distribution across the range $\{1, \ldots, m\}$. All variables are integers.

To insert an element $s \in S$ into the filter, the bits in vector $V$ at positions $h_1(s), h_2(s), \ldots, h_k(s)$ are set to one. Figure 2.2 shows the insertion of an element $s \in S$ into the filter. $s$ is processed by the hash functions to generate $k$ bit locations

$b_1, b_2, \ldots, b_k$. The bits at these locations in $V$ are set to one. The other bits in $V$ are left in their current state.

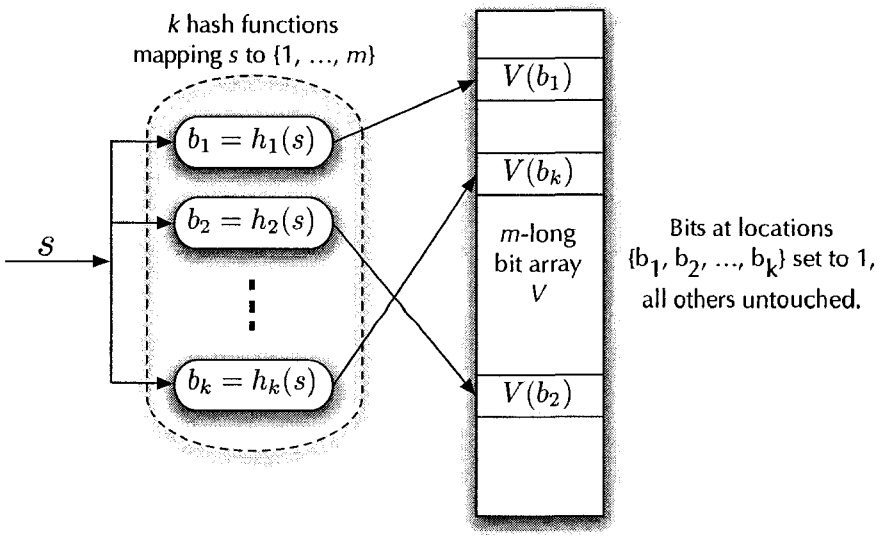


Figure 2.2: Bloom Filter Insert Operation: the element $s$ is hashed by $k$ hash functions to generate the locations $\{b_1, \ldots, b_k\}$. The bits in $V$ at those locations are set to one.

A Bloom Filter can be asked if it contains a specific element. The query operation that does this is illustrated in Figure 2.3. This operation begins as the insert operation does, using the hash functions to calculate $k$ bit locations from $s$. The difference is that instead of setting those bits to one, the query operation examines all $k$ bits. If all are one, then the element is present in the filter and the query operation returns a one. If any of the bits are zero, then element is not present and the query comes back with zero.

Query operations into a BF may return false positives that report an element as present when it was never added. This is due to the overlapping of the $k$ bits associated with each key. For example, set $k$ to three. Every element inserted into the filter sets three bits to one. Inserting an element, $s_1$, sets bits $\{2, 56, 45\}$. Inserting a

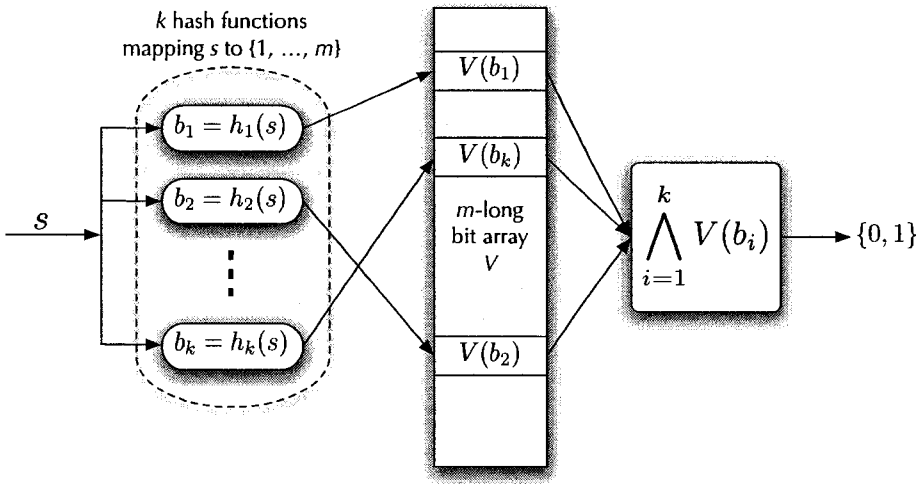


Figure 2.3: Bloom Filter Query Operation: to determine if the element $s$ is in the filter, the bits in $V$ at locations $\{b_1, \ldots, b_k\}$ are read. If all are one, then $s$ is present in the filter and the query operation returns one. Otherwise, $s$ is not present and it returns zero.

second, different, element, $s_2$ sets bits $\{45, 32, 7\}$. Bit 45 is common to both elements. While the hash functions should be selected such that no two elements generate exactly the same bit locations, nothing prevents this overlapping. Further, element $s_3$, which is never inserted, hashes to locations $\{2, 7, 56\}$. When queried, the BF will report $s_3$ as present. This false positive error is denoted the Bloom error. It is a deterministic value given by equation (2.1) [9].

$$E_b = (1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{\frac{-kn}{m}})^k \tag{2.1}$$

$E_b$ is the probability that any group of $k$ bits are all one after $n$ elements are inserted into an empty filter, assuming that the element corresponding to those $k$ bits was not inserted. After the first element is inserted, the probability that a bit is zero

is $(1 - \frac{1}{m})^k$. After $n$ insertions, the probability that the bit is zero is $(1 - \frac{1}{m})^{kn}$ and the probability that the bit is one is $1 - (1 - \frac{1}{m})^{kn}$. Expanding this to $k$ bits gives equation (2.1). The approximated error is minimized when $k = ln(2) * \frac{m}{n}$, resulting in a false positive rate of $E_b = (0.6185)^{\frac{m}{n}}$. This makes the Bloom Filter very effective when $m = cn$ for small values of a constant $c$. For example, when $c = 8$ the false positive rate is approximately 0.0214. One practical method of effectively reducing $E_b$ is to keep the filter less than half full, that is that less than half of the bits in $V$ are one.

Unlike querying a database, tree or any other similar data structure, it is not possible to ask a BF for a list of the elements it contains. In a tree data structure, an entry is created for each element inserted containing the element's identifying information. A query can then iterate through each entry and return the complete list. In the BF, the insert operation does not record the element's identifying information. Instead, a non-unique set of $k$ bits in $V$ are set to one. Not only will the bits for different elements partially overlap, but if $|U| \gg m$ then its very likely that more than one element in $U$ will map to the same bits in $V$. Therefore, it is impossible to uniquely identify the filter's contents simply by examining $V$ and attempting to reverse the hash functions. This is a fundamental limitation of the BF.

Equation (2.2) may be used as a guideline when considering which values of $k$ and $m$ to use. $\gamma$ is the ratio between the number of items hashed into the filter and the length of the bit vector $V$. In the minimized error case, $\gamma = ln(2) \approx 0.7$. $n$ should be set to the expected magnitude of $S$, after which selecting $m$ and $k$ is a matter of balancing the memory requirements of the bit vector and the computational

overheard of $k$ hash functions calculations for each insert and query operation.

$$\gamma = \frac{nk}{m} \tag{2.2}$$

### 2.2.2 The Spectral Bloom Filter

The Spectral Bloom Filter (SBF) [11] extends the Bloom Filter to multi-sets. Querying the filter for an element returns an estimate of the element's multiplicity. The SBF achieves this by changing $V$ from a bit vector to a $m$-long array of counters. The counters are all set to zero during filter initialization. The insert operation is very similar to the BF's. The element is hashed to generate $k$ locations. The difference is that the SBF insert operation increments those $k$ counters in $V$ by one. Starting with an empty filter, after inserting an element $s_1$ once, all $k$ counters for that element have the value one. Inserting that element again raises the counters' value to two. Subsequent inserts of elements who's counters overlap those of $s_1$ means that some of $s_1$'s counters will be greater than two. The SBF query operation compares all $k$ counters for an element and returns the lowest counter value as the element's multiplicity estimate. For $s_1$, this is still two.

Just as the BF will occasional report an element as present in the filter when it was never added, the SBF will occasionally report an element's multiplicity to be higher than it really is. This happens when all of the counters for an element are incremented due to overlap with the counters for other elements added to the filter. The SBF guarantees that it will never underestimate an element's multiplicity but that it may overestimate it. This includes reporting a multiplicity greater than zero for an element that was not inserted into the filter, which is the Bloom error. The

probability of overestimating the multiplicity, $E_{SBF}$ is the same as the Bloom error, $E_b$.

Compared to the Bloom Filter, the SBF consumes additional memory and the insertion and query operations are more involved. However, the multiplicity feature of the SBF allows it to serve many more applications.

# Chapter 3

# The Current State of the Art

Research into the Intrusion Detection System (IDS) area, particularly work focused on detection of network worms, began in earnest around 1999 and has been an area of active research since. Published works have proposed a wide range of ideas and methods for the detection of worms at run-time and off-line by using packet signatures, rules on traffic and searching for anomalies all running on existing router or host platforms or varying complexities of customized hardware. No single solution or approach has yet emerged as a clear winner or received widespread deployment. This chapter describes many of the currently published works and categorizes them by approach for comparison.

## 3.1 Detecting Known Threats

The easiest threats to detect are the known ones and that is where IDS research began. Existing worms are characterized by Internet Protocol (IP), Transmission Control Protocol (TCP) and Unigram Data Protocol (UDP) header fields, packet

payloads and scanning behaviour. The systems in this section maintain a database of threats and compare network traffic against them. These provide strong defences against these threats but the requirement to process all packets at line rate results in substantial costs that limit deployment. A potentially fatal limitation of this approach is the focus on keeping all known threats out as this means letting new and unknown threats in. The systems described are all vulnerable to new worms until their databases are updated. Before that happens, the worm may have penetrated into the network being protected.

### 3.1.1 Rule-Based Methods

A good base from which compare IDS approaches is Snort, proposed by [12] in 1999. It is a lightweight detection system designed for deployment as a permanent element of a network's overall IDS. Licensed under the GNU Public License [13], the program is built on top of the libpcap [14] promiscuous packet sniffing library and compares all packets flowing through the monitoring point against user-defined rules that perform content and header matching. (Content matching is done against the application-level content in the packet.) When a packet meets the conditions described by a rule, it is either silently dropped, logged, or an alert is generated depending on what action is attached to the rule. A strength of Snort is the simplicity of the language used to describe rules. Figure 3.1 shows a simple rule looking for any finger (port 79) traffic sent to the 10.1.1.0 class C network. A more complicated rule shown in Figure 3.2 examines incoming Internet Message Access Protocol (IMAP) traffic (port 143) to a specific server for a buffer overflow exploit. In this figure, the characters inside the $|\ldots|$ are hex values for non-printable characters. This simplicity allows for rapid development and deployment of new rules as new threats are identified. Snort's

authors state that when the Internet Information Server (IIS) Showcode web exploit was first publicly described, new rules to detect probes for it were publicly available within a few hours. Snort rules are very flexible and can check packet content, TCP flags, IP Time To Live (TTL) values, Internet Control Message Protocol (ICMP) type and code fields, IP fragment size, IP id values, TCP sequence numbers and packet payload size. These enable Snort to detect a wide range of attacks such as known worms, port scans and Server Message Block (SMB) probes. (Snort development has continued since the original 1999 release. The current release is technically quite different but uses the same operating concepts.)

```
log tcp any any -> 10.1.1.0/24 79
```

Figure 3.1: A Simple Snort Rule: directs Snort to all log finger (TCP port 79) from any IP address and any TCP port destined for the 10.1.1.0 class C network. [12]

```
alert tcp any any -> 192.168.1.0/24 143 (content:''|E8C0 FFFF
 FF|/bin/sh''; msg:''New IMAP Buffer Overflow detected!'';)
```

Figure 3.2: Snort Alert for IMAP Buffer Overflow Exploit: monitors IMAP traffic destined for a specific vulnerable server for a buffer exploit attack. [12]

An even simpler approach than Snort is the "Billy Goat" [15]. A Billy Goat is a dedicated machine on a network that offers and uses no services, meaning it should never receive or generate any traffic. Any traffic it does receive is likely malicious. A random scanning probe of the Billy Goat by a worm sounds an alarm to administrators. The Billy Goat emulates the services offered by other machines in the network in order to gather properties of the connection that can be used to identify and eliminate the worm.

The authors of [16] describe a behavioural detection system based on the Coloured Petri Net (CPN). The behaviour of a worm, including file and Windows Registry modifications, network activity and memory contents, is characterized as a series of states and transitions in a CPN. Once all of the worm's activities have occurred, the infection is reported and handled. Administrators describe worms using a custom Attack Definition Language (ADL), which is then compiled into a CPN. The language supports some generalizations to allow a single CPN to handle multiple variants of a common base worm type. While the system is primarily aimed at catching known worms, its generalization abilities give it a limited ability to detect new worms that are sufficiently similar to an existing CPN. There is also a learning capability to automatically refine an existing CPN to exclude conditions that cause false alarms. This design is intended for deployment on PCs rather than in a router. Figure 3.3 gives the ADL for the W32.Netsky worm, which creates and deletes some objects in the Registry, creates a file and generates some Simple Mail Transfer Protocol (SMTP) traffic. Each activity is named using the `#define XN` statements. The CPN shown in Figure 3.4 was generated from this ADL. The groupings of file, registry and network events become starting points in the CPN. To reach the Final state, all events described in the ADL must have happened. Transition D is where the system takes action to block or eliminate the worm. The authors have only proposed and have not yet implemented this design.

### Payload Analysis for Known Signatures

Snort falls into the general category of signature detection methods, approaches that compare traffic against a library of known properties of dangerous traffic. Recent research in this area has focused on expanding the matching capabilities of these

```
Attack ''Netsky'' [Regmon, Filemon, Netmon]
Regmon {
add object= ''HKEY_LOCAL_MACHINE\SOFTWARE\...\Run\''
result = $FILE1 [$FILE2 IN ($FILE1)] #define R1

del object=''HKEY_LOCAL_MACHINE\SOFTWARE\...\Run\''
result='''' #define R2

del object=''HKEY_LOCAL_MACHINE\CLSID\...\InProcServer32''
result='''' #define R3}
Filemon { add object=c:\winnt\$FILE2.  #define F1}
Netmon {
SMTP #define N1}
```

Figure 3.3: W32.Netsky Worm in Attack Detection Language: describes the worm in terms of it actions on the registry, file system and outgoing network traffic. Portions of the Windows Registry paths have been replaced with ... for brevity [16]

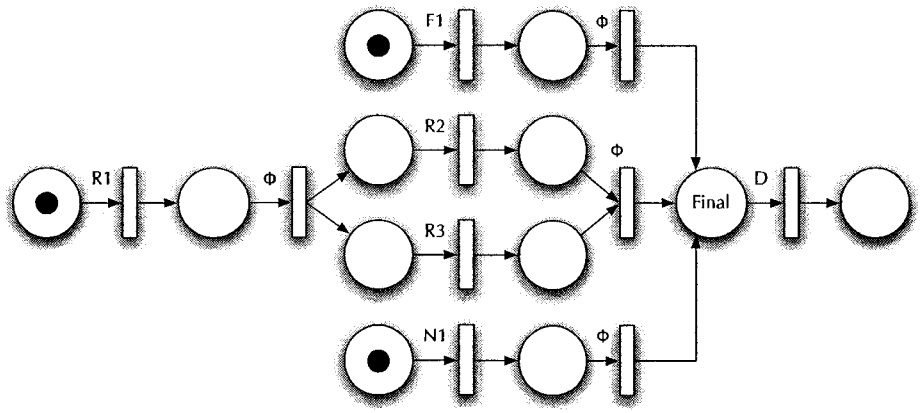


Figure 3.4: CPN W32.Netsky Model: this model corresponds to the ADL in Figure 3.3 [17]

systems and making them operate at line speed. This often requires dedicated hardware and results in costly systems that can only detect known worms. These systems must also devote resources to reassembling fragmented packets and sessions in order to have complete payloads to process. Compared to Snort and other software-only packet scanners, these hardware solutions are considerably faster and target line rate monitoring of links.

Attempting to address the challenge of content matching, often called deep packet filtering, at line speed, [17] and [18] have devised solutions that employ Ternary Content Addressable Memory (TCAM), an extension of Content Addressable Memory (CAM), to perform rapid matching of packet contents against worm signatures. A CAM does a bit-by-bit comparison of an input search word against a table of stored data and outputs the address of the matching table entry. As illustrated in Figure 3.5, CAM compares the search word stored in the search register against the stored words in the memory. The matchlines for each stored word signal if there is a match and the encoder generates a single memory address to output. The search word is compared against all stored words simultaneously, making this a very fast method to do content matching. The Ternary extension to CAM adds wildcard matching through a don't care state. Each cell in the stored data, representing a bit, is either zero, one or "–", meaning don't care. Cells containing "–" always match. If all cells in a stored word match the corresponding bit in the search word, then the matchline reports a match for that row. The fast search capabilities of CAM come at the cost of increased silicon space and power consumption compared to regular memories.

The Discrete Content sensitive Pattern Match algorithm for Imperceptible Deep packet Filtering and a System (sic) proposed by [18] in 2003 uses TCAM to compare packets passing through the detection point against a stored library of known

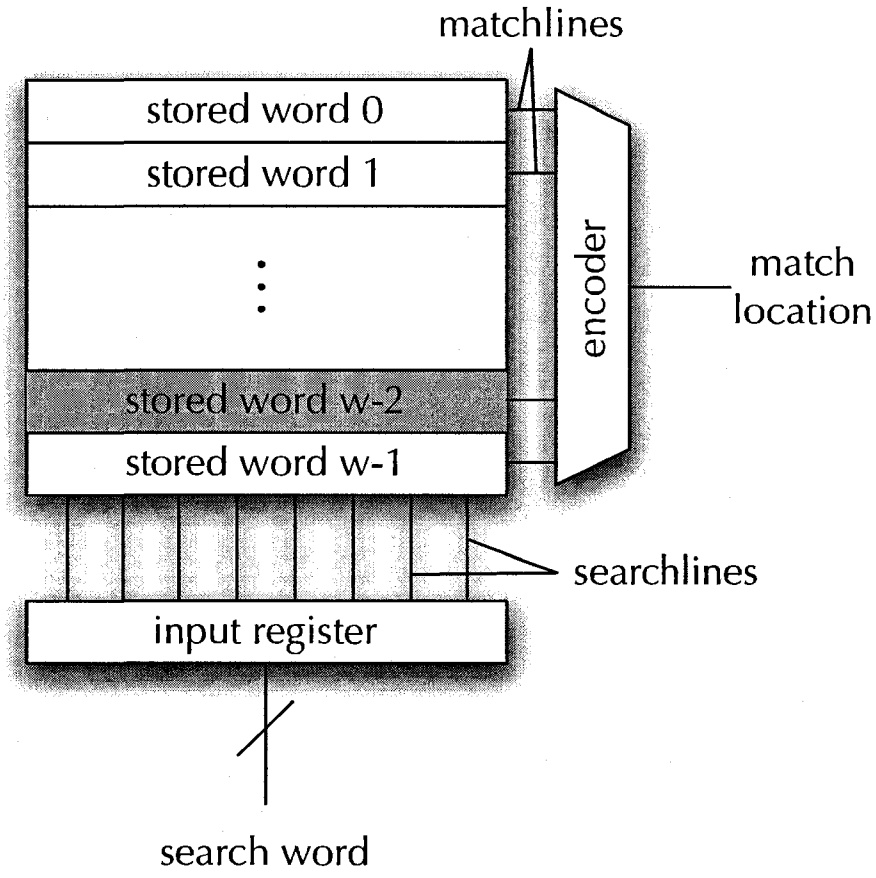


Figure 3.5: Content Addressable Memory: the input register contents are simultaneously compared against all stored words. [19]

worm signatures at line rate on a gigabit Ethernet link. The basic behaviour is as illustrated in Figure 3.6. Incoming packets are compared against the TCAM by the Search/classification block. The width of the TCAM search word is far less than a typical packet payload, so this system uses a sliding window approach to compare the entire payload against the signature library. Say the TCAM is two bytes wide. The first two bytes of the payload are compared, then the second and third bytes, then the third and fourth bytes and so on until the entire payload is compared. This has a significant impact on throughput. If a match is found, the instructions on how to handle that packet are stored in the Static Random Access Memory (SRAM) and executed by the Modify block. However, comparing payloads against the TCAM contents is still not fast enough to handle packets at the desired rate, so the authors devised a second method to augment TCAM matching and improve throughput. Once a payload has been matched by the TCAM, an entry is made into the "self-study" table containing a subset of the IP header fields such as source port, destination port, and level four protocol. Future packets are compared against the self-study table first as this is a much simpler and faster operation. The self-study table contents are dynamic and replaced as newer worms are encountered. This system is limited to signatures that are no longer than the TCAM is wide.

The TCAM-based worm detection system proposed by [17] in 2006 took advantage of improvements in TCAM technology to target a 10 Gbps line rate. Their approach supports all signatures from both Snort and the ClamAV [20] projects, both of which produce signature libraries for known worms. The system supports both the short 25 byte or less Snort content signatures and the longer ClamAV signatures, which are typically larger than 50 bytes. TCAM memory is too expensive to store the entirety of each ClamAV rule. Instead, the rule is split into a short prefix and a

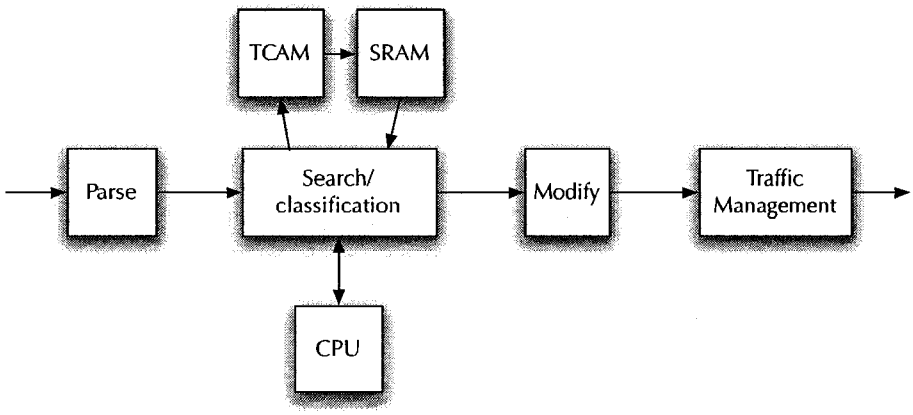


Figure 3.6: Packet Classification Engine: packet payloads are compared against the TCAM. If a match is found, the Modify block executes instructions stored in the SRAM for how to handle packets with the specific payload. [18]

longer tail. The prefix is stored in the TCAM along with a hash of the tail. On a packet arrival, the TCAM is searched for both a prefix match and tail hash match. If a match is found, the entire payload is compared against the complete signature stored in Random Access Memory (RAM). In the TCAM, each position-aware sub-pattern is stored in its own entry to speed matching. For example, say the pattern being matched against is "GATT". In a four byte-wide TCAM, this pattern has the following Position-Aware Sub-patterns (PAS), where "–" is the don't care symbol: "GATT", "–GATT——", "—GATT—" and "——GATT–". The longer patterns are divided into two TCAM entries. The Central Processing Unit (CPU) running the matching system maintains a state diagram to know that when byte $x$ of a packet matches "—GA" and the $x+1$ byte matches "TT—" then there is a signature match. This is illustrated in Figure 3.7. This figure also shows that when another signature generates the same PAS, they are shared in the TCAM to save space.

The Floating Programmable Gate Array (FPGA) is a commonly used component

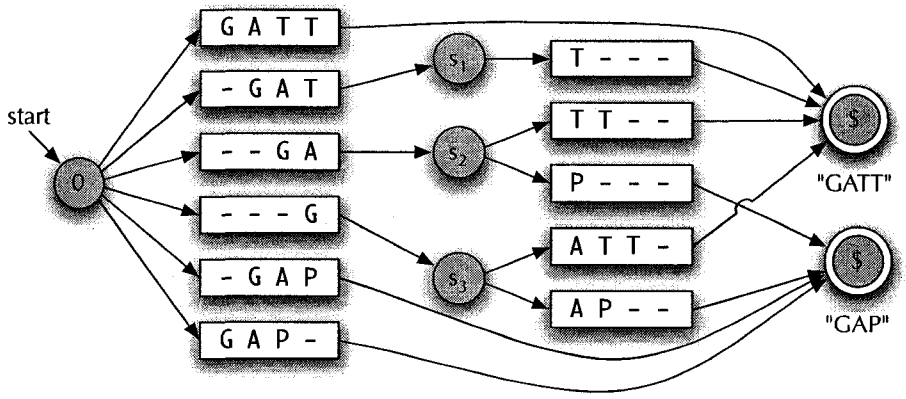


Figure 3.7: TCAM Matching State Diagram: shows all possible states for matching "GATT" and "GAP". [17]

in hardware detection systems. One solution built on FPGAs is presented by [21]. The system consists of three modules. The Data Enabling Device (DED) contains a FPGA configured to do fast processing of incoming packets against a library of known signatures. DEDs are placed throughout the network. Groups of DEDs report their findings to a Regional Transaction Processor (RTP), which directs a DED on what to do with suspicious traffic. Finally, the Content Matching Server (CMS) is used by the administrator to configure the DEDs and the RTPs. What makes this approach truly interesting is how the FPGAs in the DEDs operate. Rather than have each FPGA contain a generic matching engine that draws signatures from a memory, each signature being watched for is implemented as a custom circuit in the FPGA. When a new packet arrives all signatures are checked for in parallel by these custom circuits. Administrators use a dedicated toolchain on the CMS to enter a new worm signature, generate a detection circuit for it, synthesize the new FPGA image and push the new image to each DED as well as set the resulting behaviour in each RTP. The authors state that a new FPGA image can be generated in as little as nine minutes and that

each FPGA can hold up to 10 000 fixed-length strings.

Two recent works have used Bloom Filters to perform signature matching on hardware. The approaches of [22] and [23] are similar but have some significant differences. Both are built on Counting Bloom Filters (CBF) [24], an extension to Bloom Filters (BFs) that pairs the bit vector with an array of counters. Inserts set both the bits and increment the corresponding counters. The CBF supports removals: decrement the counters and clear the bit only if its counter is zero. Both systems hash the worm patterns into CBFs and compare hashes of a sliding window on the data against the filter contents. If query returns true, the packet is sent to a subsystem for further comparison as the true might have been a false positive. CBFs are used to support removal of out-of-date signatures from the filters during runtime. Despite being published two years after [22], [23] supports only 128 signatures of a fixed 16-byte length. The earlier proposal can handle up to 10 000 variable length strings. In the [23] system, six of the micro-engines on an Intel IXP network processor are used to implement identical BFs in parallel allowing six payloads to be processed simultaneously, one byte shift at a time. The [22] system contains multiple CBFs, one for each signature length. Parallelism is used to process payloads in multi-byte steps. The system is implemented on a FPGA platform.

## 3.2 Detecting Unknown Threats

The methods discussed in 3.1.1 require a worm's signature in order to detect it, exposing them to the zero day vulnerability. Automated generation of signatures is one way of addressing this vulnerability but this remains a less explored research area. Methods that incorporate other indicators of worm infections are discussed in 3.2.2.

### 3.2.1 Generating Signatures from Payload Analysis

The real-time work detector proposed by [25] monitors a streaming window of payload data for repeated strings. The administrator sets both a threshold and a timeout period. Put simply, when a string is seen more than threshold times within the timeout period, an alert is generated. The actual operation is slightly more complicated. Figure 3.8 shows the block diagram of the hardware system. Bytes are shifted one at a time through the sliding window. The Character Filter removes common strings such as "http://" and innocuous characters from the window to minimize false alarms. The payload is then hashed to produce an index into the Count Vector and the selected counter is incremented. When a counter reaches the threshold, an alert is generated and the counter is reset to zero. At the end of each timeout period, the counter values are reduced by the average number of arrivals per counter in that time period. The SRAM Analyzer block is used to store suspicious strings and their occurrence counts. When a Count Vector counter reaches the threshold, the string is hashed and compared against the SRAM contents for that string. If they match, then an associated SRAM counter is incremented. Otherwise, the existing string is overwritten with the new string and the SRAM counter is reset. The hardware uses the SRAM contents to filter alarms generated from the Count Vector values and reduce the false alarm rate.

### 3.2.2 Anomaly Detection Through Traffic Analysis

The methods in this category monitor headers, payloads and network metrics for anomalous behaviour, such as repeated packets from one source to multiple destinations, SYN floods or a change in the statistical distribution of some network property.

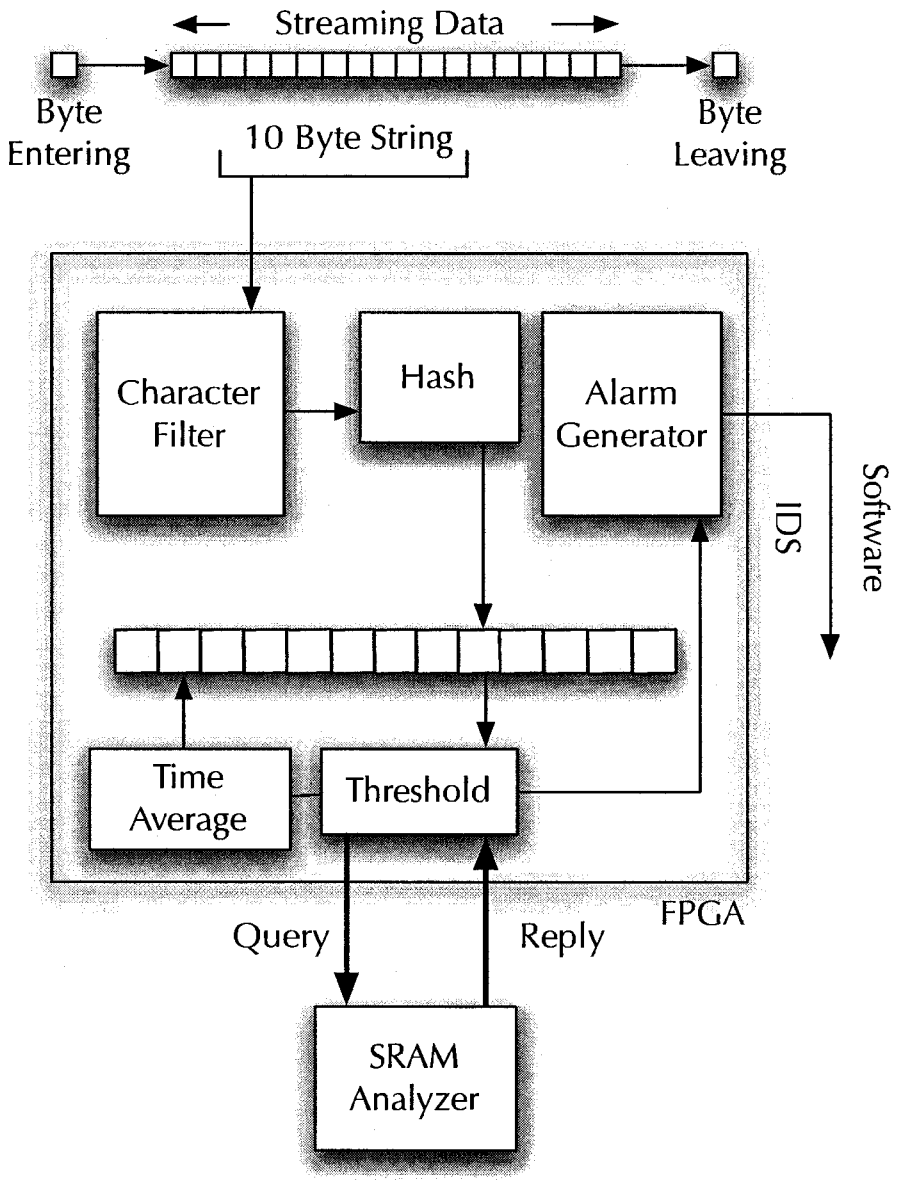


Figure 3.8: Real-time Worm Detector Block Diagram: this system monitors the frequency of repeated payloads and sounds an alarm when a threshold is exceeded. [25]

The approaches described look for these general properties of worms in an attempt to detect the widest possible range of worms with no prior knowledge of them. These solutions are typically less computationally intense than deep packet scanning and are better suited for implementation in software on general computing platforms.

A very simple approach called Destination-Source Correlation (DSC), which is focused on active scanning worms and that is capable of preventing zero-day attacks, was presented by [26]. When active scanning worms propagate, the infected hosts send probes to random addresses. Newly infected hosts then start sending probes in a similar fashion. The DSC algorithm records the destination address and destination port pair of all packets in the monitored network. A host is identified as possibly infected when the source of packet is found in the list of previously seen destinations for that port. An alert is issued when that source continually issues packets on that port. Technically, DSC is implemented with two BFs for each port monitored. The BFs, each used for time $T_b$, are used in an overlapping fashion to form a sliding window that is $1.5T_b$ long. When a destination-source pair is found it is added to a second list for further monitoring. The authors acknowledge that DSC will issue alerts for some forms of valid traffic, such as Gnutella or a busy SMTP relay, and they are looking for methods to mitigate this without creating new exploitable infection vectors. Despite this, DSC has two key strengths that should not be overlooked: speed and zero-day invulnerability. The simplicity of the DSC approach means that it can be implemented in software on general platforms, unlike the hardware solutions described previously.

Recent work by [27] proposed Frequency Detection-based Filtering (FDF) of scanning worms. The authors analysed two captured traces of network data and determined that scanning worm traffic from a host is generated at a constant frequency,

while legitimate traffic spreads out over the entire frequency band. In this case, frequency refers to the creation of new TCP sessions or UDP probes to different destinations. For example, the Code Red worm sent TCP probes about 11 times per second, while SQL Slammer sent UDP probes 4000 times per second. As most of these probes failed in about the same time, they were evenly distributed in time. In comparison, normal background traffic is generally created in response to user activity and does not create sessions with such regularity. This became the basis for FDF, which records the frequency of observed traffic grouped by source address and destination port. Entries with a constant frequency are considered worm traffic.

As an active scanning worm attempts to propagate, it is inevitable that some of the addresses it tries to contact will be unreachable. When this happens, one of the routers along the route may return an ICMP type 3 (ICMP-T3) message to the source. The worm detector described by [28] consists of modified routers that send a blind-carbon-copy of the ICMP message to a collection point, which looks for traffic patterns indicative of a worm infection. The authors supported their approach by measuring the number of ICMP-T3 messages contained in captured Nimda worm traffic. Of the hosts contacted, 2.74% returned ACK/SYN, 2.83% returned RST and 26.45% of infection attempts resulted in an ICMP-T3 being sent back to the source. The remaining approximately 68% of connection attempt had no response. What is significant from these results is that of the responses seen, 82.57% were ICMP-T3. The paper does not address what seems to be an obvious concern: the impact on the Internet of both the worm's propagation and the potentially large volume of BCC'd ICMP-T3 messages being forwarded to the processing stations.

Connection chains, proposed by [29] and refined by [30], are a sequence of connections $\{C_1, C_2, \ldots, C_l\}$ satisfying two conditions:

- the destination of connection $C_{i-1}$ is the source of connection $C_i$, where $1 \leq i \leq l$
- the time of connection of $C_i <$ the time of connection of $C_{i+x}$ where $0 < x \leq l-i$

The chains are well suited to recording worm propagation, where a host is infected by receiving a connection and then proceeds to infect those around it. A connection item contains a time stamp, source address and port, destination address and port and the first $n$ bytes of the payload. Worm detection is done through analysis of the chains for indicators such as repeated destination ports and payloads. Each connection emanating from suspected hosts are assigned a score that takes into account the number of chains that include the node and the whether the connection attempt was to a non-existent host or service on the destination. Hosts with scores that exceed a user-defined threshold are considered infected.

Several approaches key on looking for repeated traffic properties on the assumption that too much of the same is a bad thing, particularly events like repeated packet contents being sent to the same destination port at many different addresses. [31] generates fingerprints based on packet content and destination port and maintains counters for each. When a counter passes a user-defined threshold, additional counters are used to track the number of source and destination addresses involved. Alerts are generated when the number of sources, destinations or source-destination pairs for that fingerprint exceeds some other threshold. The method does not take a single hash of the entire payload but rather makes multiple hashes of subsets of the payload to address the risk that the worm mutates the payload each time. The Detector for Early Worm Propagation (DEWP) system described in [32] takes a simpler approach. The filter monitors both incoming and outgoing traffic streams. When the destination ports of two packets sampled from the stream match, the port is flagged

as suspicious. This action is based on the idea that when a worm is propagating the increase in traffic to the targeted port increases in both directions. However, this is not sufficient evidence of a worm, so DEWP then takes the second step of counting distinct destination addresses for the suspect ports and sounds the alarm when a user-defined threshold is reached. Finally, a research group at Carleton University [33] concatenated the destination port to the payload and hashed the resulting string into a Bloom Filter With Counters (BFWC), a data structure similar to the Spectral Bloom Filter (SBF). This method is aimed at detecting repeated packets and issues an alert when a specific packet is seen more than a threshold number of times.

Detection of an abrupt change in one or more network metrics is another method used to identify a worm infection in progress. The method of [34] uses auto regression to detect correlated changes in Simple Network Monitoring Protocol (SNMP) Management Information Base (MIB) variables. Specifically, these are the three MIB variables that measure the flow of IP datagrams through a router being monitored: total number received; number delivered to higher levels of this node; and number originated by higher levels. A correlated increase in two of the three variables is indicative of a worm's propagation. A later work [35] uses the same three SNMP MIB IP variables and includes two others that count the total number of incoming and outgoing octets. Taking a different approach, [36] processes flow variables, rather than datagram or octet ones, using information gathered by Cisco's NetFlow monitoring software. A training phase establishes the baseline behaviour of hosts on the network. During the operational phase, alerts are generated when the number of flows generated by a host is more than two standard deviations above the normal. However, this method is an off-line solution and therefore not suitable for run-time detection.

Further illustrating the broad range of ideas that fill the field of worm detection, these next three papers each use statistical modelling of network traffic in different ways. The packet size of many types of normal network traffic has a heavy-tailed distribution. In [37], it is discovered that the size distribution of the first packet sent from a benign source to a new destination is also heavy-tailed but that this is not true for infected hosts. Their method records the size of these first-contact connections and calculates a heavy-tailed metric for their distribution. As this is a computationally intense process, a simpler check of the proportion of failed connection attempts is performed first. If either exceeds an allowable threshold, the host is considered infected. Multi-similarity is the basis of the detection method by [38]. Noting that worm infections skew the distributions of destination ports, packet sizes and the proportion of TCP, UDP and ICMP traffic. After a training period, the method looks for changes in more than one measure to indicate a worm. The third approach [39] uses two different modelling methods. Both require a training phase of allowable traffic to generate the models. Packet Header Anomaly Detection (PHAD) monitors 33 fields in the Ethernet, IP and TCP, UDP or ICMP headers. A table maintains the allowable range of values for each field and the ratio of unique values seen to total packets processed that have that field ($r/n$). At runtime, every time a new value is seen it is given an anomaly score of $tn/r$ where $t$ is the time since the last observed anomaly. The sum of the anomaly scores is maintained and an alert is sounded when it exceeds a threshold. The second method is called Application Layer Anomaly Detection and models TCP traffic only. The training phase is used to generate five sets of probability models:

- $P(src\ IP|dest\ IP)$: models the set of clients allowed on a restrictive service;

- $P(src\ IP|dest\ IP, destport)$: like the previous model but with per-port granularity;
- $P(dest\ IP, dest\ port)$: learns which servers normally receive requests;
- $P(TCP\ flags|dest\ port)$: learns the set of TCP flags for the first, next-to-last and last packets of a connection in order to detect out-of-order operations;
- $P(keyword|dest\ port)$: used to monitor payloads for allowable content.

For the latter, keywords are the first non-white space characters after a line feed in the first 1000 characters of payload. For example, after training the keywords for port 80 (HTTP) included "Accept-Charset:" and "GET:". As with PHAD, anomalies are given a score and the sum of the scores is used to assess if a worm is active. This method appears to require a large amount of storage in order to maintain the allowable values for each IP and/or port.

# Chapter 4

# The Worm Detection Problem

## 4.1 Problem Statement

This thesis discusses the reasons for and the design of a passive IP-level network monitor for the run-time detection of active scanning worms. Passive network monitors draw conclusions about the performance and behaviour of a network through observation of the traffic carried by the network without injecting any traffic to aid in measurement. Run-time detection means using live observations rather than collecting and post-processing data. Active scanning worms are the category of network worms that use any variant of random generation of victim IP addresses. The design goals driving Source Flow Counting (SFC) were to minimize computation, storage and network overhead while maximizing detection accuracy and minimizing false alarms.

## 4.2 Gaps in Existing Solutions

As shown in Chapter 3, research into active scanning worm detection can be generally divided into rule- or signature-based methods and anomaly detection methods. When applied to the active scanning worm problem, the first type have major limitations for which there seems to be no escaping and that will realistically prevent their widespread deployment. These are the size of the signature or rule library, the cost of the specialized hardware needed to process traffic against the library at line rate and their vulnerability to new worms. While the latter is the most serious, this discussion will begin with the technical issues.

For maximum effectiveness, rule systems must maintain a substantial database of rules describing all known potential threats. Every new threat results in new entries in the database; however, as was discussed in 2.1.1, old worms do not truly die once the initial propagation phase is over. It is reasonable to conclude that the rate of new worm additions will exceed that of old worm removals and the database will grow continuously. Therefore, solutions like the Ternary Content Addressable Memory (TCAM)-based methods of [17], which was capable of holding the complete Snort and ClamAV databases at the time of publication, will eventually be unable to do so. As some of the solutions presented in 3.1.1 had capacities of up to 10 000 strings, the database size problem is not an immediate concern but looms as a long-term limitation. On a related note, the administrative challenge of deploying updates to the signature library to every monitoring node is not a trivial consideration. When a new worm is active on Internet, an update policy built around nodes downloading updates from an Internet site will likely fail due to network overload just at the time it is needed most.

Paramount when designing a deep-packet filter are maximum throughput and minimum latency. The approaches presented to date seemed to have ruled out packet matching in software as being simply too slow. The solutions presented in 3.1.1 address this through deployment of custom hardware that includes Floating Programmable Gate Arrays (FPGAs), TCAMs and dedicated processors. One example, the customized worm detection circuits within a FPGA design of [21] can process 2.4 Gbps on a single Xilinx Vertex FPGA; however, this does not take into account the second FPGA, 1 GB Synchronous Dynamic Random Access Memory (SDRAM) and 6 MB Static Random Access Memory (SRAM) needed to process and store the traffic flows being analysed. The most recent work presented, using an Intel IXP equipped with dual processing cores and a 9 Mbit TCAM, was the only design to target 10 Gbps Ethernet yet attained only 3.8 Gbps. [17] From these, it is clear that deep-packet filtering is not approach that will yield worm detection methods affordable enough to be widely deployable while being fast enough to avoid constraining the network being monitored.

Throughput, latency and database concerns are secondary to the primary limitation of rule-based systems: the zero-day vulnerability. While it is very important to block known threats, the biggest threat is always the next unknown one. The growth of newly released worms is explosive, going from only a few infected systems to worldwide penetration within the space of hours [2], very likely faster than the time needed for human or even automated systems to determine a suitable rule and deploy it to all the monitoring nodes. Unless the new worm is a close derivative of an existing one, rule-based systems such as the Coloured Petri Net (CPN) approach of [16] will allow it to penetrate into the networks they are protecting. With such limitations on rule-based Intrusion Detection System (IDS), there exists a strong need for an IDS

based on anomaly detection that looks for the generalized behaviour of all scanning worms: the traffic changes caused by probing.

The Billy Goat defence [15] is a good single host solution to detecting traffic changes, except that it cannot respond until it is scanned by the worm. This makes the argument for network-level monitoring. If the Billy Goat is not targeted by the worm until later in the infection process, many hosts in the network it is protecting may already be infected. However, the routers or switches serving that network will already have handled a potentially large volume of worm traffic and are in a position to see the properties of the network's traffic change to those indicative of a random scanning worm. Therefore, it is logical to locate the detection process on these platforms and to use network-level monitoring.

A final consideration in worm detection is that the solution must not exacerbate the problem. A specific example of this is the Internet Control Message Protocol (ICMP) forwarding system [28]. This system required monitoring nodes to forward copies of ICMP-T3 messages generated by that station to a central monitoring point for correlation of port and source information. During a major worm outbreak, this solution would inundate the network with ICMP messages that, when combined with the actual worm traffic, could have devastating consequences.

In direct contrast to the complexity of deep-packet filtering methods of identifying worm outbreaks, approaches built on anomaly detection are generally simple, fast and invulnerable to zero-day attacks. As discussed in 3.2.2, these methods monitor the network traffic flow looking for specific indicators of worm propagation such as destination machines beginning to produce packets very similar to ones they recently received, a large number of packets traversing the network with very similar payloads or a sudden increase in traffic volume through a router. The narrow focus of each

approach does mean that it is possible for worm developers to include specific countermeasures in their worm programs. This requires *a priori* knowledge of the detection scheme in use on a target network, which may not be easy to come by. Further, the processing demands of anomaly detection schemes are typically low enough that two or more may be combined on a single monitoring point to counter such attacks.

As is common with all simple takes on a complex problem, these anomaly detection schemes are occasionally wrong. They have issues with both false alarms, identifying traffic as suspect when it is not, and false negatives, declaring worm traffic as benign. While the rates for both of these can be reduced through training or parameter adjustment, they can never be pushed to zero as there remains network traffic such as Gnutella that has worm-like properties. Recognizing these limits, there is value in pursuing these approaches because their simplicity is their strength. The benefits of simple implementation, potential for widespread deployment and defence against new attacks justify the effort. There remain properties of worm traffic for which no test has yet been developed.

## 4.3 Impacts of Worms

Despite ongoing efforts by software makers to eliminate all security vulnerabilities from their products, the complexity of current operating systems and applications means that this alone is unlikely to prevent future network worm events. Further, products that are secure on their own may be vulnerable when combined together. A major worm event that exploits flaws found on widely deployed systems has the potential for a massive economic impact from data and productivity loss. Finally, worms propagate across networks far faster than human administrators can react

to block them. These arguments lay out the rational for research into methods for real-time network-level detection of worm infections. Once detected, worms can be stopped or slowed long enough for administrators to determine the proper course of action.

# Chapter 5

# Source Flow Counting Worm Detection

This chapter begins with the algorithms and data structures behind the Source Flow Counting (SFC) worm detection method developed to meet the requirements set out in 4.1. It then discusses the simulation environment developed to validate this approach and closes with results taken from that simulation.

## 5.1 Source Flow Counting Algorithm

The worm detection algorithm developed for this thesis is based on locality, specifically the size of the "working set" of destination Internet Protocol (IP) addresses each source IP contacts in a given time period. The use of locality as an indicator of normal or abnormal behaviour was explored in [40]. The thesis of that paper is "that locality principles are a key to distinguishing and understanding 'normal' behaviour in computer systems that may be subject to attack by outsiders." Four examples are

studied to support the thesis: gross scale workstation connectivity, fine scale workstation connectivity, gross scale email addressing and fine scale email addressing. When applying locality to the random scanning worm problem, the pertinent example is fine scale workstation connectivity. Data was collected from NetFlow records for a large network and analysed to determine the working set size for each host. A result of this work is Figure 5.1, which clearly shows that the vast majority of sources contact less than five destinations per hour.

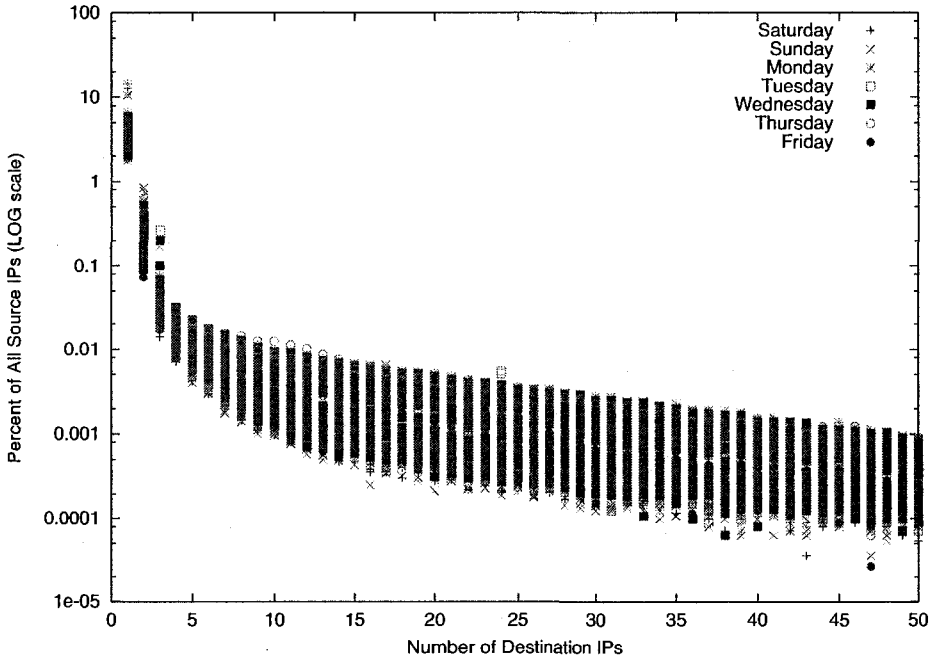


Figure 5.1: Distribution of the destination IP working set size up to 50 destinations over a one hour period [40]

It must be acknowledged that [40] is not an exhaustive study as the work relies on only a few data sets to support their conclusions on locality. However, the conclusions they reach about the value of the working set as an indicator of normal behaviour are rational. Consider a workstation PC on a desk in an office environment. During a

typical workday, when operated by an employee who controls their web surfing habits, this machine typically interacts with the local file servers, an email server, some print servers and the office intranet. Over the course of the entire day this is a working set of ten to 15 destinations. A normal random scanning worm quickly balloons the size of the working set into the thousands, an easily observable change. Further support for these conclusions is given by [41], who found that within the context of the network at the University of New Mexico the working set of destination IP addresses was completely identified within a few weeks of monitoring. Locality is therefore a solid basis for detecting worm activity through solely passive monitoring of traffic headers.

The Source Flow Counting algorithm detects worms by monitoring the working set of each host on the network under observation for abrupt increases. SFC divides time into equal-length slots, non-overlapping slots. At the start of a new time slot, all flow counters are set to zero. When a packet is observed being sent from a source to a new destination for that source in the current time slot, the flow counter for that source is incremented by one. At the end of the time slot, the algorithm calculates three statistics for each source: a flow count short-term mean encompassing a few of the most recent time slots, a flow count long-term mean that incorporates a much larger number of time slots and the variance of the long-term mean. These three statistics are used to determine whether a source is infected and are maintained in the SFC's data store. The time slots are also used to remove inactive sources from the stored data. If a source does not send an IP packet during the current time slot, its statistics are removed from the stored data set. If such a pruning did not occur, the data store would become overly burdened with valueless data.

When developing the SFC algorithm, it was assumed that hosts generally maintain a steady locality and that the primary cause of large and abrupt increases in locality is

worm activity. However, peer-to-peer applications typically cause an abrupt increase in locality similar to worms, though generally at a smaller magnitude. Distinguishing between peer-to-peer applications, such as Gnutella, and worms requires examination of packet headers and/or contents. SFC leaves this examination to other tools that would work in concert with SFC.

Potential monitoring points for SFC include any layer 3 router in the network. These monitoring points would not detect worm traffic passing between members of the same subnet but would catch worms as they probe destinations outside the subnet. Even worms that bias their probing attempts to the local subnet must probe at a high rate outside the subnet to ensure their successful propagation.

### 5.1.1 Use of the Spectral Bloom Filter

Well before the specifics of SFC were developed, the Spectral Bloom Filter (SBF) had been identified as an interesting and promising data structure for network traffic measurement and worm detection as a particular application. The SBF provides low-overhead insert and query operations to a compressed data store, all desirable properties in a network measurement storage system. The false positive ratio is low enough, particularly when the filter is prevented from becoming too full, to provide useful results in an application like worm detection where the occasional appearance of an incorrect value does not prevent the detector from operation. Pursuing a worm detector based on the SBF is a worthwhile task.

The SFC algorithm uses four SBF data structures to store the flow history for each source in the monitored network. Figure 5.2 presents a high-level overview of the data structures used by SFC. The centre portion of this diagram lists the four SBF structures. Once a key has been hashed by the $k$ hash functions, it can used to

access any of the SBFs as the same hash functions are used for all four. Additionally, the counter array length, $m$ of each SBF is the same. The Flow Counting SBF is a standard SBF, while the other three shown below it are Arbitrary Spectral Bloom Filter (ASBF). This extended SBF data structure is described in Chapter 5.1.3. The Flow Counting SBF is indexed by both the source IP address and the flow key, which is an identifier assigned to each flow seen. The three ASBFs are indexed by the source IP address only. The rightmost portion of the figure lists the values stored and output by each SBF. How these values are created, stored and used is described in the following sections.

### 5.1.2 Flow Counting and Flow History Storage

The SFC algorithm, as would any algorithm based on flow counts, requires a data store capable of maintaining both the per-source flow counts and the flow history, a record of all source-destination pairs seen during the current time slot. This is the Flow Counting SBF. A source's flow counter is incremented when a packet is observed with a source-destination pairing not yet seen during the current time slot. A flow key identifies a source-destination pair. It is generated by concatenating the lower 16 bits of the source and destination IP addresses. (Other methods of flow key generation, such as generating a hash of the two IP addresses, could be used. During the development of SFC a hashing scheme was experimented with. As it did not significantly improve results within the simulation, it was removed in favour of the simple hashing scheme used.) On each packet arrival, SFC generates the packet's flow key and inserts that flow key into the Flow Counting SBF, a process illustrated in Figure 5.3. If the insert operation returns a value greater than one, the flow has been seen previously and no further processing is needed. Otherwise, the flow is new and

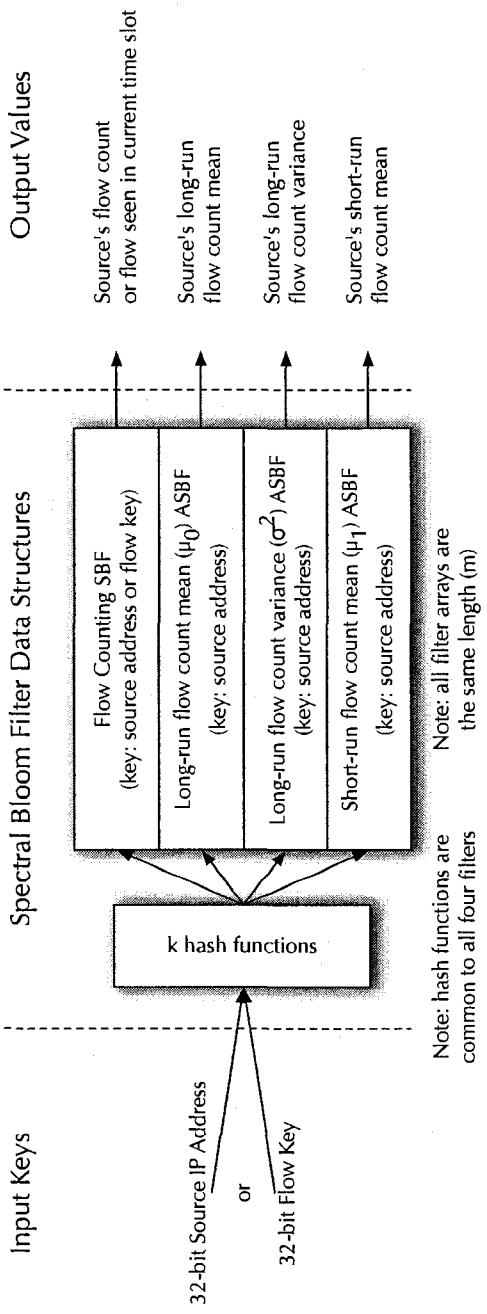


Figure 5.2: Data structures used by the Source Flow Counting algorithm.

the source's flow count must be increased by one. Within the Flow Counting SBF, a source is keyed by its 32-bit IPv4 address. The process for incrementing a source's flow count is shown in Figure 5.4. It is notable that the flow histories, keyed by the flow keys, and the flow counts, keyed by source IP addresses, are stored in the same SBF. The probability of a flow key being equal to any of the source IP addresses is fairly low, allowing for this reduction in memory demand.

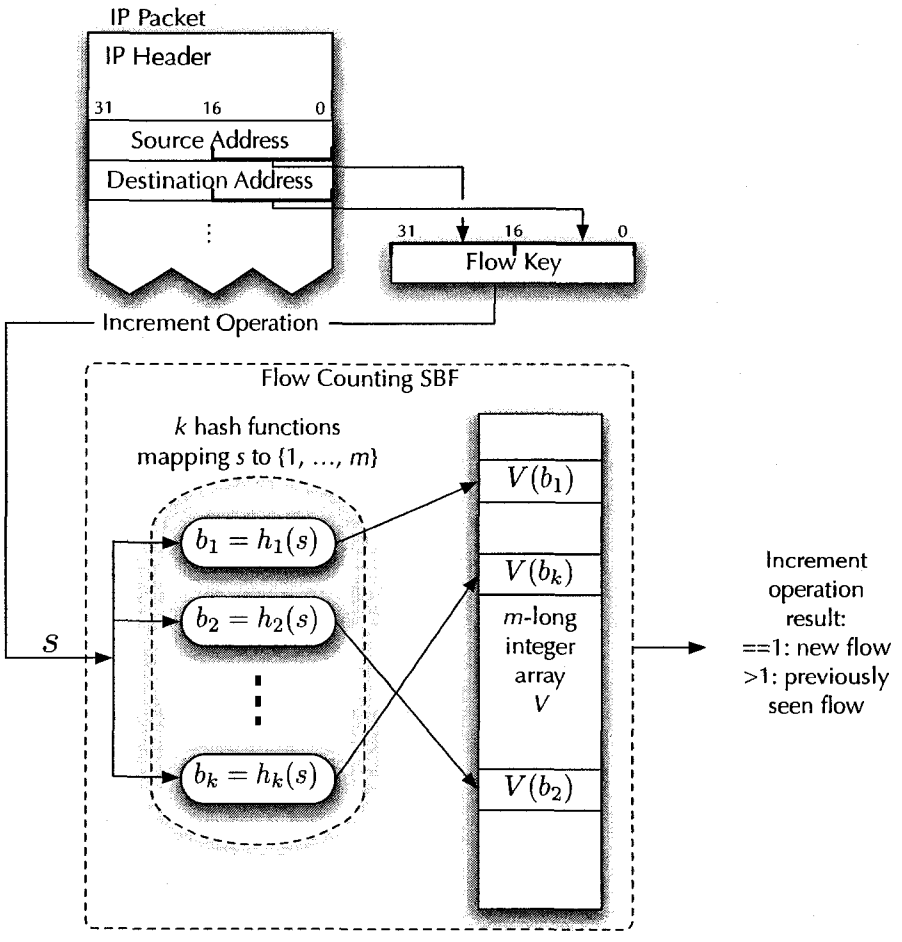


Figure 5.3: SFC process to determine if a flow is a new to the time slot.

The SBF is very well suited to storing flow counts and flow histories due its

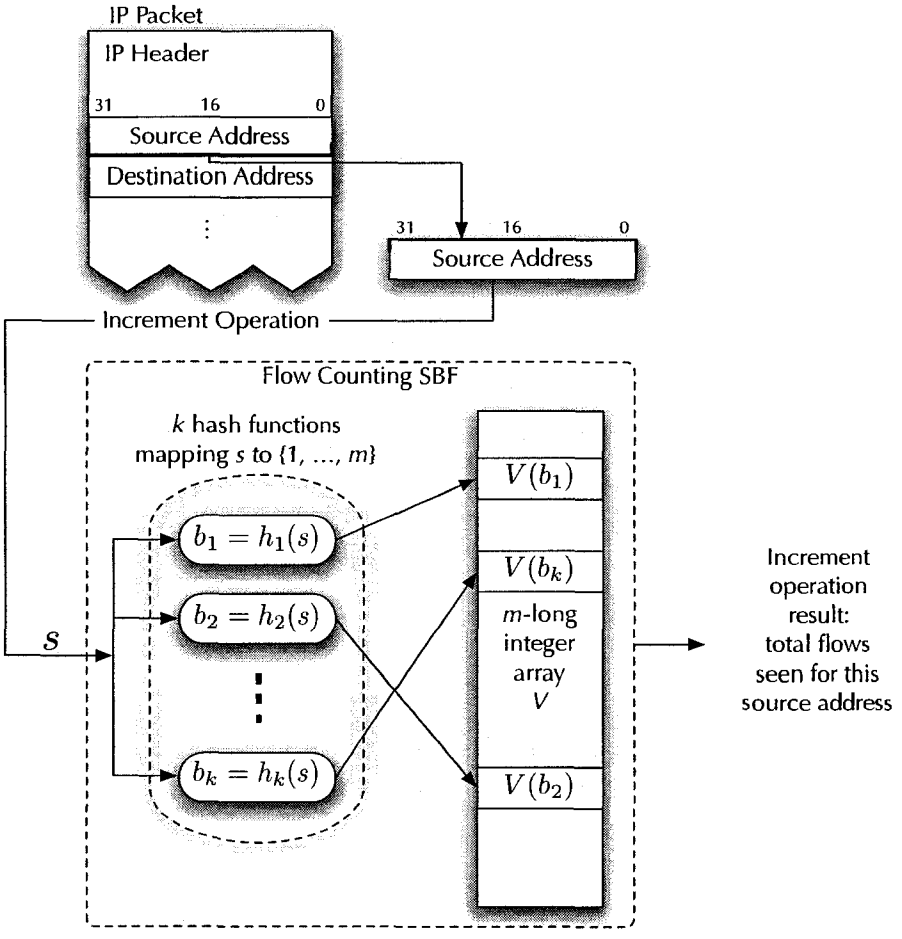


Figure 5.4: SFC process to increment a source's flow count.

compressive properties. One naive approach to storing flow counts and the flow list is to allocate an array of $2^{32}$ entries, one for each possible address in the Internet Protocol version 4 (IPv4) address space. A better approach might use an array of 65 536 pointers to another array of 65 536 pointers, representing the upper and lower 16 bits of a 32 bit value, with linked lists used to handle collisions in the space [36]. This is far more efficient than the naive approach but requires runtime memory allocation, resulting in a more costly insert operation. It also requires at least 131 072 pointers allocated before a single value is stored, each of which consumes additional memory. Because the distribution of addresses and flow IDs may span the entire range of 32-bit unsigned integers, there is no way to optimize this structure. When comparing to the SBF, equation (2.2) with $\gamma = 0.7$ can be used to determine the optimal filter size for the required inputs. A SBF designed to hold 1000 keys (the combined number of tracked source addresses and flow IDs) using five hash functions requires only a 7243 entry array. Storing 2000 keys increases this to 14285. In both cases the false positive rate determined by equation (2.1) is 3%. This can be effectively reduced by following the rule of thumb that the filter not be allowed to get more than half full. (A 131 072 entry array, using the same resources that the pointer-to-pointer method requires before holding a single item, is capable of storing over 18 000 items.)

At the end of each time slot, the SFC requires that the flow counters be reset to zero and the flow history cleared. For a SBF, this is a simple memory write operation to replace all the array entries with zeros.

### 5.1.3 Flow Count Mean and Variance Calculation and Storage

For each active source in the network under observation, SFC maintains two flow count means and one flow count variance. The long-run mean, designated $\mu_0$, is calculated using the source's flow counts at the end of a large number of previous time slots, say ten. SFC also calculates the variance of this mean, $\sigma^2$. The short-run mean, $\mu_1$, incorporates the flow counts of the current time slot and a small number of the previous ones, say three time slots total. The two means are calculated using the Exponentially Weighted Moving Average (EWMA) method; the formula is given in Equation (5.1) [42], where $g_k$ is the current mean, $g_{k-1}$ is the previous mean, $y_k$ is the current sampled value and $\alpha$ is the forgetting factor. The variance $\sigma^2$ is calculated using the similar Exponentially Weighted Moving Variance (EWMV) method, given in Equation (5.2) [42]. In this formula, $g_k$ is the current variance, $g_{k-1}$ the previous variance and $y_k$ and $\alpha$ are the same as for EWMA.

$$g_k = (1-\alpha)g_{k-1} + \alpha y_k, \text{ with } g_0 = 0 \tag{5.1}$$

$$g_k = (1-\alpha)g_{k-1} + \alpha(y_k - \mu)^2, \text{ with } g_0 = 0 \tag{5.2}$$

As was shown in Figure 5.2, one SBF stores the $\mu_0$ for all sources, another contains all the $\sigma^2$ and the third holds $\mu_1$. However, these three data structures are not strictly SBFs. A SBF is limited to two operations: query and increment. When counting

the flow's from a source, the increment operation is exactly what is needed. Every time a new flow is seen, execute the increment operation on the source address. In contrast, storing the EWMA and EWMV values associated with a source requires a data structure capable of storing arbitrary values. To expand on this concept, the flow count for a source always increases by one for each new flow seen during the time slot. When the SBF query function returns the minimum value contained by the $k$ counters associated with a given key, that value is guaranteed not to be less than the actual number of inserts on that key. The $\mu_0$, $\sigma^2$ and $\mu_1$ values tracked for each source do not have this property and therefore cannot be stored directly into a SBF. The data structure that holds these statistics must be capable storing many key-value pairs. When first calculated, the value for a key will likely not be one. Subsequent calculations for that may be greater than, equal to, or less than the current value. This is the intended meaning of arbitrary value.

The SBF remained attractive as a template because of the data compression properties. False positive errors are still acceptable to SFC when working with the mean and variance values. The solution used here is to extend the SBF. The Arbitrary Spectral Bloom Filter (ASBF), invented for SFC, is identical to the SBF with two differences. The increment operation is replaced with an insert operation that overwrites all $k$ counters indexed by the current key with the value to insert. The query function is modified by replacing the minimum function with the mode function, returning the most common value held by the $k$ counter associated with a key. The mode of the counters is most likely to be the correct value for the key. For example, in a ASBF with $k = 5$, say that the value three was inserted for key $s_1$. Immediately after the insert operation completes, the $k$ counters will be $\{3, 3, 3, 3, 3\}$. Sometime later, after a number of additional insertions, the ASBF is queried for $s_1$. At this point, the

counters may contain $\{3, 7, 3, 3, 2\}$. The query operation returns the mode of this set, three. Like the SBF, the ASBF will occasionally return an incorrect value. (Unlike the SBF, there is no guarantee relating an incorrect value to the correct one. In the SBF, an incorrect value is always greater than the correct one.) In the SFC worm detection application, these wrong values are acceptable given the other benefits of the SBF.

The ASBF does not easily lend itself to analysis of error rates because the insert mechanism overwrites all the counters for a key with the inserted value. The Bloom Filter (BF)'s $E_B$ comes from a straightforward analysis of the probability of a bit remaining at zero after $n$ random key insertions. The SBF's $E_{SBF}$ equation relies on the fact that each counter is incremented on an insert and is never decremented or overwritten. In the ASBF, these guarantees are not present and it is practically impossible to determine an equivalent to $E_{SBF}$. The effectiveness of the ASBF is therefore linked with the performance of SFC as a whole and will be empirically analysed as such.

The Sliding Window ASBF is a further extension of the ASBF. For SFC to operate continuously, it is essential that the filters be purged of outdated information. This is achieved with the Flow Counting SBF by clearing it on every time slot change. The ASBFs that store the flow statistics require a more complex approach. The Sliding Window ASBF consists of two $m$-long counter arrays. Each is called a pane. In any given time slot, one pane is the active pane and the other is the inactive pane. At the end of the time slot, the active pane becomes the inactive pane and vice versa. The newly active pane is cleared. Therefore, at the start of each time slot, the newly inactive pane holds the historical information carried from the previous time slot. The design of the Sliding Window ASBF is such that information is propagated from

the inactive pane to the active pane whenever a new flow is observed. Figure 5.5 shows the process for calculating an EWMA using the Sliding Window ASBF. In the overall SFC flow, the flow count statistics for a source are recalculated whenever a new flow is observed from that source. For $\mu_0$ and $\mu_1$, the input shown in step 3 of Figure 5.5 is the current flow count for that source, which is read from the Flow Counting SBF. At the end of a time slot, the active pane contains updated statistics for each source that was active during that time slot. Statistics for inactive sources, ones for which statistics are stored in the inactive pane but were not active during the time slot just ended, are not propagated from the inactive pane and are therefore purged when the pane change occurs. The process for calculating the EWMV $\sigma^2$ is very similar to that of the EWMA, except that the input in step 3 is the $\mu_0$ from the $\mu_0$ Sliding Window ASBF's current pane.

The flow count statistics are recalculated for every new flow because of the inherent limitation of the SBF: it cannot be asked for a list of what it contains, only if it contains a certain key. Therefore, it is impossible to do a batch update of the active source's statistics at the end of each time slot. One upside to the approach taken is that the flow insertion time is deterministic, whereas a batch process at the end of the time slot would vary greatly depending on the number of active flows in that time slot.

### 5.1.4 Abrupt Change Detection Method

The Shewhart control chart has been widely used for process monitoring since 1931. A process is deemed to be out of control when its sampled mean is more than some number of standard deviations away from the process mean. Equation (5.3) gives the Shewhart alarm criteria, where $\overline{y}(K)$ is the current sampled mean, $\mu_0$ is the expected

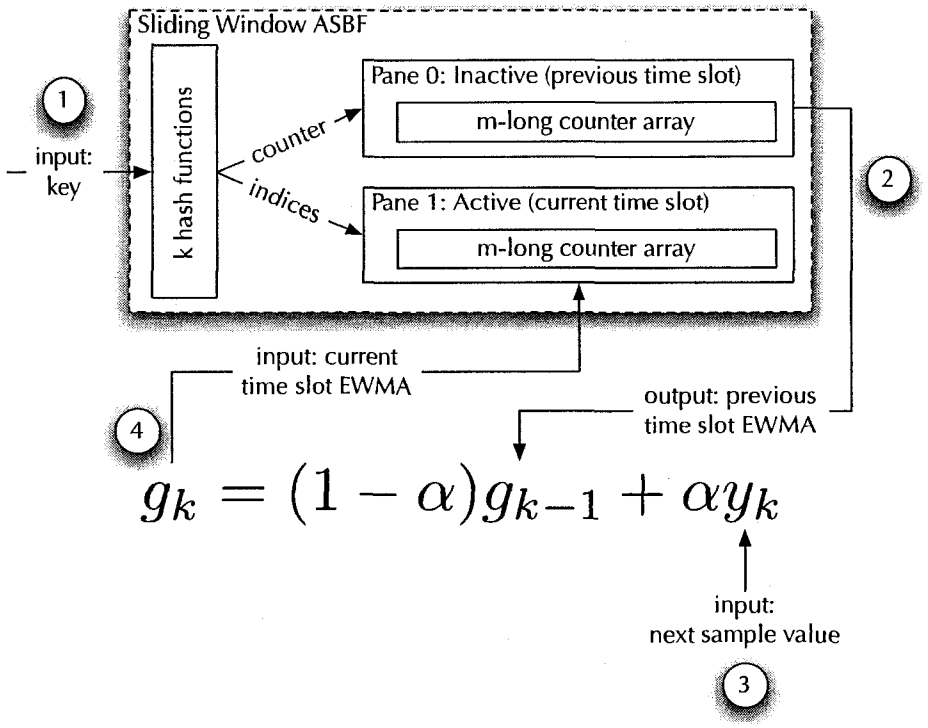


Figure 5.5: EWMA calculation using the Sliding Window ASBF. First, the key is processed through the hash functions to generate $k$ counter indices. Second, the ASBF query operation is performed on the inactive pane to fetch the EWMA calculated during the previous time slot. In step 3, that value is updated with the current sample value to generate the EWMA for the current time slot. The process is completed in step 4 when the new EWMA value is inserted into the active pane. When the time slot ends, the active and inactive panes are swapped and the newly active pane is cleared.

mean, and $\kappa$ is the number of standard deviations. Equation (5.4) is the slightly modified variant that looks only for increases in the mean. This is the form used by SFC.

$$|\overline{y}(K) - \mu_0| > \kappa \frac{\sigma}{\sqrt{N}} \tag{5.3}$$

$$\overline{y}(K) > \mu_0 + \kappa \frac{\sigma}{\sqrt{N}} \tag{5.4}$$

In general use, the process statistics are known a priori. However, this approach cannot be used for SFC because modelling a single host on a network, which could be running any number of different services, is extremely difficult. Instead, SFC uses the measured long-run mean, $\mu_0$, and its variance, $\sigma^2$, as the process statistics.

For each source, SFC calculates an Upper Control Limit (UCL), which is the value $\mu_0 + \kappa\sigma^2$. The $\mu_0$ and $\sigma^2$ values are drawn from the inactive ASBF. A source is declared to be infected by a worm when its $\mu_1$ value from the active ASBF is greater than the UCL. The short-run mean, $\mu_1$, is used to smooth out potential spikes in the source's traffic during a single time slot. To be effective, an active scanning worm must generate a continuously high volume of outbound traffic looking for hosts to infect. Other legitimate traffic generators on the source may produce short bursts of high-volume traffic before falling back to previous levels. The smoothing from $\mu_1$ prevents these sources from being inaccurately marked as infected.

A broad estimate of SFC's error probability, the probability that a given $\mu_1$ value is more than $\kappa$ deviations above $\mu_0$, can be determined using the Chebyshev inequality.

Assuming that $E(\mu_1) = \mu_1$ and that the variance of $\mu_1 = \sigma^2$, then the Chebyshev bound is given by Equation (5.5). This can be further refined by considering the one-sided variant of the Chebyshev inequality, called Cantelli's inequality. This result is shown in Equation (5.6). Using a standard value of $\kappa = 3$, the error bound is $\frac{1}{9}$ for the two-sided Chebyshev and $\frac{1}{10}$ for the one-sided version.

$$\begin{aligned} P\{|x-\nu| \geq \epsilon\} &\leq \frac{\sigma^2}{\epsilon^2} \\ P\{|\mu_1-\mu_0| \geq \kappa\sigma\} &\leq \frac{\sigma^2}{(\kappa\sigma)^2} \\ P\{|\mu_1-\mu_0| \geq \kappa\sigma\} &\leq \frac{1}{\kappa^2} \end{aligned} \tag{5.5}$$

$$P\{(x-\nu) \geq \kappa\sigma\} \leq \frac{1}{1+\kappa^2} \tag{5.6}$$

### 5.1.5 SFC Packet Processing

To summarize the SFC algorithm, this section describes the processing done by SFC for each packet arrival. This processing is easily divided into two parts. The steps shown in Figure 5.6 are performed on every observed packet. The second portion of SFC, illustrated in Figure 5.7 is done only for packets belonging to new flows.

On every packet arrival, SFC compares the current time against the end time of the current time slot. Each time slot is of equal length and the end time of a time slot is determined by adding the time slot length to the end time of previous time slot. If the current time slot has expired, SFC changes time slots by blanking the flow

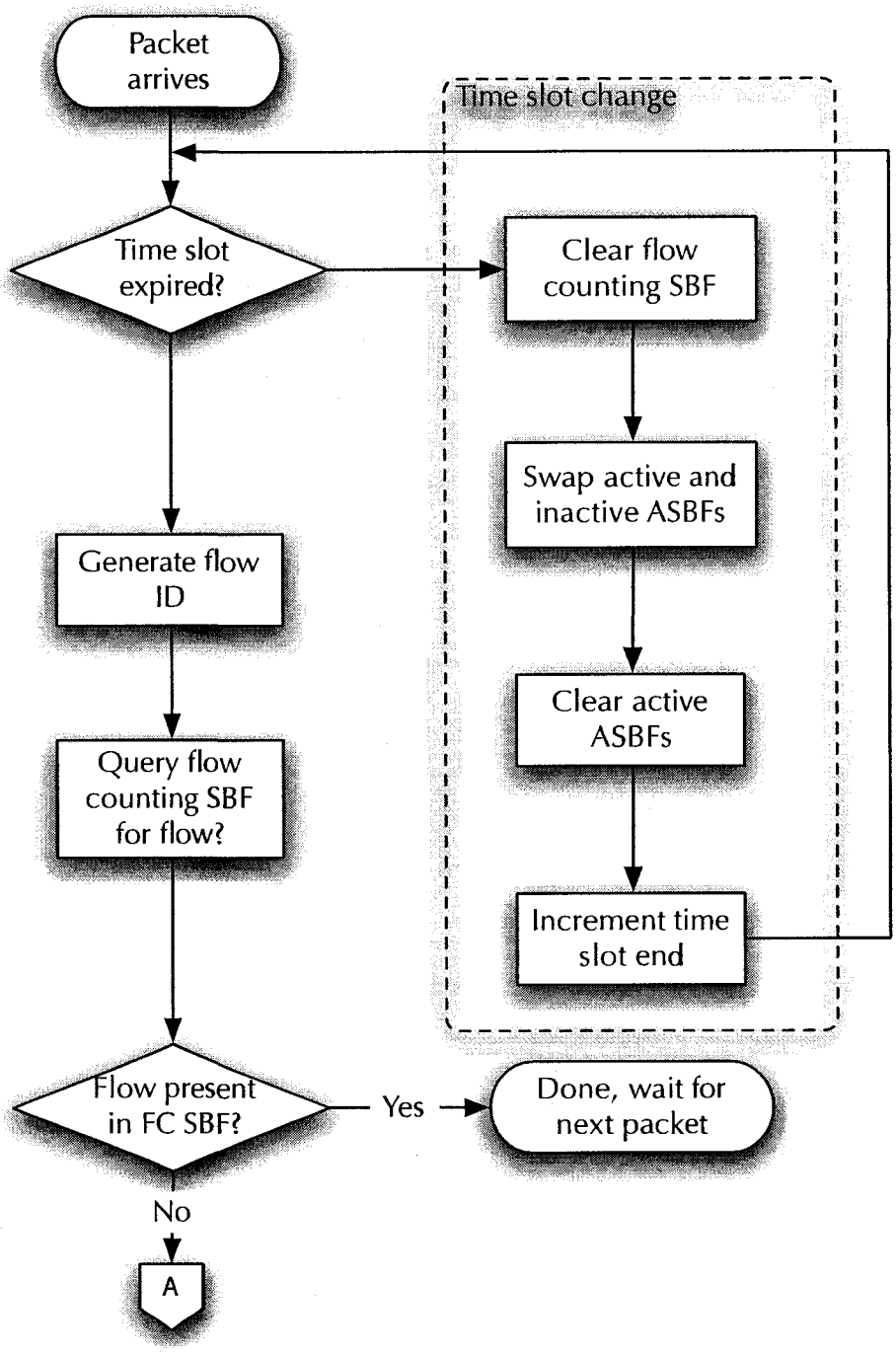


Figure 5.6: Source Flow Counting Flow Chart Part I

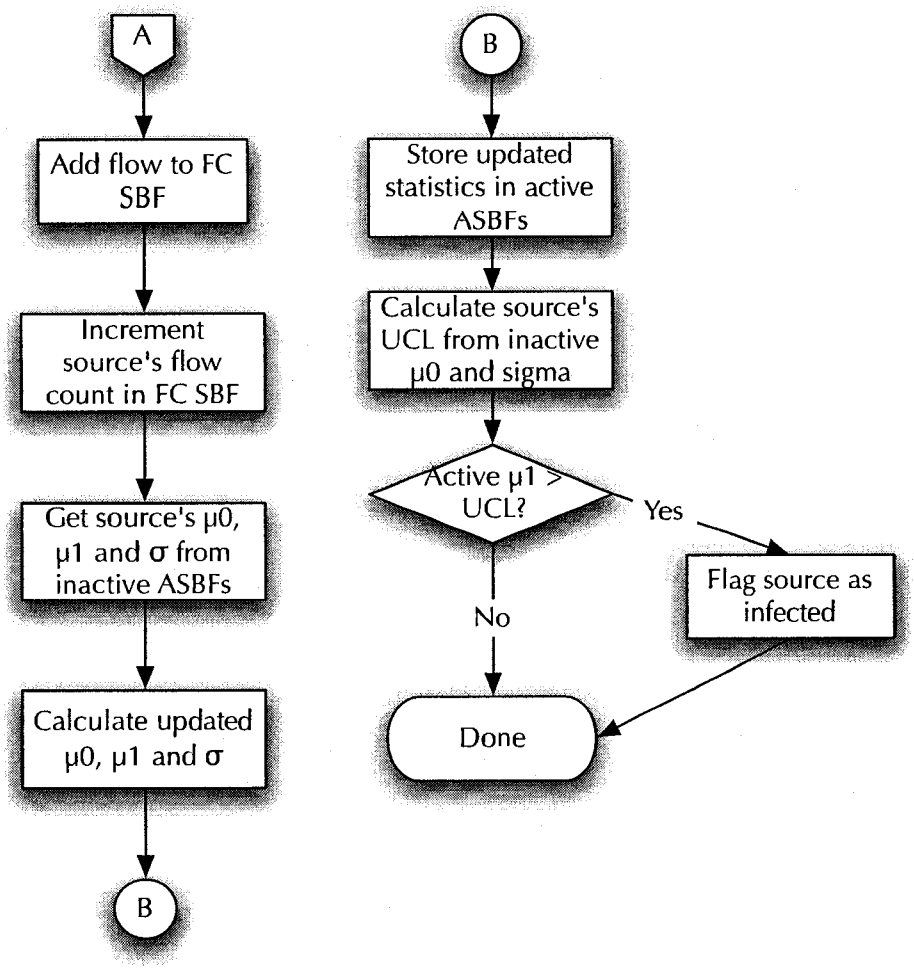


Figure 5.7: Source Flow Counting Flow Chart Part II

counting SBF, swapping the inactive and active ASBFs and clearing the new inactive ASBFs.

Once any required time slot changes are complete, SFC generates a flow ID to identify the flow. In the current implementation, the flow ID is a simple concatenation of the lower 16 bits of the source and destination IPv4 addresses into a single 32-bit key. This method was chosen to capture the most frequently changing bits of the address and for greater flow tracking accuracy in the local sub-net. Other options include hashing the source and destination addresses into a single 32-bit value to better distribute all possible flows through the 32-bit space. Using this key, the flow counting SBF is queried for the flow. If the flow is already present in the flow counting SFC, then no further processing is required and SBF becomes idle until the next packet arrives.

When a flow is not present in the flow counting SBF, it means that this is the first time a packet belonging to that flow has been seen in the current time slot. The flow is processed and added to the flow counting SBF by inserting the flow key and incrementing the source's flow count. (In the flow counting SBF, each source's key is its 32-bit IPv4 address.) A side effect of the SBF insert operation is that it returns the new value assigned to the key, in this case the source's flow count. This value is needed when calculating the active $\mu_0$ and $\mu_1$ values. These calculations also require the statistics stored in the inactive ASBFs, so the next step is to fetch them using the source's key.

With the inactive $\mu_0$, $\mu_1$ and $\sigma^2$ values and the current flow count, the flow's active statistics can be calculated using the EWMA and EWMV methods. These are stored in the active ASBFs. These statistics are updated on every new flow arrival. When the time slot ends and the active and inactive ASBFs are switched, the now inactive

ASBF contains the statistic values calculated for the last new flow seen from each source.

Worm detection on a source is performed using the Shewhart method. The inputs are $\mu_0$ and $\sigma^2$ from the inactive ASBF and $\mu_1$ from the active ASBF. The most recent $\mu_1$ is used to ensure the fastest possible reaction time to an infection. Otherwise, an infection would not be detected until the next time slot begins, giving the worm additional time to propagate. $\mu_0$ and $\sigma^2$ are taken from the inactive ASBFs because these statistics should be invariant with respect to the current sampled value.

Once a source's flow count reaches a pre-defined limit, the algorithm does not record any additional flows for that source in the time slot. The limit is chosen to be large enough to handle the vast majority of normal sources and is intended to place an upper bound on the number of unique keys inserted into the SFC SBFs by a single infected source. For example, a UDP worm is capable of generating up to 4000 packets per second. A single infected host would then be contacting upwards of 120 000 destinations per 30 s time slot, quickly overwhelming the flow counting SBF. The limit used for all experiments was 500 flows per source per time slot. When first implemented, the check for the limit was done by querying the flow count SBF for each packet arrival. However, the overheard of this operation quadrupled the simulation run time and was obviously unacceptable. The current implementation stores the set of source address that have reached the maximum flow limit for the current time slot in a C++ Standard Template Library Set. It could also be implemented using a small Bloom Filter.

The response of the system running SFC to a suspected infection depends on the system and is outside the scope of this work. Possible reactions span from completely blocking that source's traffic to notifying a parallel payload signature analysis to focus

on the infected source. This latter option would address the occasions when a healthy source's traffic output changes suddenly in a manner similar to an active scanning worm. SFC would quickly detect the source as infected. In response, the system could slow the source's traffic rate while examining the packet payloads for deeper signs of worm infection.

## 5.2 Comparison Against Other Methods

Chapter 3.2.2 describes several methods that identify worms through anomaly detection. On the surface, the most similar of these to SFC is Destination-Source Correlation (DSC). However, there are some key differences. Before comparing a source's current locality against a historical average, DSC looks for a precursor indicating a potential worm infection. When a source sends a packet to a port that it recently received a packet on, DSC considers the source to be infected. It then compares the source's scan rate on the suspected port against an expected scan rate for that port. The expected rate was set during a training period. SFC differs in that it does not monitor for the precursor, does not consider port when monitoring traffic and, most significantly, SFC maintains normal statistics for each source rather than globally. DSC also uses BFs in its implementation but their use is very different from SFC.

The same chapter presented a number of methods that look for abrupt changes in a source's network statistics. Two uses Simple Network Monitoring Protocol (SNMP) Management Information Base (MIB) variables queried from the hosts. As there is no MIB variable that tracks the number of unique destinations, these approaches do not use the same decision method as SFC. A third method did look for abrupt changes

in flow counts read from Cisco NetFlow records. However, this method was presented as an offline solution. The remaining algorithms described differ quite dramatically from SFC.

## 5.3 Simulation Setup

Verification of SFC was performed in simulation. The simulated network consisted of healthy clients, infected clients and servers. All traffic was directed through a single router, which ran SFC as part of its packet processing routine. In general, clients generated Transmission Control Protocol (TCP) traffic that simulated a mix of common internet protocols directed at the servers, which responded accordingly. Infected clients generated worm traffic modelled after Code Red II or SQL Slammer, as well as the same types of background traffic generated by healthy clients. When the router running SFC deemed a host to be infected it took no action beyond logging the detect to a file.

A sample instantiation of the SFC simulation is shown in Figure 5.8. In this figure, the simulation is configured with five clients that communicate with the four servers along the bottom of the figure through the central router. A single infected node, worm[0], is shown on the left. The three icons across the top of the figure represent administrative blocks. At the start of the simulation, the clients and worms send normal uninfected background traffic to the servers. The servers respond to their requests. Clients and worms do not communicate with each other and the servers do not initiate communication with any node. At some random time after the simulation has run long enough that the source statistics maintained by SFC have been fully calculated ($\lceil 1/\alpha_0 \rceil$ time slots), the worms on the worm nodes become

active and begin scanning. Given the relatively few nodes belonging to the network, virtually all scan request is responded to by the router with an Internet Control Message Protocol (ICMP) destination unavailable message. As SFC tracks each node individually, there is little to be gained in terms of exercising the algorithm to model actual worm propagation. Therefore, in the low probability event that an infected node does scan a client or server node, it is assumed that these nodes are immune to the worm and do not become infected.

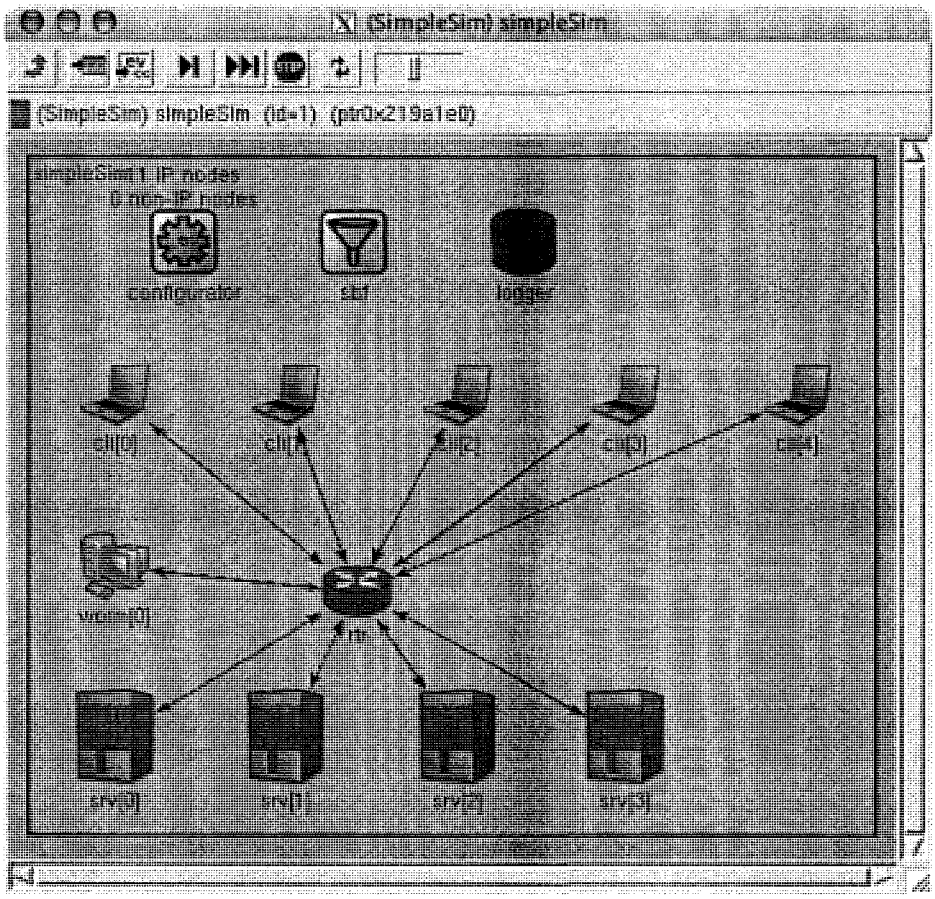


Figure 5.8: Simplified simulation model. The experiments were performed with additional client, worm and server machines as given in Table 5.1.

The SFC simulation has many obvious limitations. Ideally, a simulation to test an Intrusion Detection System (IDS) should meet the following fidelity requirements [43]:

1. Internet-level fidelity: as the spread of a worm through the Internet depends on the network topology and changes to routing as routers fail under the increased load, the simulation should model both the topology of the network and the behaviour of the routers driving it.

2. Packet-level fidelity: an IDS simulation should capture the diversity of network hosts and the networks that make up the Internet. Worms will spread at different rates depending the properties of the infected hosts and the networks they belong to.

3. Background-traffic fidelity: there is a strong need to simulate realistic background traffic to both measure the impact of the worm's spread on such traffic and to evaluate the benefits of any proposed IDS.

Meeting this list is no small feat. For example, the Georgia Tech Network Simulator [44] fulfills most of these requirements but runs on a 128 core supercomputer. The authors of the list developed their own worm simulator, PAWS, that requires the services of Emulab, a 212 node computer centre. With resources of this scale clearly unavailable, this project's simulation was designed to verify SFC within the constraints of what was available.

As SFC works only with packet headers, the contents of transmitted packets are ignored. Client and worm nodes generate only Hypertext Transport Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), Secure Shell (SSH), Post Office Protocol

(POP) and File Transfer Protocol (FTP) traffic. The HTTP traffic generation engine uses the empirical model from [45]. The other protocols are generated using statistical models determined by [46]. The generators calculate the outgoing packet size, the response packet size and the inter-request period. The response packet size is included in the packet sent to the server nodes, which reply by simply sending a packet back to the source node with the length asked for by the source. The source node then sleeps that protocol until the time to send the next packet.

When examining the simulation results, the relative simplicity of the simulation must be taken into account. Primarily, the network under study in the simulation is very small compared to the global scope of worm traffic and it has both a low volume of background traffic and a limited distribution of that traffic. It is likely that some types of Internet traffic possess the worm-like properties that SFC is designed to detect and these sources would be deemed infected in SFC was deployed into a real-world scenario. The simulation is sufficient to exercise the concepts behind SFC. However, before any wide-scale usage more exhaustive testing on captured data or on a larger simulated environment with greater fidelity is required.

The simulation is implemented using the OMNeT++ [47] open-source network simulator and its INET [48] Internet modelling package. This simulation engine was selected based as much on price as the fact that it runs on the author's PowerBook G4. The specific versions used were OMNeT++ 3.2p1 and INET-20060330.

The primary tuneable parameters to the simulation are the Shewhart deviation multiplier $\kappa$, the SBF parameters $\gamma$ and $n$, the base start time and random deviation. The length, $m$, of the counter array $V$ is determined from $n$, $k$ and $\gamma$. Recall that $\gamma = \frac{nk}{m}$; therefore, $m = \frac{nk}{\gamma}$.

## 5.4 Experimental Results

For all experiments in this section, the simulated network was configured to have 20 healthy clients, five infected clients and four servers. Immediately after the simulation is begun, all clients begin to send healthy (HTTP, SMTP, SSH, POP and FTP) to the servers. The infected clients, designated the worms, generated no worm traffic for the first 310 s of simulated time. At 310 s, each worm waited an additional delay period before beginning to send worm traffic. This delay period was drawn from an exponential random variable with an expected value of 20 s. The simulation time limit was 1000 s. Table 5.1 lists the values assigned to all static simulation parameters:

Table 5.1: Static Simulation Parameters

| Name | Value | Description |
|---|---|---|
| sim_time_limit | 1000 s | Simulation time limit |
| num_cli | 20 | Number of healthy clients |
| num_srv | 4 | Number of servers |
| num_worm | 5 | Number of infected clients |
| worm_start_time | 310 s | Minimum time from start of simulation for worm launch |
| worm_delay_time | exponential(20 s) | Additional worm start delay from minimum time |
| $\gamma$ | 0.7 | Expected BF utilization |
| $\kappa$ | 3 | Shewhart UCL multiplier |
| $\alpha_{\mu_0}$ | 10 | Long-run average spans $\lceil 100/\alpha_{\mu_0} \rceil$ timeslots |
| $\alpha_{\mu_1}$ | 33 | Short-run average spans $\lceil 100/\alpha_{\mu_1} \rceil$ timeslots |
| start_time | 3 s | Delay from beginning of simulation before SFC begins |
| time_slot_length | 30 s | Time slot length |

Experiments were run a sufficient number of iterations to generate results within a 95% confidence interval. The statistics under observation were:

- $level_b$: the flow tracking SBF load level from the start of the simulation until the first worm became active;
- $level_a$: the flow tracking SBF load level after the first worm became active until the end of the simulation;
- $fp$: the Bloom error rate from the flow tracking SBF;
- $tp$: the true positive ratio is the proportion of worms correctly detected as worms during the simulation;
- $tn$: the true negative ratio is the proportion of healthy clients and servers not detected as worms during the simulation;

Simulations were run using a scripted framework that ensured that the random seed value fed to OMNeT++ differed for all runs of a single set of parameter values. Post-processing of results was performed using MATLAB.

### 5.4.1 Code Red II

The Code Red II simulations are intended to exercise SFC under the conditions of a typical TCP-based active scanning worm. Each infected host issues probes at approximately the rate of the Code Red II worm, about 10 times per second. The probes are TCP SYN packets marking the beginning of a TCP exchange. The real Code Red II worm launches up to a thousand threads, each of which attempts to infect one host at a time. Each thread sends a TCP SYN packet to the target and waits for a response from the target or until the TCP timeout has expired. Because no host in the simulation is capable of responding to the worm's infection attempts, the simulated Code Red worm has been simplified to a single thread. The worm sends

attempts to open a TCP connection to the target address. If there is no response after 100 ms, the worm closes unilaterally closes the socket and moves onto the next target. As the probing rate of the real Code Red worm and the simulated one are similar, this is an acceptable simplification for the purposes of exercising SFC.

The plots shown in Figure 5.9, Figure 5.10 and Figure 5.11 are selected results from one run of the Code Red simulations chosen to illustrate the operation of SFC and the simulation engine. Figure 5.9 shows the monitored flow counts for a single client. In this figure, the data points are the contents of the relevant SBF or ASBF at the end of each time slot, when the filters are written to file for post-processing. The flow count dots show the number of flows generated by this client in each time slot. This is the number of destinations contacted by the client during the time slot. The $\mu_0$ and $\mu_1$ lines show the long-run and short-run moving averages of these flow counts. The standard deviation of the $\mu_0$ is displayed as $\sigma^2$. Finally, the Shewhart UCL for this client is plotted as UCL. Notice that the UCL decreases with the standard deviation. As this is an uninfected client, $\mu_1$ never exceeded the UCL and no worm activity was detected. This figure clearly shows the differences between $\mu_0$ and $\mu_1$ as they each track their respective means.

Figure 5.10 plots the SFC values for a SSH server within the simulation. Compared to the client, the server contacts a larger number of destinations and the flow counts, means and UCL reflect this. This server is also not infected during the simulation, so $\mu_1$ remained well below the UCL.

The behaviour of a worm machine is shown in Figure 5.11. This figure is noticeably different from Figure 5.9 and Figure 5.10. The worm is dormant for the first 310 s of the simulation. In this portion of the plot, the worm's SFC values are typical of a healthy client. At some point after 310 s, the worm becomes active and the flow count

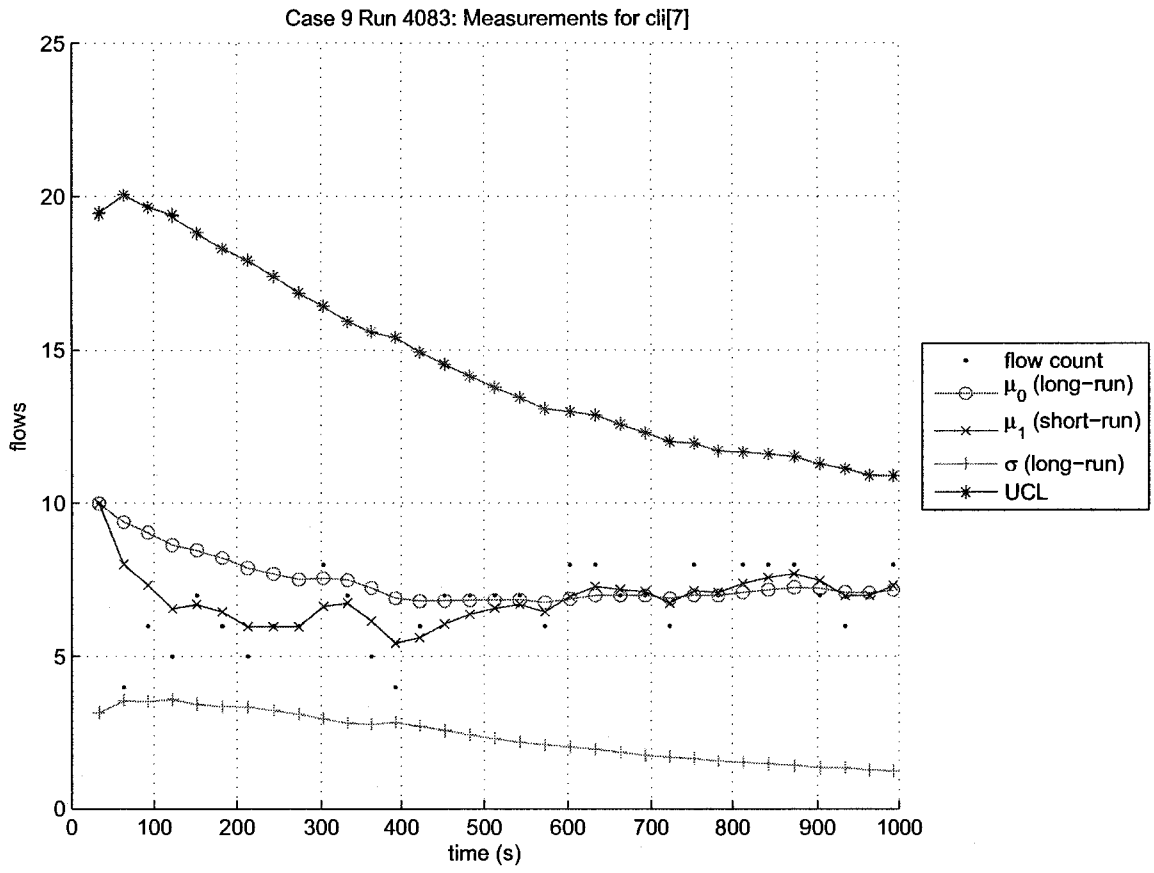


Figure 5.9: Flow counts for a client in the Code Red simulation showing the the long-run moving average $\mu_0$, the short-run moving average $\mu_1$, the variance $\sigma^2$ and the Shewhart Upper Control Limit.

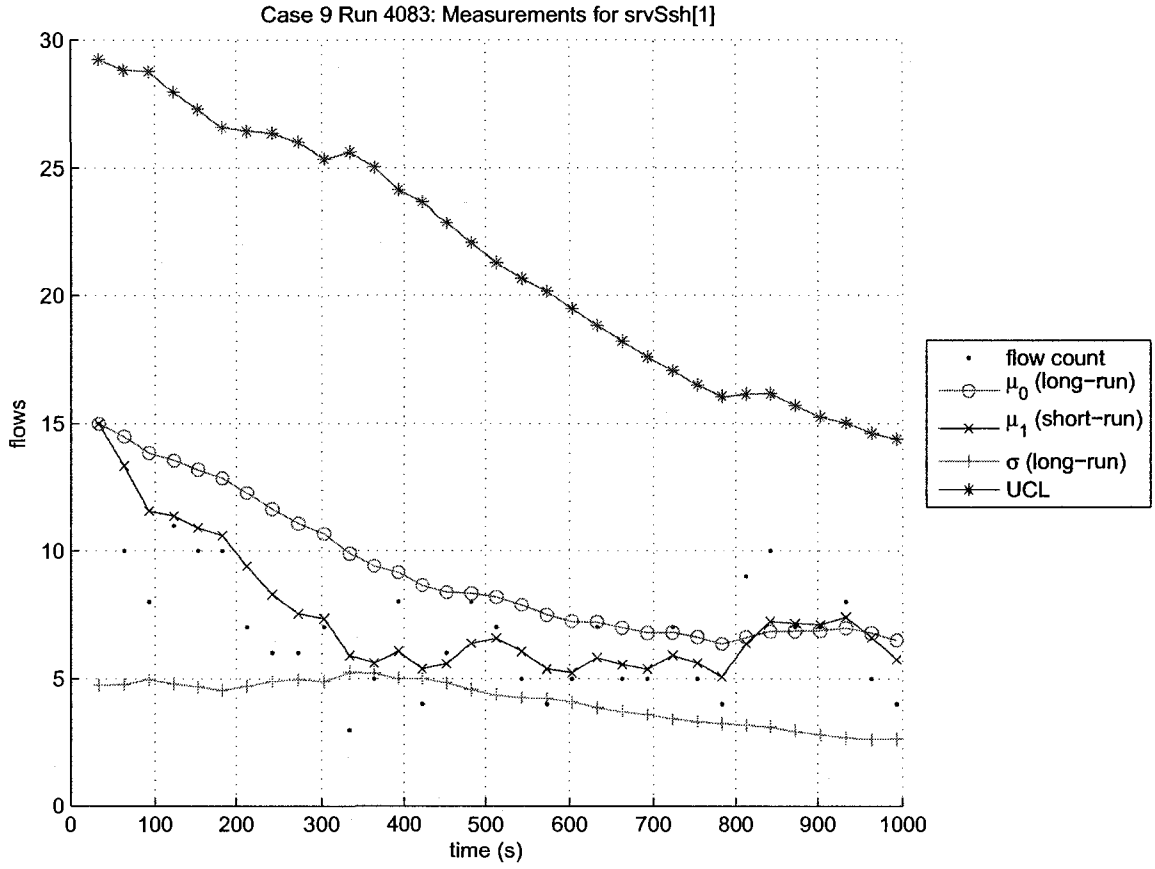


Figure 5.10: Flow counts for a SSH server in the Code Red simulation showing the the long-run moving average $\mu_0$, the short-run moving average $\mu_1$, the variance $\sigma^2$ and the Shewhart Upper Control Limit.

mean rises to approximately 270. The short-run mean, $\mu_1$ tracks this rise very quickly. The $\leftarrow$ **worm**[**2**] notations mark the time and flow count value when the worm was detected in each time slot. When reading this plot, it is important to remember that SFC uses the $\mu_0$ and $\sigma^2$ values from the previous time slot when calculating the UCL for the current time slot. A useful mental trick is to shift the UCL line one time slot to the right, after which the $\mu_1$ line will be above the UCL line in the time slots where worms are detected. Recall that the $\mu_1$ line plots the value of $\mu_1$ at the completion of the time slot, therefore the detection points will always be at or below the $\mu_1$ line.

Table 5.2, Table 5.3, Table 5.4 and Table 5.5 give the results of the Code Red v2 simulations. Every set of simulation parameters, referred to as a case, was simulated for a sufficient number of repetitions to generate 95% confidence intervals for each statistic. Each table contains the results for a different value of $n$, the expected number of items inserted into each SBF and ASBF in the system. From left to right, the columns show the impact of increasing $k$ from 3 to 9 in four steps on the statistics listed in the left-most column.

Some expected BF behaviours are clearly apparent in the four tables. The statistics $level_b$ and $level_a$ are constant as $k$ increases. This is an expected result because the size $m$ of the SBF increases with $k$. The fill levels decrease as $n$ increases, as the same number of items are being inserted into larger filters with greater capacity. The Bloom error rate, $f_p$, is inversely proportional to $n$ and $k$. This is in accordance with the Bloom error estimate given in equation (2.1).

The performance of SFC is assessed through the $t_p$ and $t_n$ statistics. $t_p$ is the true positive rate and measures SFC's accuracy in detecting all the infected hosts as infected at least once during the simulation. $t_n$ is the true negative rate and represents the proportion of healthy clients that SFC did not declare to be infected during the

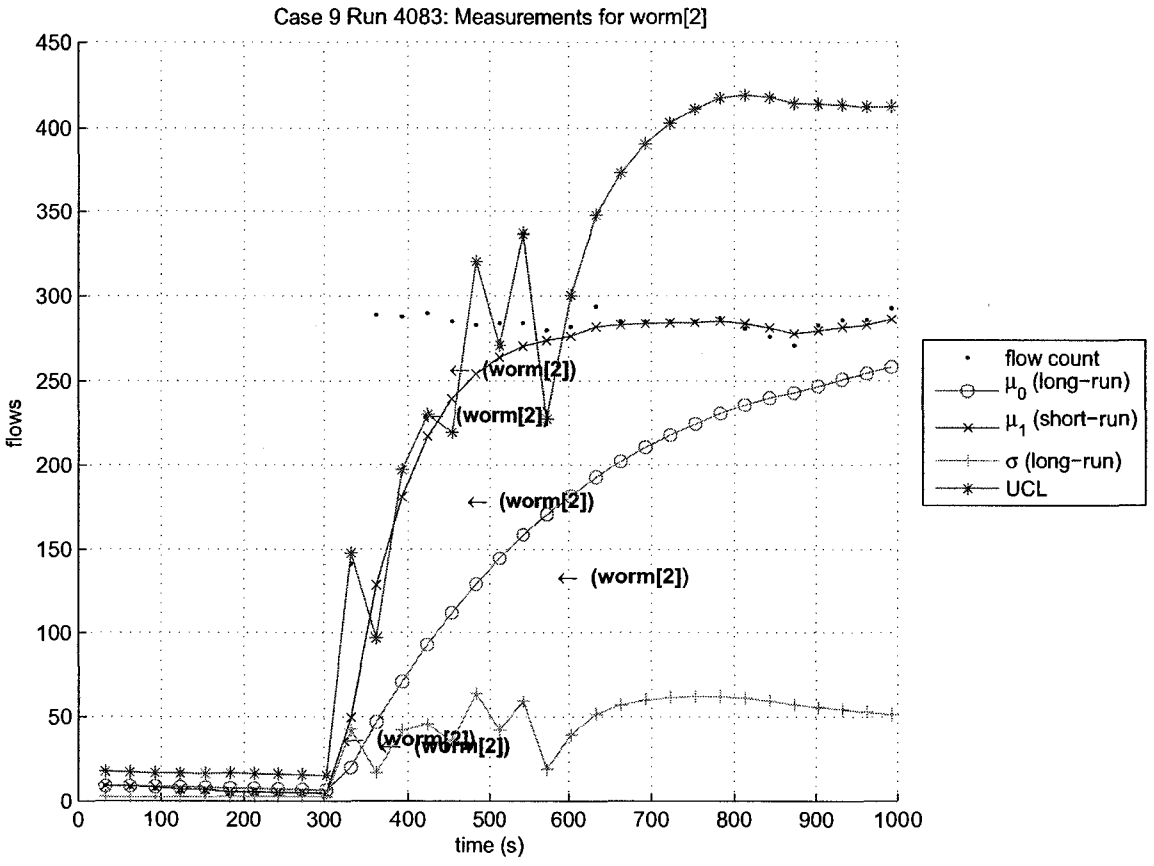


Figure 5.11: Flow counts for a Code Red worm showing the the long-run moving average $\mu_0$, the short-run moving average $\mu_1$, the variance $\sigma^2$ and the Shewhart Upper Control Limit. SFC detected the worm as infected at the points marked with $\leftarrow$ (**worm**[2]).

simulation. When SFC is operating correctly with what could be considered to be 100% accuracy, both $t_p$ and $t_n$ are 1. Any value less than one represents detection errors by SFC.

Table 5.2: SFC Code Red v2 Results for $n = 500$ determined to a 95% confidence interval

| k | 3 | 5 | 7 | 9 |
|---|---|---|---|---|
| $level_b$ | 0.4156 ± 0.0161 | 0.4138 ± 0.0143 | 0.4046 ± 0.0147 | 0.4033 ± 0.0098 |
| $level_a$ | 0.9220 ± 0.0033 | 0.9224 ± 0.0019 | 0.9219 ± 0.0024 | 0.9214 ± 0.0022 |
| $f_p$ | 0.4476 ± 0.0048 | 0.3305 ± 0.0057 | 0.2554 ± 0.0037 | 0.2037 ± 0.0044 |
| $t_p$ | 0.9111 ± 0.1054 | 0.9556 ± 0.0882 | 0.9556 ± 0.0882 | 0.9778 ± 0.0667 |
| $t_n$ | 0.8611 ± 0.0651 | 0.9611 ± 0.0546 | 0.9778 ± 0.0264 | 1.0000 |

Table 5.3: SFC Code Red v2 Results for $n = 1000$ determined to a 95% confidence interval

| k | 3 | 5 | 7 | 9 |
|---|---|---|---|---|
| $level_b$ | 0.2327 ± 0.0068 | 0.2331 ± 0.0082 | 0.2267 ± 0.0046 | 0.2291 ± 0.0051 |
| $level_a$ | 0.7224 ± 0.0035 | 0.7191 ± 0.0043 | 0.7191 ± 0.0036 | 0.7163 ± 0.0084 |
| $f_p$ | 0.2305 ± 0.0031 | 0.1370 ± 0.0021 | 0.0937 ± 0.0019 | 0.0704 ± 0.0021 |
| $t_p$ | 1.0000 | 1.0000 | 0.9556 ± 0.0882 | 1.0000 |
| $t_n$ | 0.9889 ± 0.0220 | 1.0000 | 0.9833 ± 0.0250 | 0.9944 ± 0.0167 |

For the network under observation, the results show that a filter configured for $n = 1000$ and $k = 3$ gives the best detection rates with the minimum of resources. For this configuration, each pane in the SBFs and ASBFs is about 4300 elements long, so the total memory requirements for all four filters with two panes each is around 36 000 elements.

Detection errors are the result of collisions in the SBF and/or the ASBFs. Figure 5.12 shows the combined effect of collisions in both types of filters. The client dis-

Table 5.4: SFC Code Red v2 Results for $n = 2000$ determined to a 95% confidence interval

| **k** | 3 | 5 | 7 | 9 |
|---|---|---|---|---|
| $level_b$ | 0.1226 ± 0.0042 | 0.1219 ± 0.0055 | 0.1190 ± 0.0032 | 0.1217 ± 0.0049 |
| $level_a$ | 0.4695 ± 0.0054 | 0.4698 ± 0.0042 | 0.4705 ± 0.0026 | 0.4711 ± 0.0030 |
| $f_p$ | 0.1081 ± 0.0017 | 0.0616 ± 0.0010 | 0.0489 ± 0.0018 | 0.0345 ± 0.0008 |
| $t_p$ | 1.0000 | 1.0000 | 1.0000 | 0.9778 ± 0.0667 |
| $t_n$ | 0.9833 ± 0.0250 | 1.0000 | 1.0000 | 1.0000 |

Table 5.5: SFC Code Red v2 Results for $n = 4000$ determined to a 95% confidence interval

| **k** | 3 | 5 | 7 | 9 |
|---|---|---|---|---|
| $level_b$ | 0.0618 ± 0.0035 | 0.0628 ± 0.0027 | 0.0628 ± 0.0013 | 0.0630 ± 0.0033 |
| $level_a$ | 0.2714 ± 0.0026 | 0.2715 ± 0.0043 | 0.2700 ± 0.0036 | 0.2727 ± 0.0025 |
| $f_p$ | 0.0526 ± 0.0013 | 0.0312 ± 0.0005 | 0.0219 ± 0.0006 | 0.0174 ± 0.0007 |
| $t_p$ | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| $t_n$ | 0.9900 ± 0.0224 | 1.0000 | 0.9900 ± 0.0224 | 0.9889 ± 0.0220 |

played in this figure normally generates less than ten flows per time slot. Due to a SBF collision with at least one more active source, this client is recorded as having over 900 flows just after time 500. Collisions with the same source in the ASBFs have inflated both tracking averages, the deviation and, as a result, the UCL. The final outcome of these cascading errors is a single false positive detection, marked by the arrow. During the remainder of the simulation the algorithm slowly recovers. The tracking averages and the UCL are declining. (The abrupt drop at the right edge of the plot is an artifact of the simulation's termination.) For this client, the ASBF query function is able to find a mode value for all queries.

Another cause of detection errors is when the ASBF query function is unable to find a mode value. The cause is collisions in the ASBF that result in none of the values stored in the $k$ entries for the key having a plurality. When this happens, the query function returns zero. This does not cause an immediate false positive or false negative because the algorithm does not declare a host to be infected when an ASBF query operation returns zero. However, a side-effect of this return value is that SFC "loses" the flow history for the affected host. The algorithm resets the tracking averages to the current flow count. When this flow count is anomalous for the host, oftentimes an incorrect detection later in the algorithm's operation is the outcome. Such an instance is shown in Figure 5.13. The three time slots where all lines on the plot fall to zero mark occurrences of mode failures. Almost 50 seconds after the last failure, SFC declares the host to be infected. (The vertical axis on this plot should be noted. This particular detect could easily be discarded by a simple filter that discards all detects where the flow count is less than some floor value.)

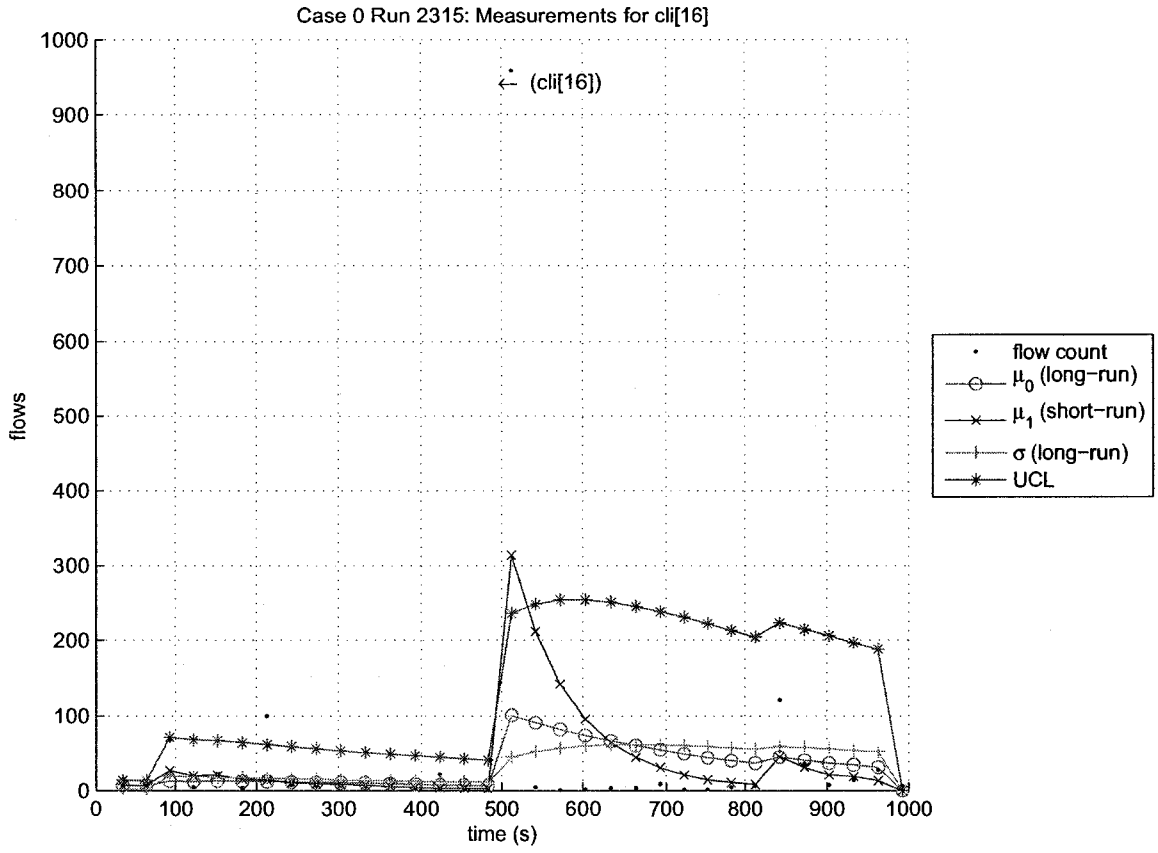


Figure 5.12: Collisions in both the SBF and the ASBF result in a false positive on this client.

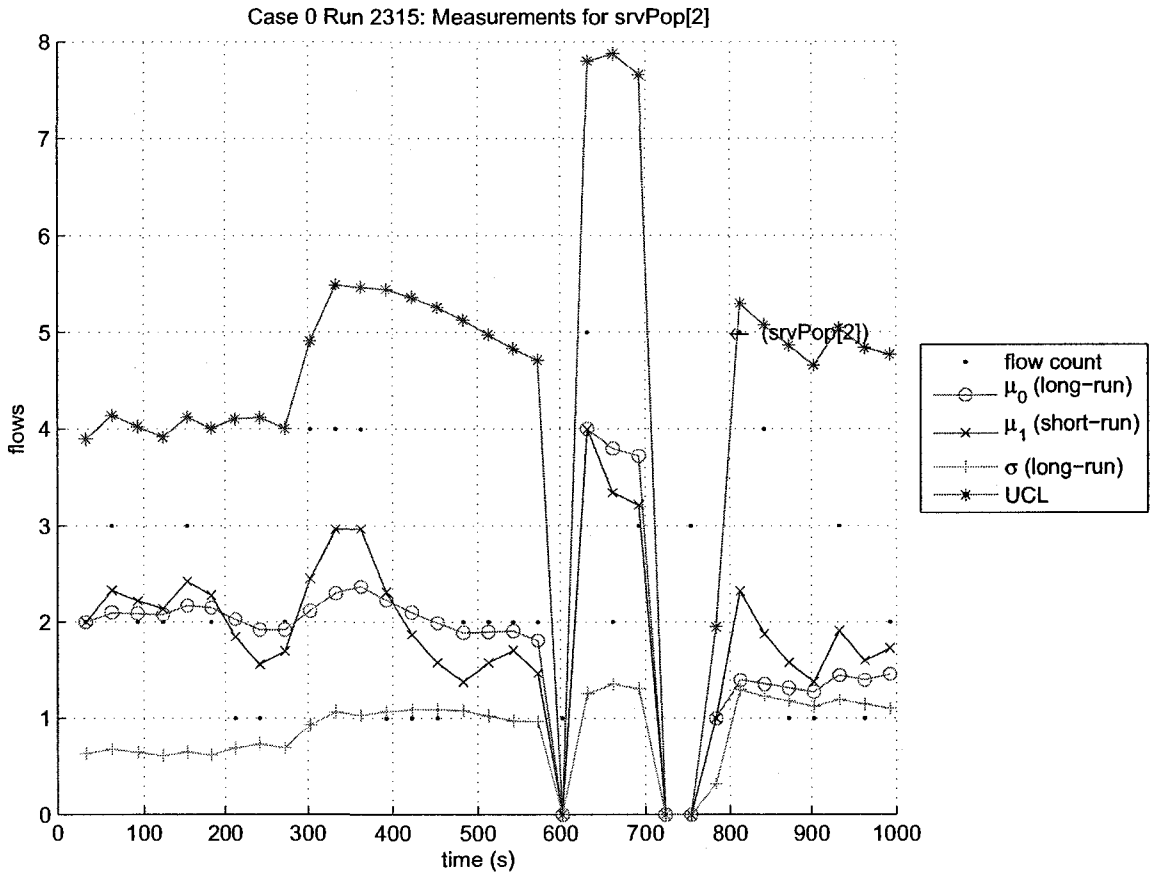


Figure 5.13: When the ASBF query operation is unable to find a mode, the result is often a false positive.

### 5.4.2 SQL Slammer

The SQL Slammer simulations exercise SFC under more stressful conditions than the Code Red II simulations. A host infected by SQL Slammer produces a much larger volume of outgoing packets to a wider range of destinations. In the simulation, each infected host generates up to 4000 probes per second. Each probe consists of a single 404 byte Unigram Data Protocol (UDP) packet, which is the same probe rate and packet size used by the SQL Slammer worm.

The stored flow count for each worm source is limited to 500 by the maximum flow limiter. When run without the limiter in place, once the worms become active they completely fill the flow counting SBF within the first couple of seconds of each time slot. The effect of this is that all queries into the flow counting SBF after this point return a non-zero value. The algorithm interprets these continued false positives as indicators that the flow has already been recorded. Figure 5.14 illustrates the effect of this on flow tracking. This figure is for a standard client, which generally contacts less than 20 destinations per time slot. The flow counts, means and variance are correct until the worms became active around 340 s. After this time, collisions in the SBF due to overflow have driven up the flow counts reported at the end of each time slot to vast over-estimations. Further, because the client often did not generate any traffic during the time slot before the SBF overflowed, the recorded means and variance are almost all zero after the worm begins. The situation for an infected source is shown in Figure 5.15. As calculated earlier, the worm should be contacting around 120 000 destinations per time slot, yet the recorded flow counts are approximately 1% of this. However, the abrupt upward change in the flow count after the worm begins is present and the worm is detected. In general, the simulations with no flow count limit were unable to detect some worms and also classified some healthy sources as infected.

The flow count limit was introduced as a result of these experiments.

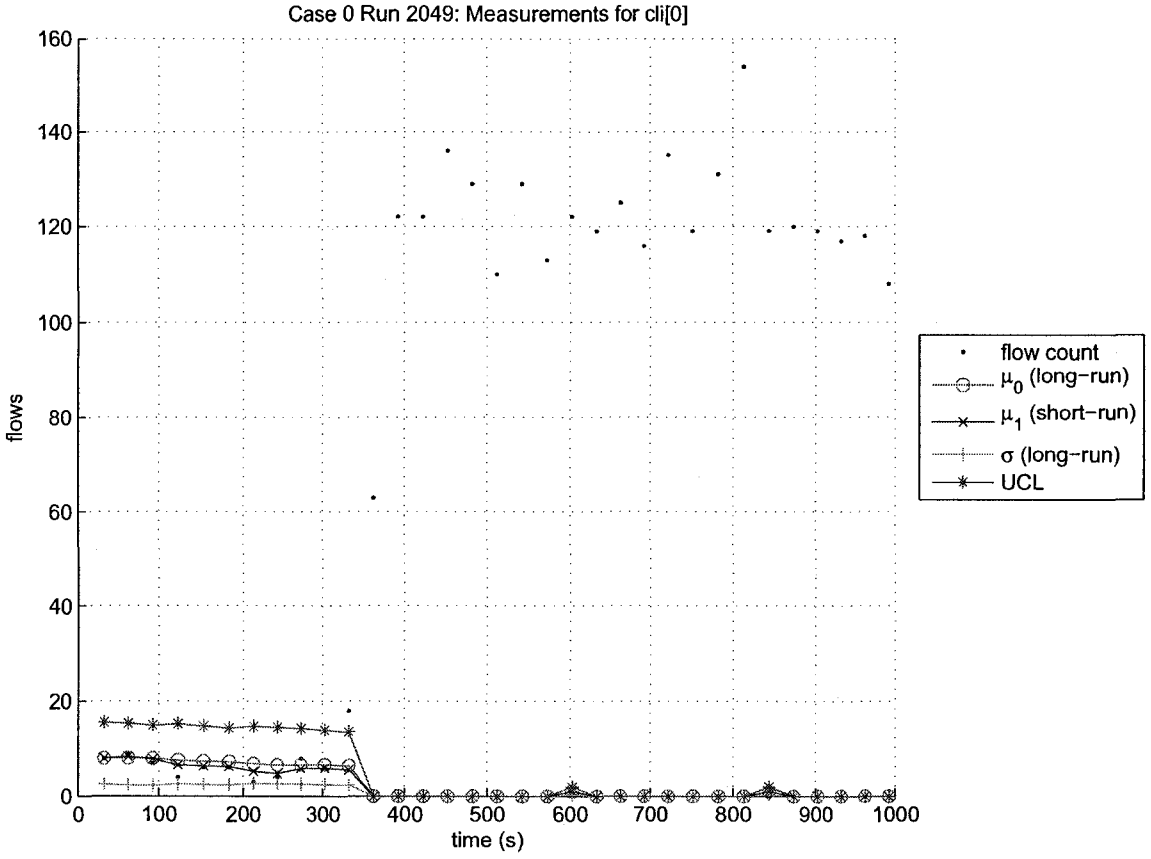


Figure 5.14: When the flow counting SBF overflows, the flow counts for healthy sources are too large yet the means and variance are not recorded.

Table 5.6, Table 5.7, Table 5.8 and Table 5.9 give the results of the SQL Slammer simulations. As was the case with the Code Red v2 simulations, every set of simulation parameters, referred to as a case, was simulated for a sufficient number of repetitions to generate 95% confidence intervals for each statistic. Each table contains the results for a different value of $n$, the expected number of items inserted into each SBF and ASBF in the system. From left to right, the columns show the

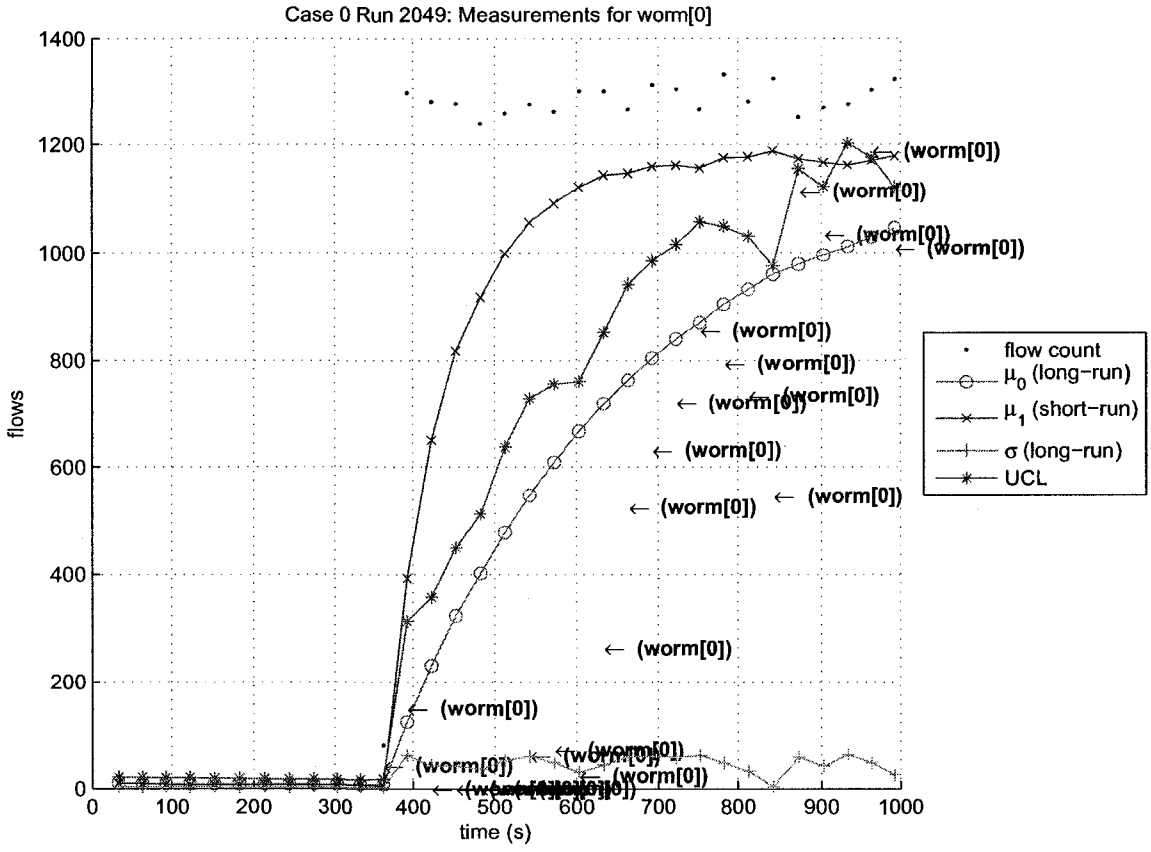


Figure 5.15: For an infected source, a flow counting SBF overflow means underestimated flow counts but the infection is still detected.

impact of increasing $k$ from 3 to 9 in four steps on the statistics listed in the left-most column. As each SQL Slammer source generates a much larger volume of packets than a Code Red v2 source, the SQL Slammer simulations take considerably longer to execute. To mitigate this impact, the number of worm sources was reduced to 2 and the number of clients reduced to 5. Further, while the Code Red v2 simulations used $n = \{500, 1000, 2000, 4000\}$, the SQL Slammer simulations were run with $n = \{1000, 2000, 4000, 8000\}$. The lowest $n$ value was increased because it was known in advance that the two worm sources alone would generate 500 flows each. The large $n$ value was added to explore the possibility that a very large filter improves accuracy.

Table 5.6: SFC SQL Slammer Results for $n = 1000$ determined to a 95% confidence interval

| **k** | 3 | 5 | 7 | 9 |
|---|---|---|---|---|
| $level_b$ | 0.0737 ± 0.0009 | 0.0737 ± 0.0143 | 0.0769 ± 0.0007 | 0.0739 ± 0.0024 |
| $level_a$ | 0.5803 ± 0.0019 | 0.5549 ± 0.0008 | 0.5485 ± 0.0017 | 0.5834 ± 0.0022 |
| $f_p$ | 0.1510 ± 0.0015 | 0.0819 ± 0.0007 | 0.0562 ± 0.0009 | 0.0430 ± 0.0006 |
| $t_p$ | 1.00000 | 0.9167 ± 0.0833 | 1.0000 | 1.0000 |
| $t_n$ | 0.9667 ± 0.0333 | 0.9667 ± 0.0833 | 1.0000 | 0.9667 ± 0.0333 |

Table 5.7: SFC SQL Slammer Results for $n = 2000$ determined to a 95% confidence interval

| **k** | 3 | 5 | 7 | 9 |
|---|---|---|---|---|
| $level_b$ | 0.0391 ± 0.0017 | 0.0393 ± 0.0009 | 0.0397 ± 0.0011 | 0.0404 ± 0.0010 |
| $level_a$ | 0.3311 ± 0.0019 | 0.3238 ± 0.0006 | 0.3179 ± 0.0019 | 0.3201 ± 0.0006 |
| $f_p$ | 0.0674 ± 0.0021 | 0.0385 ± 0.0002 | 0.0267 ± 0.0004 | 0.0214 ± 0.0004 |
| $t_p$ | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| $t_n$ | 0.9333 ± 0.0667 | 1.0000 | 0.9667 ± 0.0333 | 1.0000 |

Table 5.8: SFC SQL Slammer Results for $n = 4000$ determined to a 95% confidence interval

| k | 3 | 5 | 7 | 9 |
|---|---|---|---|---|
| $level_b$ | 0.0191 ± 0.0005 | 0.0182 ± 0.0005 | 0.0190 ± 0.0004 | 0.0193 ± 0.0007 |
| $level_a$ | 0.1758 ± 0.0006 | 0.1741 ± 0.0004 | 0.1724 ± 0.0009 | 0.1714 ± 0.0011 |
| $f_p$ | 0.0318 ± 0.0007 | 0.0189 ± 0.0004 | 0.0135 ± 0.0005 | 0.0115 ± 0.0002 |
| $t_p$ | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| $t_n$ | 0.9667 ± 0.0333 | 1.0000 | 1.0000 | 1.0000 |

Table 5.9: SFC SQL Slammer Results for $n = 8000$ determined to a 95% confidence interval

| k | 3 | 5 | 7 | 9 |
|---|---|---|---|---|
| $level_b$ | 0.0096 ± 0.0002 | 0.0095 ± 0.0002 | 0.0095 ± 0.0003 | 0.0098 ± 0.0002 |
| $level_a$ | 0.0910 ± 0.0008 | 0.0908 ± 0.0001 | 0.0904 ± 0.0003 | 0.0905 ± 0.0003 |
| $f_p$ | 0.0164 ± 0.0006 | 0.0098 ± 0.0003 | 0.0069 ± 0.0002 | 0.0057 ± 0.0003 |
| $t_p$ | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| $t_n$ | 1.0000 | 0.9667 ± 0.0333 | 1.0000 | 1.0000 |

For the network under observation, the results show that a filter configured for $n$ = 1000 and $k = 7$ gives the best detection rates with the minimum of resources. For this configuration, each pane in the SBFs and ASBFs is about 10 000 elements long, so the total memory requirements for all four filters with two panes each is around 80 000 elements. This is about twice the resources of the optimum configuration needed to detect the Code Red v2 worm. However, it should be noted that with those optimum settings, $n = 1000$ and $k = 3$, SFC was still able to correctly detect both SQL Slammer sources and had a true negative rate of only $0.9667 \pm 0.0333$. Increasing $n$ to 8000 did not appreciable improve detection accuracy.

The plots shown in Figure 5.16 and Figure 5.17 are selected results from one run of the SQL Slammer simulations. Figure 5.16 shows the monitored flow counts for a single uninfected client while Figure 5.17 presents that information for an infected client. Note that the flow count is capped at 500 once the worm becomes active; this is the limiter preventing SBF overflow. Figure 5.18 plots the flow counts of all active sources during this SQL Slammer simulation. It is given here to illustrate just how different the traffic profile of SQL Slammer is from that of normal traffic.

Based on these results and on the design of SFC, setting $n$ to the number of devices in the network under observation multiplied by the flow limiter value will result in near perfect detection accuracy. Decreasing $n$ from this level means that SFC will detect the first worms to start on the network but that if most sources are simultaneously generating worm traffic the filter will overflow. However, by the time this occurs SFC will have detected the worm's presence.The value of $k$ is less critical and since a smaller $k$ means fewer per-packet calculations, 3 or 5 are both good values for this parameter, with 5 resulting in slightly better detection accuracy.

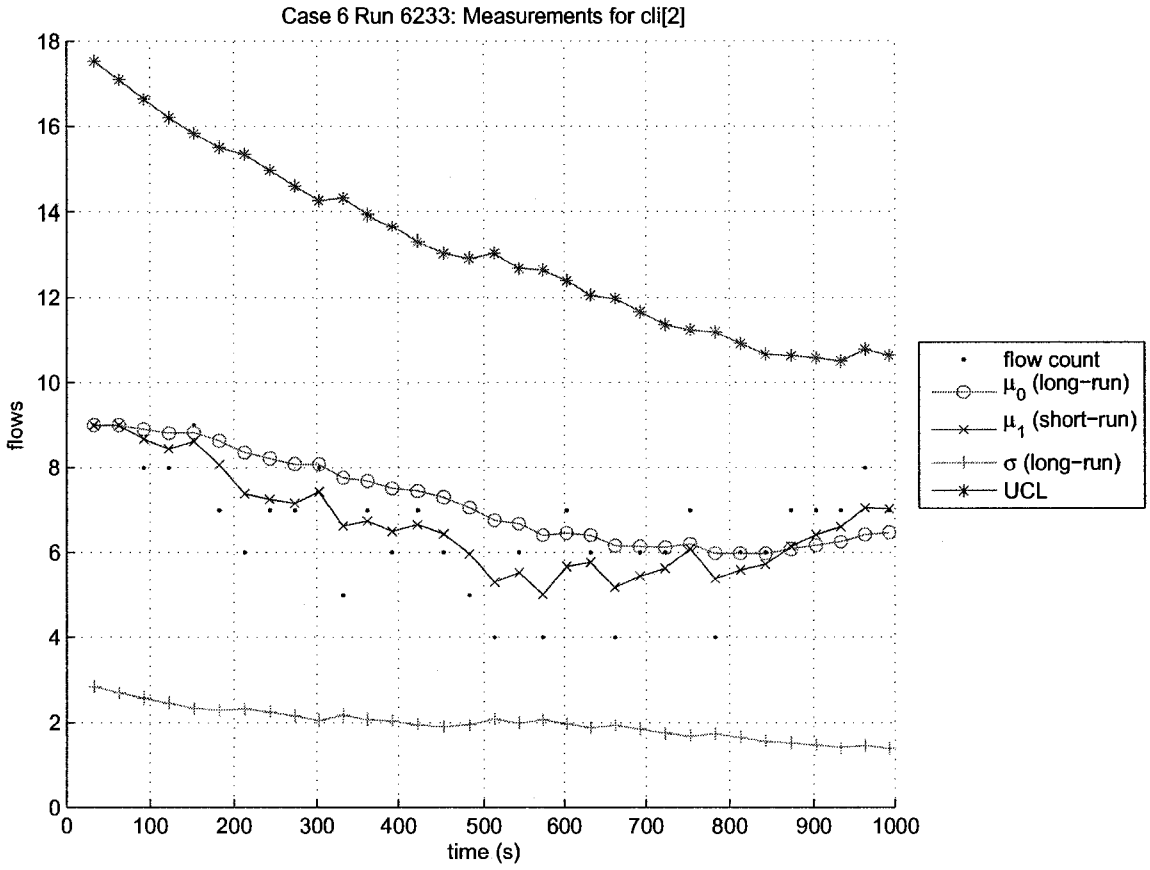


Figure 5.16: Flow counts for a client in the SQL Slammer simulation showing the the long-run moving average $\mu_0$, the short-run moving average $\mu_1$, the variance $\sigma^2$ and the Shewhart Upper Control Limit.

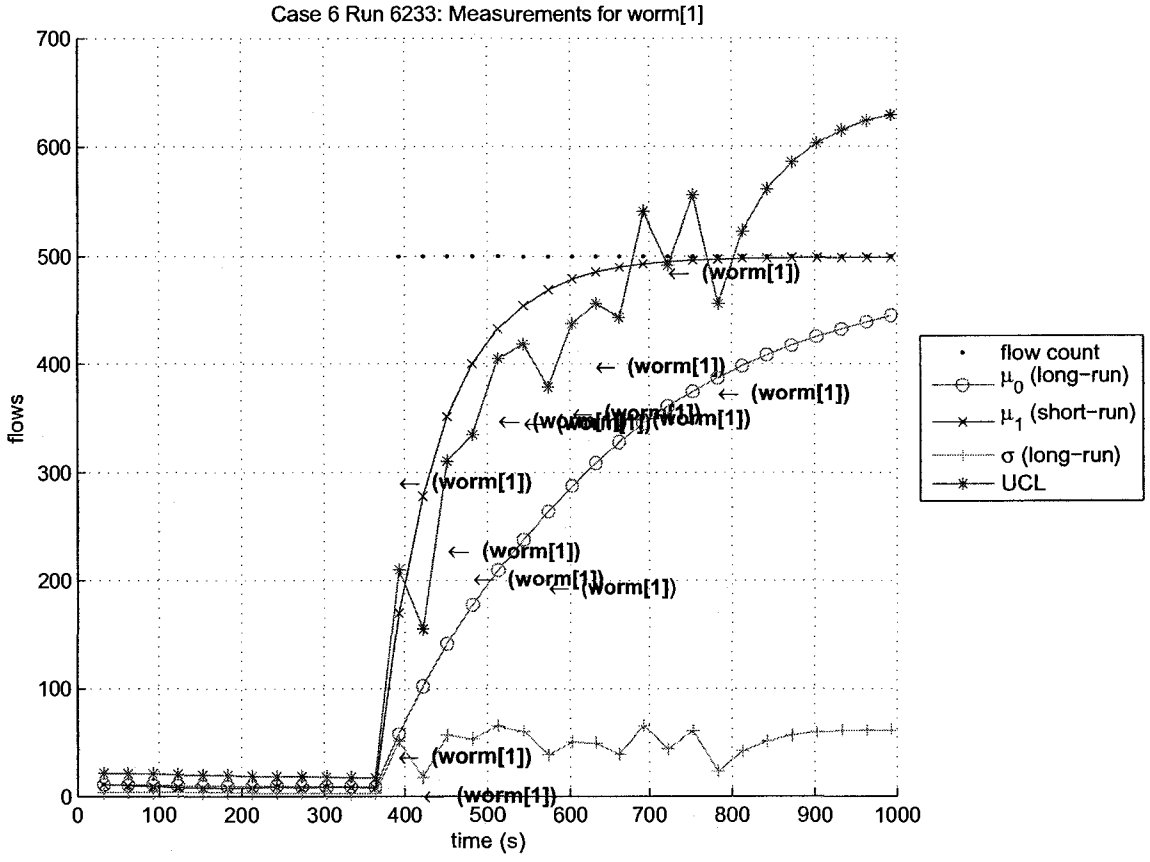


Figure 5.17: Flow counts for a SQL Slammer worm showing the the long-run moving average $\mu_0$, the short-run moving average $\mu_1$, the variance $\sigma^2$ and the Shewhart Upper Control Limit. SFC detected the worm as infected at the points marked with $\leftarrow$ (**worm**[**2**]).

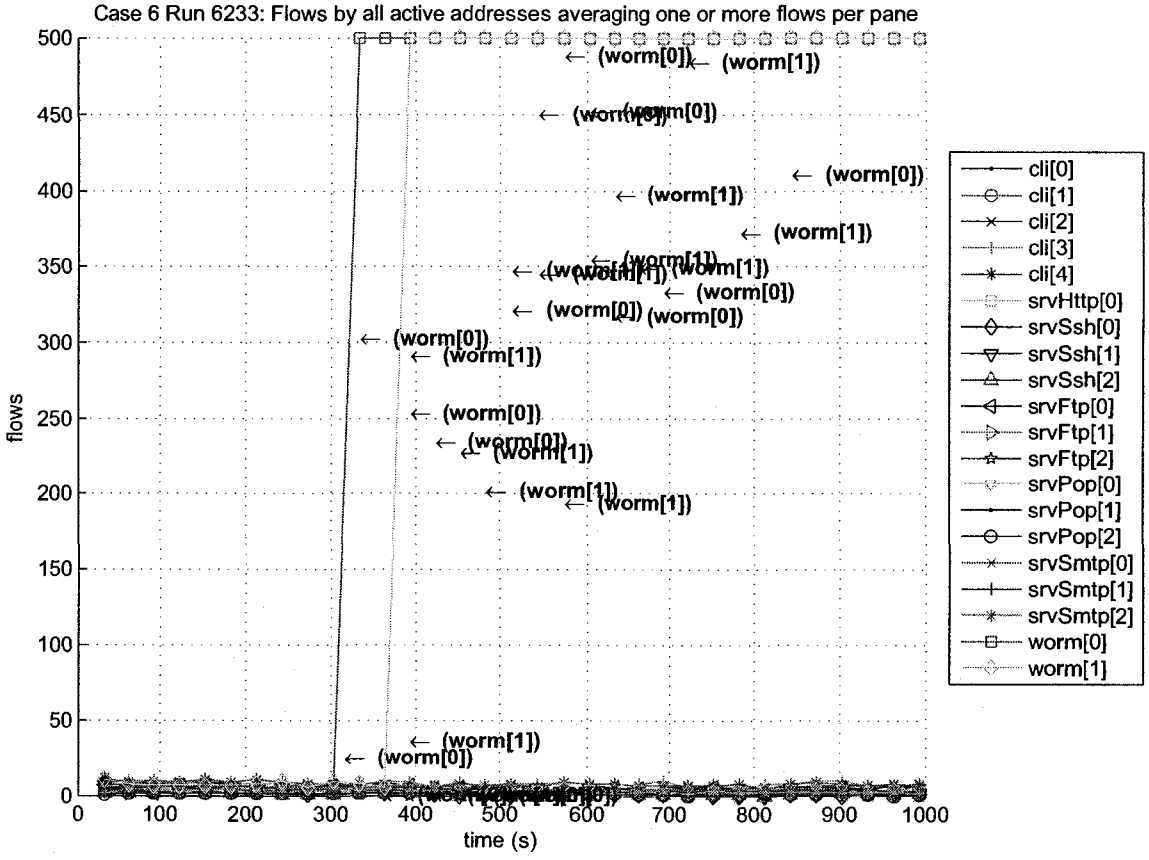


Figure 5.18: Flow counts for all sources in a SQL Slammer simulation that had at least one flow per time slot.

### 5.4.3 Performance Comparison

To properly compare SFC's performance against other worm detection schemes requires that they be completely developed within the simulation. This substantial task requires time and resources beyond that available for this work. The individual papers for each anomaly detection-based method present results showing very high detection accuracy within their test environments. It can be said that SFC gives comparable results. To draw any further conclusions is difficult without comparative testing.

# Chapter 6

# Conclusions

## 6.1 Conclusions

The Source Flow Counting (SFC) algorithm given in Chapter 5 partially solves the worm detection problem described in Chapter 4. Through passive monitoring of network traffic, the algorithm is able to detect abrupt changes in locality indicative of worm activity. Within the simulation used for experimentation, the algorithm was able to detect all infected sources with no false positives.

The SFC algorithm achieves sufficiently high detection accuracy while consuming a relatively small amount of computational resources compared to other approaches to the problem.

Application of the Shewhart Control Chart abrupt change detection method to source locality counts is an effective method for identifying infected hosts on a network.

The Bloom Filter and Arbitrary Spectral Bloom Filter data structures were successfully used to maintain source flow counts. The effect of the Bloom error on

detection accuracy was minimized when the false positive rate was kept below 5%.

## 6.2 Limitations

There are normal traffic patterns that will likely cause SFC to report a worm infection. The best example is peer-to-peer applications such as Gnutella. Particularly when these applications are in their search phase, where they seek out new peers, their activity results in an abrupt increase in locality that mimics that of a worm. This similarity has presented challenges to other worm detection algorithms [26].

A second possible limitation relates to the traffic properties of servers. In the simulation, the servers did not behave as major servers would on a physical network. The servers responded only to a small set of clients, whose requests were evenly divided across them. Normal server traffic is very bursty. A long period of stable traffic followed by a burst to new destinations would appear to be a worm when it is actually normal.

## 6.3 Future Research

Future work should spend some time exploring optimizations of the SFC algorithm. As currently implemented, the algorithm maintains statistics for all sources seen. A straightforward improvement is to limit statistics to members of given subnets or ranges. This change will also reduce error potential when querying the data store.

Any future investigation of SFC should include an increase in the capabilities of the simulator used to exercise the algorithm. Testing could be augmented, or possibly replaced by, traffic captures. Direct comparisons with other worm detection schemes

through their implementation into the SFC simulation environment would also be beneficial in evaluating the algorithm. The expanded simulator should also strive to model realistic server traffic to exercise SFC on active servers.

More substantially, a major change that has the potential to increase the detection accuracy in environments beyond the simulation used for this work is to consider packet contents. Certainly the suggestion is not that SFC become a deep-packet filtering method. However, as the probe packets generated by most worms are identical except for the destination address, the scheme could be improved by hashing packet contents into a Spectral Bloom Filter (SBF) or variant. When a host is suspected of infection, one of the decision factors would be a measure of how similar the traffic generated by the host is and how similar it is to other hosts, both healthy and infected, on the network under observation.

# References


[1] J. F. Shoch and J. A. Hupp, "The "worm" programs–early experience with a distributed computation," *Commun. ACM*, vol. 25, no. 3, pp. 172–180, 1982.

[2] D. Moore, C. Shannon, and J. Brown, "Code-red: a case study on the spread and victims of an internet worm," in *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment.* New York, NY, USA: ACM Press, 2002, pp. 273–284.

[3] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE Security & Privacy Magazine*, vol. 1, no. 4, pp. 33–39, 2003.

[4] "Malicious code attacks had $13.2 billion economic impact in 2001," http://www.computereconomics.com/article.cfm?id=133, Computer Economics, Jan 2002. [Online]. Available: http://www.computereconomics.com/article.cfm?id=133

[5] "Microsoft security bulletin ms01-044," http://www.microsoft.com/technet/security/bulletin/MS01-044.mspx, Microsoft Inc., August 2001. [Online]. Available: http://www.microsoft.com/technet/security/bulletin/MS01-044.mspx

[6] "Microsoft security bulletin ms02-039," http://www.microsoft.com/technet/security/bulletin/ms02-039.mspx, Microsoft Inc., July 2002. [Online]. Available: http://www.microsoft.com/technet/security/bulletin/ms02-039.mspx

[7] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson, "Characteristics of internet background radiation," in *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement.* New York, NY, USA: ACM Press, 2004, pp. 27–40.

[8] S. Staniford, V. Paxson, and N. Weaver, "How to own the internet in your spare time," in *Proceedings of the 11th USENIX Security Symposium.* Berkeley, CA, USA: USENIX Association, 2002, pp. 149–167. [Online]. Available: http://www.usenix.org/events/sec02/full_papers/staniford/staniford_html/

[9] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[10] M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java.* John Wiley & Sons, Inc., 1998.

[11] S. Cohen and Y. Matias, "Spectral bloom filters," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data.* New York, NY, USA: ACM Press, 2003, pp. 241–252.

[12] M. Roesch, "Snort-lightweight intrusion detection for networks," *Proceedings of USENIX LISA*, vol. 99, pp. 229–238, 1999.

[13] R. Stallman, "Gnu general public license," http://www.gnu.org/copyleft/gpl.txt, Free Software Foundation, Inc., 1989. [Online]. Available: http://www.gnu.org/copyleft/gpl.txt

[14] V. Jacobsen, C. Leres, and S. McCanne, "libpcap," http://www.tcpdump.org/, Lawrence Livermore Berkeley National Laboratory, 1994. [Online]. Available: http://www.tcpdump.org/

[15] J. Riordan, A. Wespi, and D. Zamboni, "How to hook worms [computer network security]," *IEEE Spectrum*, vol. 42, no. 5, pp. 32–36, 2005.

[16] L. Peishun, W. Jianbo, and H. Dake, "Worm detection using CPN," in *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, vol. 5, 2004, pp. 4941–4946.

[17] S.-M. Kang, I.-S. Song, Y. Lee, and T.-G. Kwon, "Design and implementation of a multi-gigabit intrusion and virus/worm detection system," in *Communications, 2006. ICC 2006. 2006 IEEE International Conference on*, 2006.

[18] B. Jiang and B. Liu, "High-speed discrete content sensitive pattern match algorithm for deep packet filtering," in *Computer Networks and Mobile Computing, 2003. ICCNMC 2003. 2003 International Conference on*, 2003, pp. 149–156.

[19] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, March 2006.

[20] "Clamav," http://www.clamav.net/, 2006. [Online]. Available: http://www.clamav.net/

[21] J. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks, "Internet worm and virus protection in dynamically reconfigurable hardware," *Proceedings of the Military and Aerospace Programmable Logic Device Conference*, 2003.

[22] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, 2004.

[23] Z. Chen, C. Lin, J. Ni, D.-H. Ruan, B. Zheng, Z.-X. Tan, Y.-X. Jiang, X.-H. Peng, A. an Luo, B. Zhu, Y. Yue, Y. Wang, P. Ungsunan, and F.-Y. Ren, "Antiworm npu-based parallel bloom filters in giga-ethernet lan," in *Communications, 2006. ICC 2006. 2006 IEEE International Conference on*, 2006.

[24] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[25] B. Madhusudan and J. Lockwood, "Design of a system for real-time worm detection," in *High Performance Interconnects, 2004. Proceedings. 12th Annual IEEE Symposium on*, 2004, pp. 77–83.

[26] G. Gu, M. Sharif, X. Qin, D. Dagon, W. Lee, and G. Riley, "Worm detection, early warning and response based on local victim information," in *Computer Security Applications Conference, 2004. 20th Annual*, 2004, pp. 136–145.

[27] B. Kim, S. Bahk, and H. Kim, "Fdf: Frequency detection-based filtering of scanning worms," in *Communications, 2006. ICC 2006. 2006 IEEE International Conference on*, 2006.

[28] V. Berk, G. Bakos, and R. Morris, "Designing a framework for active worm detection on global networks," in *Information Assurance, 2003. IWIAS 2003. Proceedings. First IEEE International Workshop on*, 2003, pp. 13–23.

[29] Y. Al-Hammadi and C. Leckie, "Anomaly detection for internet worms," in *Integrated Network Management, 2005. IM 2005. 2005 9th IFIP/IEEE International Symposium on*, 2005, pp. 133–146.

[30] T. Toth and C. Kruegel, "Connection-history based anomaly detection," in *3rd IEEE Information Assurance Workshop*, vol. 3. IEEE Computer Society Press, June 2002.

[31] S. Singh, C. Estan, G. Varghese, and S. Savage, "The earlybird system for real-time detection of unknown worms," University of California, San Diego, Technical Report CS2003-0761, 2003. [Online]. Available: http://www.cse.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2003-0761

[32] X. Chen and J. Heidemann, "Detecting early worm propagation through packet matching," USC/Information Sciences Institute, Tech. Rep., 2004.

[33] M. Martin, J. Robert, and P. Van Oorschot, "A monitoring system for detecting repeated packets with applications to computer worms," Carleton University Department of Computer Science, Technical Report TR-04-02.1, 2004. [Online]. Available: http://www.scs.carleton.ca/research/tech_reports/2004/TR-04-02.1.pdf

[34] M. Thottan and C. Ji, "Anomaly detection in IP networks," *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, vol. 51, no. 8, pp. 2191–2204, 2003.

[35] Q. Wu and Z. Shao, "Network anomaly detection using time series analysis," in *Autonomic and Autonomous Systems and International Conference on Networking and Services, 2005. ICAS-ICNS 2005. Joint International Conference on*, 2005, pp. 42–42.

[36] C. Gates and D. Becknel, "Host anomalies from network data," in *Systems, Man and Cybernetics (SMC) Information Assurance Workshop, 2005. Proceedings from the Sixth Annual IEEE*, 2005, pp. 325–332.

[37] Y. Cheng, Y. Dong, D. Lu, Y. Pan, and Z. Xiang, "Detecting randomly scanning worms based on heavy-tailed property," in *Networking, Sensing and Control, 2005. Proceedings. 2005 IEEE*, 2005, pp. 354–358.

[38] H. He, H.-L. Zhang, W.-Z. Zhang, M.-Z. Hu, and Z.-J. Tang, "Early warning of active worms based on multi-similarity," in *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, vol. 6, 2005, pp. 3876–3883.

[39] M. V. Mahoney and P. K. Chan, "Learning nonstationary models of normal network traffic for detecting novel attacks," in *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining.* New York, NY, USA: ACM Press, 2002, pp. 376–385.

[40] J. McHugh and C. Gates, "Locality: a new paradigm for thinking about normal behavior and outsider threat," in *NSPW '03: Proceedings of the 2003 workshop on New security paradigms.* New York, NY, USA: ACM Press, 2003, pp. 3–10.

[41] S. A. Hofmeyr, "An immunological model of distributed detection and its application to computer security," Ph.D. dissertation, University of New Mexico, 1999.

[42] M. Basseville and I. Nikiforov, *Detection of abrupt changes: theory and application.* Prentice Hall, 1993. [Online]. Available: http://www.irisa.fr/sisthem/kniga/

[43] S. Wei, J. Mirkovic, and M. Swany, "Distributed worm simulation with a realistic internet model," in *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 71–79.

[44] G. F. Riley, "The georgia tech network simulator," in *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research.* New York, NY, USA: ACM Press, 2003, pp. 5–12.

[45] B. Mah, "An empirical model of HTTP network traffic," in *INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 2, Kobe, 1997, pp. 592–600.

[46] S. Luo and G. A. Marin, "Realistic internet traffic simulation through mixture modeling and a case study," in *Proceedings of the 2005 Winter Simulation Conference*, M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, Eds., vol. 1, 2005, pp. 2408–2416.

[47] (2007, June) Omnet++ discrete event simulation system. [Online]. Available: http://www.omnetpp.org/

[48] (2007, June) Inet framework documentation and tutorials. [Online]. Available: http://www.omnetpp.org/staticpages/index.php?page=20041019113420757

# List of Acronyms

**ASBF** Arbitrary Spectral Bloom Filter

**ADL** Attack Definition Language

**BF** Bloom Filter

**BFWC** Bloom Filter With Counters

**CAM** Content Addressable Memory

**CPU** Central Processing Unit

**CPN** Coloured Petri Net

**CMS** Content Matching Server

**CBF** Counting Bloom Filters

**DSC** Destination-Source Correlation

**DEWP** Detector for Early Worm Propagation

**DDoS** Distributed Denial of Service

**DEC** Digital Equipment Corporation

**DED** Data Enabling Device

**EWMA** Exponentially Weighted Moving Average

**EWMV** Exponentially Weighted Moving Variance

**FDF** Frequency Detection-based Filtering

**FPGA** Floating Programmable Gate Array

**FTP** File Transfer Protocol

**HTTP** Hypertext Transport Protocol

**IDS** Intrusion Detection System

**ICMP** Internet Control Message Protocol

**IIS** Internet Information Server

**IMAP** Internet Message Access Protocol

**IP** Internet Protocol

**IPv4** Internet Protocol version 4

**MIB** Management Information Base

**PARC** Palo Alto Research Center

**PAS** Position-Aware Sub-patterns

**PHAD** Packet Header Anomaly Detection

**POP** Post Office Protocol

**RAM** Random Access Memory

**SFC** Source Flow Counting

**SBF** Spectral Bloom Filter

**SMB** Server Message Block

**SMTP** Simple Mail Transfer Protocol

**SNMP** Simple Network Monitoring Protocol

**SDRAM** Synchronous Dynamic Random Access Memory

**SRAM** Static Random Access Memory

**SSH** Secure Shell

**RTP** Regional Transaction Processor

**TCAM** Ternary Content Addressable Memory

**TCP** Transmission Control Protocol

**TTL** Time To Live

**UCL** Upper Control Limit

**UDP** Unigram Data Protocol

# List of Mathematical Symbols

$E_b$ Bloom error measuring a Bloom Filter (BF)'s false positive rate

$E_{SBF}$ Bloom error measuring a SBF's false positive rate

$\gamma$ Measure of expected BF utilization

$h_{a,b}(s)$ Hash function that maps a value $s$ from $U$ into the range $\{1, \ldots, m\}$ where $0 < a < m$ and $0 \leq b < m$

$k$ Number of hash functions associated with a BF

$\kappa$ Standard deviation multiplier for the Shewhart Upper Control Limit (UCL)

$\mu_0$ SFC flow count long-run mean

$\mu_1$ SFC flow count short-run mean

$n$ Number of keys inserted into a BF

$s$ Key value inserted into BF

$S$ Set of keys inserted into a BF

$\sigma^2$ SFC flow count long-run variance

$U$ Universe of all keys that could be inserted into a BF

$V$ Bloom Filter bit vector of length $m$ or SBF counter vector of length $m$