

# Design of a System for Real-Time Worm Detection

Bharath Madhusudan, John Lockwood  
Department of Computer Science and Engineering  
Washington University, St. Louis  
{bharath, lockwood} @arl.wustl.edu

## Abstract

*Recent well publicized attacks have made it clear that worms constitute a threat to Internet security. Systems that secure networks against malicious code are expected to be a part of critical Internet infrastructure in the future. Intrusion Detection and Prevention Systems (IDPS) currently have limited use because they can filter only known worms. In this paper, we present the design and implementation of a system that automatically detects new worms in real-time by monitoring traffic on a network. The system uses Field Programmable Gate Arrays (FPGAs) to scan packets for patterns of similar content. Given that a new worm hits the network and the rate of infection is high, the system is automatically able to detect an outbreak. Frequently occurring strings in packet payloads are instantly reported as likely worm signatures.*

## I. Introduction

Internet worms work by exploiting vulnerabilities in operating systems and application software that run on end systems. The attacks compromise security and degrade network performance. They cause large economic losses for businesses resulting from system down-time and loss of worker productivity. Several approaches to defend against worms are described in [1]:

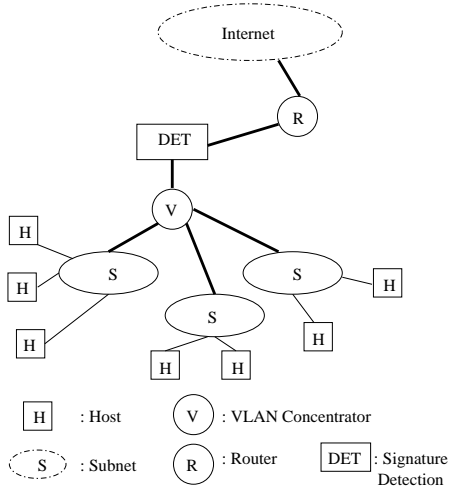
- **Prevention** : Worms usually work by exploiting vulnerabilities in software. Prevention involves writing secure code. Much work has been done by the Language Security community towards achieving this goal.
- **Treatment** : Programmers analyze the vulnerability that the worm exploits and release a “patch” that fixes it. However, it takes time to analyze and patch software. In addition, many end users may never apply the patch. As a result, a significant fraction of machines in the network remain vulnerable.

- **Host-based containment systems** : An example of host-based defense would be one that restricts the number of active connections. This would restrict the number of worm “probes” that hit other vulnerable machines on the Internet. A problem with host-based defense is that it is not easy to deploy. It requires that a privileged process be run and maintained on every end system in the network.
- **Containment by the network** : The advantage of using network based containment is that a small number of IDPSs can protect an entire corporate or university department network. Current intrusion detection systems use signatures to detect only known worms. Hence the network is left vulnerable to unknown worms.

This paper focusses on the design and implementation of a containment system that protects the network against unknown worms. Design goals of the system include:

- **Low reaction time** : The system should detect anomalies quickly and begin blocking malicious data flows so that damage caused by the worm can be limited.
- **High throughput** : Systems must keep up with today’s high speed links, like Gigabit Ethernet and OC-48, in order to monitor all traffic in real-time. Software-based systems can perform only a limited number of operations within the time period of a packet transmission. This necessitates the use of hardware.
- **Low cost** : The system should fit within the limits afforded by today’s FPGAs and ASICs.
- **Low false positive rate** : The system must generate a minimum number of false positives. Otherwise, warnings will be ignored and/or legitimate traffic will be blocked.
- **Robust to simple countermeasures** : The system should have the ability to detect polymorphic worms that work by inserting no-ops and random code in the worm payload.

Our system is designed to work in tandem with an IDPS such as the one presented in [2]. The short time taken to detect a worm signature allows an IDPS to be



**Fig. 1. Typical Configuration of System**

quickly programmed with the new signature and hence to limit further damage. A typical configuration of the system can be seen in Figure 1. The signature detection device (DET) resides between the Internet router and one or more subnets. It monitors traffic flowing between subnets and analyzes the content for patterns that resemble Internet worms.

## II. Related Work

The authors of [3] show why worms are a major threat to security. They model the spread of a worm with the susceptibles/infectives (SI) model. They also describe techniques that could be used by worm authors to improve the effectiveness of their code. Many worms today spread by random probing. The authors propose “hit-list scanning” wherein the worm author decides on a list of vulnerable machines before deciding to attack.

In [1] the requirements of a system such as the one presented in this paper are examined. The authors simulate the spread of a worm and examine the relative effectiveness of two approaches in containing spread. One approach is address blacklisting and another is content scanning and filtering (our approach). Their results demonstrate content filtering to be superior to address blacklisting, and highlight the need for a system capable of real-time worm detection.

The EarlyBird System [4] outlines the design of a system that detects unknown worms in real time. The system uses three criteria to flag a virus:

- 1) Identifying large flows of content : Since worms consist of malicious code, frequently repeated content on the network can be a useful warning of

worm activity. The authors of [5] outline methods to identify large flows. In this case, a hash of packet content in combination with destination port is taken to be the flow identifier because different worms target different ports. Inserting no-ops and random data at the beginning and the end of their code are still detected by computing multiple hash values over each packet.

- 2) Counting the number of unique sources and destinations : In [6] mechanisms to estimate the number of sources and destinations on a link are outlined. The sources and destinations corresponding to the signatures identified by the previous mechanism are counted. A drastic increase in the number of sources and destinations is thought to be more likely for a worm and less so for a Flash Crowd.
- 3) Determining port scans by counting number of connection attempts to unused portions of IP Address space : Certain portions of the IP address space are known to be unused. Since many worms use random probing, activity on those addresses may be taken as a warning sign.

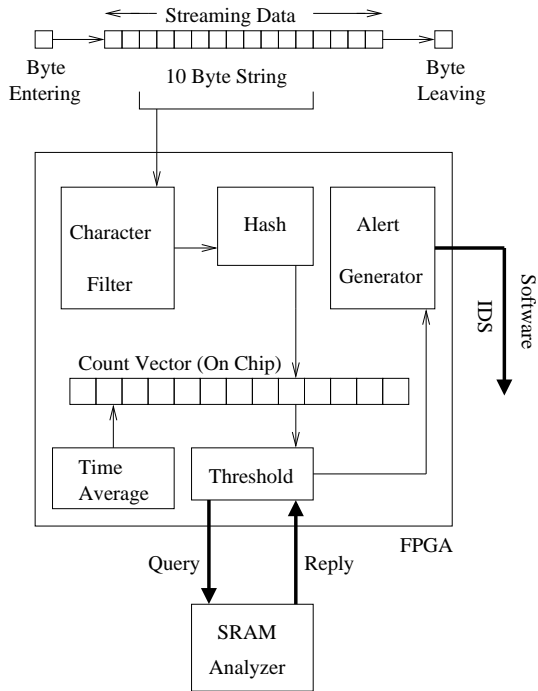
The authors of [7] use a data structure similar to ours to infer flow size distribution. Their method involves calculating the number of partitions of different counter values which is known to be a computationally intensive problem. They are aided by the fact that the distribution of flow sizes is Zipfian (a small number of flows consume most of the bandwidth). However, we are not at liberty of making assumptions about fixed length keyword distributions in random traffic. Hence, we believe that their methods are not applicable to our problem owing to their computational infeasibility.

## III. Overview of System

This system draws from two ideas presented in the EarlyBird system: (1) A worm detection system should look for frequently occurring content (2) It should be resistant to polymorphic techniques employed by worm authors. The contribution of our work lies in providing the detailed design of a system that has been implemented to run in hardware at high speed. A functional block diagram of the system is shown in Figure 2. The functionality of the blocks in the figure is explained in the following sections.

### A. Hash

To detect commonly occurring content, we calculate a  $k$ -bit hash over a 10 byte (80 bit) window of streaming data. In order to compute the hash, we generate a set  $k \times 80$  random binary values at the time the system is



**Fig. 2. Functional Block Diagram of the System**

configured. Each bit of the hash is computed as the xor over the randomly chosen subset of the 80-bit input string.

By randomizing the hash function, adversaries cannot determine a pattern of bytes that would cause excessive hash collisions. As with [4], the multiple hash computations over each payload ensure that simple polymorphic measures are thwarted.

## B. Count Vector

The hash value is used to index into a vector of counters. When a signature hashes to a counter, it results in the counter being incremented by one. At periodic intervals (henceforth called *timeouts*), the counts in each of the vectors are decremented by the average number of arrivals due to normal traffic. When a counter reaches a pre-determined threshold, an alert is generated and its value is reset to zero.

For the implementation of the circuit on a Xilinx FPGA, the count vector is implemented by configuring dual-ported, on-chip block RAMs as an array of memory locations. Each of the memories can afford one read operation and one write operation every clock cycle. This allows us to implement a 3-stage pipeline to read, increment and write memory every clock cycle. Since the signature changes every clock cycle and every occurrence of every

signature needs to be counted, the dual-ported memories allow us to write back the number of occurrences of one signature while another is being read.

## C. Character Filter

A *character filter* allows selected characters to be excluded from the hash computation. It ignores characters that are known to be innocuous, such as nulls, line breaks and new lines in text streams. To pre-process entire strings, a stream editor can be used to search for regular expressions and replace them with strings, as presented in [8].

## D. SRAM Analyzer

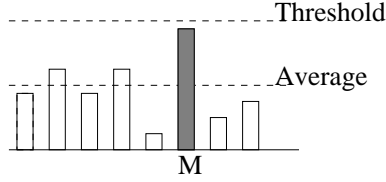
A *SRAM analyzer* holds suspicious signatures and estimates how often a certain signature has occurred. When an on-chip counter crosses the threshold, we hash the corresponding signature to a table in off-chip SRAM. The next time the same string causes the counter to exceed the threshold, we hash to the same location in SRAM and compare the two strings. If the two strings are the same, we know that it is not a false positive and we increment a counter in SRAM. If the two strings are different, we perform an overwrite of the SRAM location to reset the count and store the other string. By tracking the number of times specific signatures are matched, the hardware greatly reduces the number of false positives reported.

## E. Alert Generator

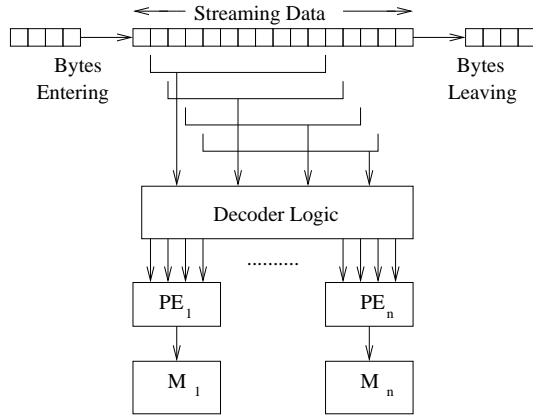
On receiving confirmation from the SRAM analyzer that a signature is indeed frequently occurring, a User Datagram Protocol (UDP) control packet is sent to an external PC that is listening on a well known port. The packet will contain the offending signature which is the string of bytes over which the hash was computed. The most frequently occurring strings are then flagged as being suspicious.

## F. Periodic Subtraction of Time Averages

There are two approaches to handle timeouts. One approach as mentioned in [5] resets the counters periodically. After a fixed window of bytes pass through the link, all of the counters are reset by writing the values to zero. However, this approach has a shortcoming which is illustrated in Figure 3. If the value of a counter corresponding to a malicious signature (labeled M in the figure) is *just* below the threshold at the time near the end of the timeout period, then resetting this counter will result in the signature going undetected.



**Fig. 3. Using Timeouts**



**Fig. 4. Minimizing contention by segmenting count vectors into smaller memories**

An alternative to resetting the counters is to periodically subtract an average value from all the counters. The average value is computed as the expected number of bytes that would hash to the bucket in each interval. Our simulations show that using periodic subtraction rather than counter reset provided a dramatic increase in the number of signatures found.

## IV. Design Considerations

So far, we have covered the major aspects of the system’s design. In this section, we detail how certain common and edge cases are accommodated in our design.

### A. Maintaining Throughput

In Figure 2, the system that was shown processes one byte at a time. To achieve higher throughput, multiple strings can be processed in each clock cycle. By augmenting the system with multiple parallel count vectors, multiple windows of bytes can be scanned in parallel. To get an accurate count of how often a signature has occurred, a sum of the corresponding counters in all count vectors needs to be computed in every clock cycle. Therefore, multiple reads are required out of the memory of each count vector every clock cycle.

To allow multiple memory operations to be performed in parallel, we segment the count vectors into multiple banks (Block RAMs). The higher order bits of the hash value are used to determine which Block RAM to access. The lower bits are used to determine which counter to increment within a given Block RAM. The modified system is shown in Figure 4. It is possible that more than one string could hash to the same Block RAM ( $M_1$  through  $M_n$  in the figure). We call this situation a *bank collision*. A bank collision is resolved using a priority encoder ( $PE_1$  through  $PE_n$  in the figure). This means that between 1 and 3 strings may not be counted every clock cycle for a system that runs at OC-48 line rates. The probability of collision,  $c$ , is given by :

$$c = 1 - \prod_{i=N-B+1}^{N-1} \frac{i}{N} \quad (1)$$

Where  $N$  is the number of Block RAMs used and  $B$  is the number of bytes coming in every clock cycle. We examine the impact of bank collisions on the reliability of the system in Section V.

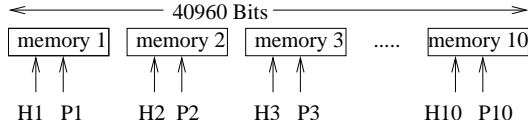
### B. Benign Strings

Certain strings such as the first several bytes of a HTTP request are commonly occurring strings but are not themselves a worm or virus. The system administrator can mark certain strings as being *benign*. A counter corresponding to a benign string is populated with a value beyond threshold. When a counter has this value, the circuit skips the increment and write back steps.

### C. Dealing with False Positives

We have considered how an adversary may try to overwhelm the system with false positives. One way to do this would be to send packets with the same string repeated multiple times. Another related approach would be to repeat the same string across multiple closely spaced packets. We address this problem with a window based scheme for counting signatures. If the window size is  $T$ , then signatures are counted only once in  $T$  bytes. This approach handles the trivial case of repeating strings within the same packet as well as an attacker sending several closely spaced packets with the same string. This scheme can be implemented by using a Bloom filter [9].

A Bloom filter is a data structure which can store a set of signatures compactly and can handle membership queries on these signatures. It has a zero false negative probability and a quantifiable false positive probability. The false positive probability,  $f$  of a Bloom filter is expressed by :



**Fig. 5. Smaller memories configured to act as a Bloom filter with desired access capacity**

$$f = (1 - e^{-\frac{nk}{m}})^k \quad (2)$$

where  $n$  is the number of signatures,  $k$  is the number of hash functions used and  $m$  is the length of the Bloom filter in bits.

The basic idea is to maintain two Bloom filters. For each incoming signature, a Bloom filter is queried to see if that signature has been counted before. If not, then the Bloom filter is programmed with that signature and the counter corresponding to that signature is incremented. At the end of the window of  $T$  bytes, we switch to the other Bloom filter to process the next window while we reset the one that is currently in use. From Equation 2 it is clear that the false positive probability increases with the number of signatures. We dimension the Bloom filter and  $T$  such that the maximum false positive probability is acceptably low. For  $f=0.001$  and  $m=40960$  bits (10 Block RAMs), we have  $T=2848$  (which exceeds the minimum packet length).

Each hash query and each programming action corresponds to two memory accesses in the  $m$ -bit long array. Thus for 10 hash functions, the Bloom filter memory should be able to support a maximum of 20 memory accesses per clock cycle. Such access capacity is achieved by using dual-ported on-chip memories as shown in Figure 5. Note that a traditional Bloom filter is a contiguous block of  $m$  bits. It is shown in [10] that this configuration does not affect the false positive probability.

#### D. The Relationship Between Threshold and Timeout

Since the timeout is approximately equal to the number of signatures that we see, the problem of determining threshold, given timeout reduces to a well studied problem in hashing. Namely – if  $m$  elements are hashed to a table with  $b$  buckets, the probability that the number of elements  $c$  hashing to the same bucket exceeds  $i$  is given by [11]:

$$Pr(c \geq i) \leq b \binom{m}{i} \frac{1}{m^i} \leq b \left( \frac{em}{ib} \right)^i \quad (3)$$

In our case,  $m$  signatures are hashed to  $b$  counters. In the above equation,  $c$  is the count of any given counter and  $i$  is the threshold. Hence, given a value of timeout, we can vary the threshold to make the upper bound on the probability

of a counter exceeding the threshold acceptably small. This in turn reduces the number of unnecessary SRAM accesses. The idea is that since incoming signatures hash randomly to the counters, anomalous signatures are likely to cause counters to exceed threshold for appropriately large thresholds.

We used the traces mentioned in Section V to simulate the system and check how well Equation 3 predicts the rate of SRAM accesses. General trends showed the number of unnecessary SRAM accesses to decrease with increasing threshold and decreasing timeout. But given Equation 3, one would expect a drastic drop in the number of SRAM accesses beyond a certain threshold. However, this was not the case in our simulation. It was found that the FTP traces contained recurring data and were not truly random. We expect that if the benign strings found in the traces were programmed into the system, then the number of SRAM accesses would follow the trend predicted by Equation 3 more closely.

## V. Performance Evaluation

We examine the resilience of the system to various worm attacks and the effect of memory bank conflicts on the system.

### A. Simulation Setup

Traces found in [12] were used to provide realistic background traffic for simulations. They contain all anonymous FTP connections to public FTP servers at the Lawrence Berkeley National Laboratory during a 10 day interval. Each of the traces spans a day. The methods used to anonymize the traces can be found in [13]. Two of the traces were concatenated and the following changes were made to them:

- Header information was stripped off and only packet payloads were preserved since only they are of interest. Packet boundaries were preserved.
- Synthetic worm signatures of lengths ranging from 500 to 50000 bytes were inserted randomly along packet boundaries in the trace data to constitute a fixed percentage of the overall traffic.

The simulation in the following sub-section also utilizes random traffic. Counters were incremented at random. A “worm” was introduced by incrementing a pre-determined set of counters deterministically.

The configuration consisted of 64 memory banks, each with 512 8-bit counters. The threshold was 230 and the timeout was 2.5 MBytes. By Equation 3 the probability of a counter crossing threshold is  $1.5 \times 10^{-6}$ .

## B. Effectiveness of System

**TABLE I. Effectiveness of System**

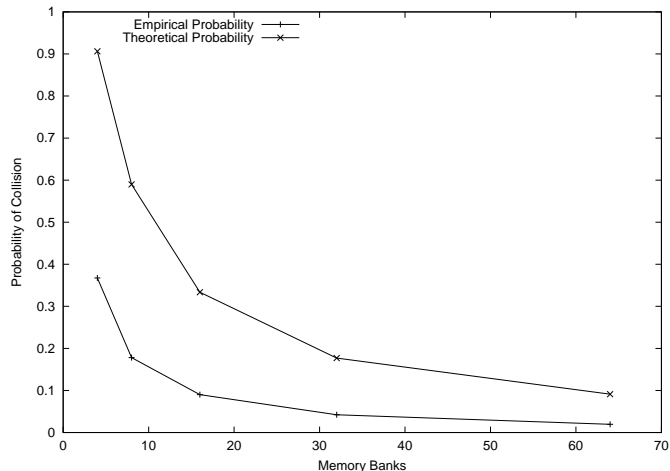
Signature Length (Bytes)	Concentration in Trace Data	Concentration in Random Data
500	1%	2%
1000	2%	3%
5000	3%	6%
20000	7%	14%
50000	11%	36%

In this section, we demonstrate the effectiveness of the system against various worm attacks. Since the system relies on a counter exceeding a threshold before it issues an alert, the more the occurrences of the signature, the better the chances of it being detected by the system. This means that there is an inverse relationship between the size of the worm and the sensitivity of the system to *stealth*. This idea is best illustrated with an example. Consider two worms of sizes 500 bytes and 10,000 bytes. In both cases, the worms would have to pass through the system a fixed number of times before being detected. But given the difference in sizes between the two worms, the concentration of the smaller worm in the Internet traffic will be far less than that of the larger worm. Worms vary widely in size from a few hundred bytes (SQL Slammer) to thousands of bytes (Sasser). Hence we have considered the robustness of the system to various worm sizes. The minimum required percentage for detection is taken to be a metric for effectiveness of the system.

The results are shown in Table I. The system is quite effective at detecting smaller worms at an early stage. But detecting larger worms becomes a much harder task. It is easy to see why this is the case. If the size of the worm exceeds the number of counters, the worm will cause all the counters to increment each time it passes (assuming the worm itself is sufficiently random). In this case, it will be indistinguishable from normal data. The results for trace data are better than for random data owing to the fact that there is regularity in real traffic.

## C. Impact of Bank Collisions

Results of this simulation answers the question of how much on-chip memory is needed in order to keep the number of collisions at an acceptable level. With the value of  $B$  in Equation 1 taken to be 4, the theoretical and empirical probability of collisions are shown for 4, 8, 16, 32 and 64 Block RAMs. The empirical probability of collision turns out to be much lower than the theoretical probability for large amounts of Block RAM. For instance, the theoretical probability of collision with



**Fig. 6. Memory collisions decrease with the use of more dual-ported memories**

64 Block RAMs (256 Mbits) is 0.09. But the empirical probability is only 0.02. Therefore, we conclude that with sufficiently high amounts of Block RAM the collision rate can be kept low. The disparity between the empirical and theoretical probabilities can be explained by the fact that we assume perfectly random data while calculating theoretical probability. However, as mentioned previously the trace data is not random.

## VI. Functional Prototype

The circuit discussed in this paper has been targeted to run on the FPX platform [14]. The Field-programmable Port Extender (FPX) provides for the processing of high speed network flows with large FPGAs. Application circuits can be downloaded to the Xilinx XCV2000E FPGA on the FPX card for processing of high speed network flows. Network traffic is clocked into this device using a 32-bit-wide data word. The circuit has been designed to fit into the layered protocol wrappers [15] that have been developed for the FPX. The wrappers allow the processing of high level packets in reconfigurable logic and allow a client application to send and receive packets with FPGA hardware. The functionality of the system has been modularized as follows:

- **Character Filter:** The character filter receives 4 bytes per clock cycle from the wrappers and determines which of the 4 bytes are valid (nulls, newlines and tabs are considered invalid).
- **Byte Shifter :** Shifts the signature by an appropriate number of bytes depending upon the output of the character filter.

- Hash : Calculates four hash values per clock cycle on four signatures.
- Priority Encoder : Resolves collisions between Block RAMs based on the high order bits of the hash value. The number of valid signatures per clock cycle varies from 0 to 4 due to the character filter.
- Read, Increment and Write Back : Each of these steps can be achieved in 1 clock cycle. The address and data buses of the block RAMs are accessed based on the output of the priority encoder.

A functional prototype of the system that can process traffic at full line rate has been implemented on the FPX as proof of concept. The circuit runs at a frequency of 91.5 MHz. Utilization statistics of the FPGA can be seen in Table II.

**TABLE II. Xilinx Virtex XVC2000E Device Utilization**

Resources	Device Usage	Utilization Percentage
Logic Slices	2804 out of 19200	14%
Block RAMs	34 out of 160	21%
Flip Flops	3360 out of 38400	8%

As can be seen, the circuit fits well within the resources afforded by contemporary FPGA logic. The speed at which the circuit operates allows processing at OC-48 line speeds on the FPX platform. The circuit has a pipeline delay of 7 clock cycles which introduces a delay of 70 ns in the datapath. Each of the modules outlined above has a 1 clock cycle delay. This implies that the system is capable to detecting and responding to worms in real-time.

## VII. Conclusions

The design of a system for detecting frequently occurring signatures in network traffic has been presented. We showed in Section 4 that the system is capable of scaling to very high throughputs. Since we exploit the parallelism afforded by hardware, the system is able to scan a far larger amount of traffic than software based approaches. Throughput was improved by hashing several windows of bytes in parallel to on chip memories. Each of these memories can be updated in parallel. In software, the hash followed by a counter update will require several instructions executed sequentially. The relationship between timeout and threshold in combination with the SRAM analyzer ensures a low false positive rate.

Current network monitoring tools rely on the system administrator's intuition to detect anomalies in network traffic. We believe that our system takes the process a step closer to automation by automatically detecting that a

spike in network traffic corresponds to frequently occurring content.

## Acknowledgements

We would like to thank the reviewers for their comments. Bharath would like to thank Sarang Dharmapurikar for useful discussions. This work was supported by Global Velocity. The authors of this paper have received equity from the license of technology to that company. John Lockwood has served as a consultant and co-founder to GlobalVelocity.

## References

- [1] D. Moore, C. Shannon, G. Voelker, and S. Savage, "Internet quarantine: Requirements for containing self-propagating code," in *IEEE INFOCOM*, 2002.
- [2] J. Lockwood, J. Moscola, D. Reddick, M. Kulig, and T. Brooks, "Application of hardware accelerated extensible network nodes for internet worm and virus protection," in *International Working Conference on Active Networks (IWAN)*, (Kyoto, Japan), Dec. 2003.
- [3] S. Staniford, V. Paxson, and N. Weaver, "How to own the internet in your spare time," in *Usenix Security Symposium*, Aug. 2002.
- [4] S. Singh, C. Estan, G. Varghese, and S. Savage, "The earlybird system for the real-time detection of unknown worms." UCSD, Department of Computer Science, Technical Report CS2003-0761, Aug. 2003.
- [5] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *ACM SIGCOMM*, (Pittsburgh, PA), Aug. 2002.
- [6] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Internet Measurement Conference*, Oct. 2003.
- [7] A. Kumar, M. Sung, J. Xu, and J. Wang, "Data streaming algorithms for efficient and accurate estimation of flow size distribution," in *ACM SIGMETRICS*, June 2002.
- [8] J. Moscola, M. Pachos, J. W. Lockwood, and R. P. Loui, "Implementation of a streaming content search-and-replace module for an internet firewall," in *Hot Interconnects*, (Stanford, CA, USA), pp. 122–129, Aug. 2003.
- [9] B. Bloom, "Space/time tradeoffs with hash coding with allowable errors," in *CACM*, pp. 13(7):422–426, May 1970.
- [10] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," in *Symposium on High Performance Interconnects (HotI '03)*, Aug. 2003.
- [11] G. Gonnet and E. Brewer, "Handbook of algorithms and data structures," (Reading, MA: Addison-Wesley, 1991).
- [12] R. Pang and V. Paxson, "Anonymized Traces." <http://www-nrg.ee.lbl.gov/~anonymized-traces.html/>.
- [13] R. Pang and V. Paxson, "A high level programming environment for packet trace anonymization and transformation," in *ACM SIGCOMM*, (Karlsruhe), Aug. 2003.
- [14] J. W. Lockwood, "An Open Platform for Development of Network Processing Modules in Reprogrammable Hardware," in *IEC DesignCon'01*, (Santa Clara, CA), pp. WB-19, Jan. 2001.
- [15] F. Braun, J. Lockwood, and M. Waldvogel, "Protocol wrappers for layered network packet processing in reconfigurable hardware," *IEEE Micro*, vol. 22, pp. 66–74, Jan. 2002.