FPGA-based SoC for Real-Time Network Intrusion Detection using Counting Bloom Filters

Jared Harwayne-Gidansky, Deian Stefan and Ishaan Dalal
The Center for Signal Processing, Communications and Computer Engineering Research
The Cooper Union for the Advancement of Science and Art
51 Astor Place, Room 406B, New York, NY 10003 USA
{harway, stefan, ishaan}@cooper.edu

***Abstract*—Computers face an ever increasing number of threats from hackers, viruses and other malware; effective Network Intrusion Detection (NID) before a threat affects end-user machines is critical for both financial and national security. As the number of threats and network speeds increase (over 1 gigabit/sec), users of conventional software based NID methods must choose between protection or higher data rates.**

To address this shortcoming, we have designed a hardware-based NID system-on-a-chip using data structures called Counting Bloom Filters (CBFs). Our design has extremely high throughput (up to 3.3 gigabits/sec) and can successfully detect and mitigate known threats, and is, to our knowledge, the only known CBF based NID system-on-a-chip to be implemented on a Virtex 4 FPGA.

In this project, we present the first optimized, Counting Bloom Filter based Network Intrusion Detection FPGA SoC (system-on-chip) implemented on a Virtex 4 FPGA: our design is scalable through further parallelization and, to our knowledge, is one of the highest throughput NID systems in existence.

***Index Terms*—Data structures, Computer network security, Field programmable gate arrays**

I. Introduction

Viruses, worms, and hacker attacks on networks cost billions of dollars (*Klez* virus: $9 billion [1]) and affect hundreds of thousands of users (*MSBlast* worm: over 350,000 hosts) [2]. Additionally, recent high profile attacks on our national security infrastructure—such as the infiltration of DoD networks by Chinese hackers [3]—reveal that defense against network intrusion is now a matter of not just financial, but also national security.

It is incorrect to assume that each user or individual machine on a network is secure [4]. This is due to the large number of machines that need to be secured; each of which requires time, technical expertise and constant vigilance to ensure protection against the latest threats. To address these short comings organizations are moving toward *Network Intrusion Detection* (NID): preemptive detection of hacking attacks, worms, and other threats present in data *when it enters the network*, i.e. before it can reach the user's machine.

Network Intrusion Detection involves looking at patterns in network data that match known signatures stored in an existing *threat-database*. Currently, NID is performed using dedicated devices from manufacturers such as Cisco. These devices are essentially full-blown computers using software solutions such as pattern matching with hash tables to perform NID [5]. However, as both network data rates and the number of potential threats increase, the ability to effectively scan network data in real time using such devices becomes impractical. For example, while Gigabit (1000 megabits/sec) networks are increasing in popularity, a typical Cisco NID device [5] has a maximum throughput of only 150 megabits/sec.

As a result, there has been a move toward custom hardware implementations of network intrusion detection which can have significantly higher throughput [2], [6]. Rather than fabricate custom ICs, Field-Programmable Gate Arrays (FPGAs) are used for this purpose. FPGAs are reconfigurable chips that contain programmable logic and can perform multiple, complex operations in parallel, and have become the dominant platform for hardware–based NID. Such NID devices can offer real-time protection against a wide array of threats while achieving throughputs of over 1 gigabit/sec [7]. To ensure the highest possible throughput we used *Bloom filters* an efficient data structure for hardware-based pattern matching. Bloom Filters store a compact randomized representation of the threat-database, allowing for a number of advantages over traditional hash tables which have a large amount of overhead and require enough memory to store the full threat-database. The basic Bloom Filter can be improved into a *Counting Bloom Filter* which allows for even more flexibility.

In this project, we present the first optimized, counting-bloom-filter-based Network Intrusion Detection "system-on-chip" (i.e., on a single FPGA chip) implemented on a Virtex 4 FPGA: our design is scalable through further parallelization and, to our knowledge, is one of the highest throughput NID systems in existence.

We first present the probabilistic mathematics that are the basis of Bloom Filters, and the evolution of Bloom Filters into Counting Bloom Filters. We then discuss the design process and computer architectural challenges in hardware implementations of Bloom Filters. Finally, we use hardware Counting Bloom Filters to set up a Network Intrusion Detection system, test it with real-world threats and present performance benchmarks.

The design and results presented here were part of an undergraduate project, advised by I. Dalal, at the Cooper Union

Outline

The outline of our paper is as follows:

- **Overview of Bloom Filters and Hash Functions:** The structure and mathematical background of Bloom Filters, followed by performance analysis. •Choosing a hardware-efficient hash function for Bloom Filters.
- **Counting Bloom Filters (CBFs) and overview of FPGAs:** The limitations of Bloom Filters, and their evolution into CBFs. •Discussion of trade-offs in CBF design. •A brief introduction to FPGAs and their suitability for Bloom Filter implementation.
- **Hardware Implementations:** Three Bloom Filters and three Counting Bloom Filters running at up to 3.3 gigabits/sec. •Design results and conformance to theoretical predictions.
- **Network Intrusion Detection using Counting Bloom Filters:** Discussion, implementation and analysis of an innovative one-chip system for real time, full-speed (gigabit ethernet) detection of network threats using hardware Counting Bloom Filters.
- **Conclusion:** Review of presented material, summary of results, and discussion of future work.

II. Overview of Bloom Filters and Hash Functions

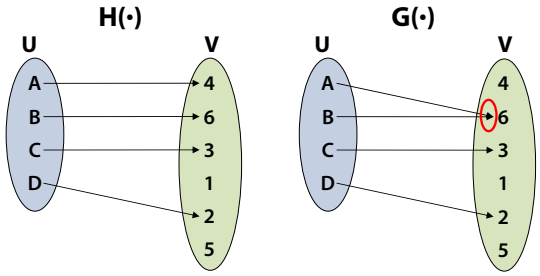
A Bloom Filter is a data structure used to determine if a given piece of data belongs to a set [8], [9]. To perform this "pattern-matching", Bloom Filters do not store the full set, but only a much smaller randomized representation; this makes them much more efficient than conventional hash tables [10].

A. Definition

The goal of the Bloom Filter is to determine if a given piece of data (e.g., network packet) belongs to a predefined set (e.g., threat-database). Bloom Filters consist of two key parts: an array of bits (known as the *Bloom array* and initialized to ZEROs) and one or more *hash functions*. The Bloom array has a length of m-bits and a total of k hash functions. First, the Bloom Filter must be *programmed* with a predefined set which consists of n elements. This is accomplished by hashing each element of the set through the k hash functions; the output of the hash functions are used as indices (addresses) for the m-bit array and the bits at the corresponding addresses are 'set' (i.e. changed to ONE).

After programming, to check if a piece of data is a member of the set (known as *querying*), that data is hashed. If all the corresponding bits of the Bloom array are set (ONE), then that piece of data is *almost certainly* part of the predefined set; if *any* of the corresponding bits are not set (ZERO), the data is *definitely not* a part of the set (i.e., for NID, there is no threat). The check thus consists of performing an AND logical operation on the corresponding bits and checking if the output is ZERO or ONE.

Thus, a Bloom Filter can definitely tell if a piece of data does not belong to the set (i.e., is *not* a threat), but there is a very small chance of a *false positive*: data is identified as belonging to the set (a threat) when it actually is not. This *false positive probability* can be controlled and made arbitrarily small, so is not usually an issue.



(a) An *ideal* hash function, H; no collisions.

(b) A *non-ideal* hash function, G; note the collision.

Fig. 1. Ideal and non-ideal hash functions mapping set U to set V.

B. Choosing Hash Functions for Hardware Bloom Filters

The Bloom Filter's accuracy depends critically on the choice of hash function [9]. A hash function is essentially a one-way mapping between two sets. For two sets U and V, an *ideal* (or *perfectly random*) hash function uniformly maps each element in U to a unique element in V (Fig. 1a). Real-world hash functions are rarely ideal and can have *collisions*, i.e. the mapping is no longer unique and more than one element in U may map to the same element in V (Fig. 1b). Expressed mathematically, for elements $x, y \in U$, $x \neq y$, an ideal hash function $H(\cdot)$ will have no collisions, i.e. $H(x) \neq H(y)$; on the other hand, for a non-ideal hash function $G(\cdot)$ collisions can occur (i.e., $G(x) = G(y)$).

Collisions in a Bloom Filter lead to false positives and decrease its effectiveness. Hence, hash functions for Bloom Filters in hardware must not only be as close to ideal as possible (i.e., have a low collision rate), but also have low hardware complexity (which allows for high throughput). Additionally, since the length of the Bloom array varies depending on the application, the hash function should have variable-length outputs. Given these requirements we used the H_3 hash functions, a member of the *Universal Class* of hash functions. These hash functions are low-complexity randomized algorithms that attempt to distribute the data inputs along the set of hash outputs as evenly as possible; with the appropriate choice of random numbers they have low collision rates and can be configured for arbitrary length outputs [11]. H_3 hash functions are an excellent choice for Bloom Filters, and have recently been successfully used for this purpose [7], [8].

C. H_3 Hash Functions

Given its desirable properties (low-complexity and low collision rate), we decided to use the H_3 hash function. H_3 hash functions can be computed with only exclusive-OR (XOR) operations, making them very efficient for hardware implementation. Let us illustrate H_3 hashes with an example. If the data inputs to the hash function are, say, 8 bits and the hash outputs must be 4 bits, we begin by choosing a matrix **D**

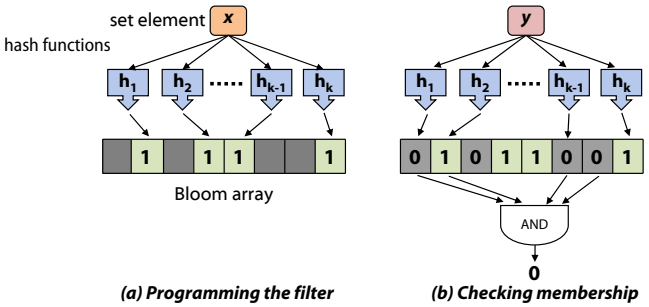


Fig. 2. (a) Programming a Bloom Filter (b) Querying a Bloom Filter; the data being checked is not a member of the set (AND=0)

of eight 4-bit random numbers $\{d_1,\ldots,d_8\}$. Then, an element $x = \texttt{00011010}$ is hashed as:

$$\mathbf{D} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \end{bmatrix} = \begin{bmatrix} \texttt{1100} \\ \texttt{0001} \\ \texttt{0100} \\ \texttt{1110} \\ \texttt{1001} \\ \texttt{0110} \\ \texttt{1101} \\ \texttt{1111} \end{bmatrix}$$

$$\begin{aligned} \therefore h(x) &= h(\texttt{00011010}) \\ &= x \cdot \mathbf{D} \\ &= \texttt{0} \cdot d_1 \oplus \texttt{0} \cdot d_2 \oplus \texttt{0} \cdot d_3 \\ &\quad \oplus \texttt{1} \cdot d_4 \oplus \texttt{1} \cdot d_5 \oplus \texttt{0} \cdot d_6 \\ &\quad \oplus \texttt{1} \cdot d_7 \oplus \texttt{0} \cdot d_8 \\ &= \texttt{1110} \oplus \texttt{1001} \oplus \texttt{1101} \\ &= \texttt{1010}, \end{aligned}$$

where $\cdot$ (AND) and $\oplus$ (XOR) are logical operations. The result $h(x)$ is used to address the bit array of the Bloom Filter; this is demonstrated in Fig. 2. By using different random matrices, as many different hash functions as necessary can be generated. It is important to note that H_3 hash functions behave like an ideal hash function when used with real-world data. This is because real-world data can be treated as random for the purposes of determining hash function performance [12].

D. Formal definition of Bloom Filters

Mathematically, a Bloom Filter is defined as a compact representation of a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements stored in an array of m bits (all of which are initially ZERO); the k hash functions $\{h_1,\ldots,h_k\}$ are independent and have a range of $\{1,\ldots,m\}$ (i.e. each hash function maps uniformly to a random number from 1 to m).

1) Programming the Bloom Filter (Fig. 2a): The set S must first be programmed in the Bloom Filter (e.g., S can be a certain set of known threats). For all $x \in S$, each x is hashed by the k hash functions h_i $(1 \le i \le k)$ and the bits in the Bloom array corresponding to the outputs $h_i(x)$ are set to ONE as shown in Fig. 2a.

2) Querying the Bloom Filter (Fig. 2b): After the Bloom Filter is programmed, it is put into operation. To check if an item y is a member of S (e.g., for intrusion detection, does a packet contain a virus signature y?), you must hash it with all k hashes and then check to see if all the bits in the Bloom array corresponding to $h_i(y)$ are set to ONE (Fig. 2b). If they are *not* ONE, then y is *definitely not* a member of S; but, if they are *all* ONEs, then we assume y is a member of S, although it may not be with a very small finite probability (i.e. a *false positive*) [9].

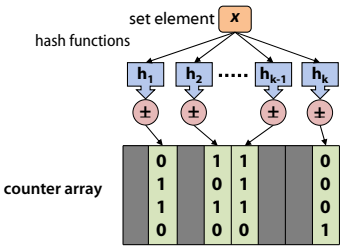


Fig. 3. A Counting Bloom Filter with 4-bit up/down counters.

E. False Positive Probability

A false positive occurs when an element that is *not* in the set is hashed and all the corresponding k bits in the Bloom array turn out to be ONE; the probability of this occurring is

$$\frac{n \cdot k}{m} = \ln(2), \tag{1}$$

where k is the optimal number of hash functions for a given m-bit Bloom array that will hold n entries.

A derivation of this can be found in the appendix.

III. Counting Bloom Filters and FPGA Overview

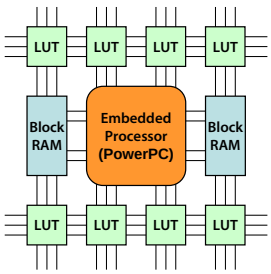
A major limitation of Bloom Filters is that once an element has been programmed into the array, it cannot be deleted without erasing and reprogramming the filter from scratch. This is because there is no way to reset the bits corresponding to one element to ZERO and ensure that *none* of those bits were needed by another element still in the Bloom Filter. To address this deficiency, Bloom Filters have been adapted into *Counting* Bloom Filters (CBFs).

A CBF is identical to a standard Bloom Filter in concept, but differs in implementation: the Bloom array for a CBF consists of *counters* and not individual bits (Fig. 3). By using counters for a CBF, you can add *and remove* items from the CBF. Instead of setting a bit when programming it, you increment $(+1)$ the corresponding counters; when deleting an item you decrement (-1) the corresponding counters. To query the CBF for the existence of some piece of data, you check if all the corresponding counters in the Bloom array are non-zero (like checking if all corresponding bits are set to ONEs in a standard Bloom Filter).

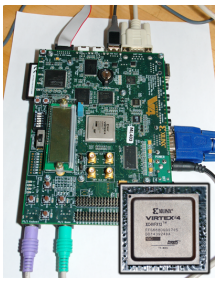
A. Designing Counting Bloom Filters

Since each counter in the CBF array has a limited size, some new challenges are introduced, such as what to do if a counter overflows because too many elements are added. Empirically, a reasonable counter size that minimizes overflows for most applications is four bits [9].

Regardless of the counter size, there is always the possibility that you reach the maximum value of a counter and an overflow occurs: in this situation it is best to leave the counter at its maximum value [9]. While this could potentially cause a *false negative* (which is impossible with standard Bloom Filters), if the deletions from the CBF are more-or-less random (as is the case with real-world data), and the counter is of



(a) Block diagram of an FPGA



(b) ML403 dev. board with the Virtex-4 FX12 FPGA (inset)

Fig. 4. FPGA structure and our FPGA system-on-chip

TABLE I
RESULTS FOR BLOOM FILTER PROTOTYPES
(k=8 HASHES, m=16,384 BITS; 4 BLOCK RAMS EACH)

Bloom Filter	Input Size (bytes)	Freq (MHz)	Slices	Throughput (Gbps)
Regular	2	205.1	127	3.28
	4	216.0	254	6.91
	8	200.6	529	12.8
Counting	2	205.3	185	3.28
	4	203.9	319	6.52
	8	201.6	594	12.9

a reasonable size (i.e. 4 bits), the average time until a false negative occurs is extremely large [9].

We used CBFs for our network intrusion detection as they allow for dynamic re-programming (add/delete) of the filter. This flexibility is highly desirable to ensure that the filter is up to date, e.g. if a new worm outbreak occurs (add), or if an attack is no longer a threat because the affected software was patched (delete), etc. The CBFs drawbacks are outweighed by their advantages, the abundant resources on the FPGA (mitigating the memory requirement), and the very low probability of a false negative.

B. Overview of FPGA Architecture

The Field Programmable Gate Array is made up of 4-bit or 6-bit Look-Up Tables (LUTs) which can be programmed to compute any Boolean function of 4 or 6 inputs, respectively. Thousands of these LUTs make up an FPGA and are connected by reprogrammable interconnects, allowing for immense flexibility in implementing any digital function which can be optimized by performing many operations in parallel. Fig. 4a shows the basic structure of a contemporary FPGA; apart from LUTs, the FPGA also contains blocks of fast static random-access memories (block RAMs) and (optionally) an embedded microprocessor, such as the PowerPC, to perform control and data transfer functions. The FPGA chip itself is usually integrated on a board that contains additional I/O and storage peripherals such as Ethernet controllers, DDR RAM modules, and serial ports.

The FPGA is an excellent platform for hosting a Bloom Filter as it contains the fast RAM (i.e. Block RAMs) needed to minimize the time it takes access the bit array, and its LUTs are ideal for performing multiple H_3 hashes (XOR operations) in parallel. It is due to this fact that a number of Bloom Filter implementations on FPGAs exist for applications such as spell checking, medical imaging and network intrusion detection [7], [8], [10], [13].

IV. HARDWARE IMPLEMENTATIONS AND ANALYSIS

Before moving on to Network Intrusion Detection, we first designed, optimized and implemented the necessary Bloom Filters on the FPGA. We have implemented three Bloom Filters in both standard and Counting variants on the Xilinx *Virtex-4 FX12* FPGA hosted on the ML403 development board (Fig 4b). The FPGA consists of 5,472 *slices* (a slice contains two LUTs), 36 block RAMs (of 18-kbits each), and an embedded PowerPC 405 processor. The embedded processor eliminates the need for a separate computer to perform control functions such as TCP/IP network communication, making intelligent decisions in the NID threat detection process, etc; this enabled us to make this a "system-on-a-chip."

A. Standard Bloom Filter Implementation

The Bloom array ($m = 16384$-bits) is stored in Block RAM on the FPGA. To maximize throughput and minimize the control and routing logic, we give each hash function exclusive access to the block RAM via a dedicated I/O port. We achieve this as follows: since each block RAM has two I/O ports (i.e., you can perform two independent reads/writes to different addresses per clock cycle), and we have $k = 8$ hashes, we split the bit array across $\frac{k}{2} = 4$ block RAMs that contain $\frac{m}{4} = \frac{16384}{4} = 4096$ bits each. While each hash function has to be modified so its output falls within the range of its "exclusive" block RAM, this does not effect the false positive rate or the effectiveness of the Bloom Filter [7]. An AND gate is used to perform the final membership check.

B. Counting Bloom Filter Implementation

As described in section III, going from a standard Bloom Filter to a counting Bloom Filter design simply involves modifying the Bloom array so that each "bit" is a 4-bit counter instead. We implemented our CBF by storing the counter values in block RAM; each block RAM (there are 36 on the FPGA) has a capacity of 18-kbits. The RAM was used at a 4-bit depth, and 4-bit adders were included to increment/decrement the RAM counters while comparators were used to check if the counters were 'set' (i.e., its value was greater than zero). Using the LUTs as the "counters" was considered (as a counter is a 4-bit Boolean function) but rejected due to an insufficient number of LUTs on the FPGA and the large routing overhead which would have substantially decreased performance.

CBFs are somewhat slower at programming than a standard Bloom Filter: while the standard Bloom Filter takes one clock cycle to change a bit, the CBF takes two clock cycles to modify a counter; the original value must be read, incremented (for an add) or decremented (for a delete), and then written

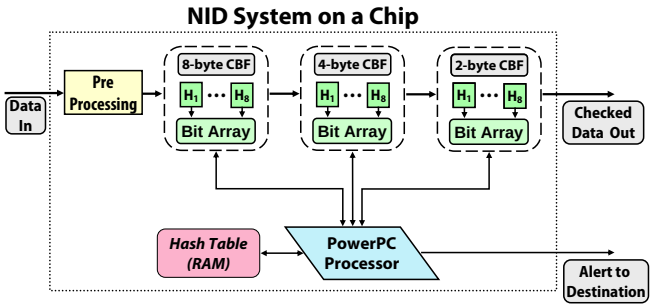


Fig. 5. A simplified block diagram of our NID design.

back to update the counter. The extra programming time is immaterial since after the initial programming, the CBF array only needs occasional reprogramming (for updates). We note that checking (reading) the CBF array, which is by far the major operation in an NID device, only requires a single clock cycle; thus, using CBFs has negligible effect on throughput compared to standard Bloom Filters.

Our maximum throughput was approximately 3.3 Gbps; a complete summary of our results for the CBF prototype can be found in Table I.

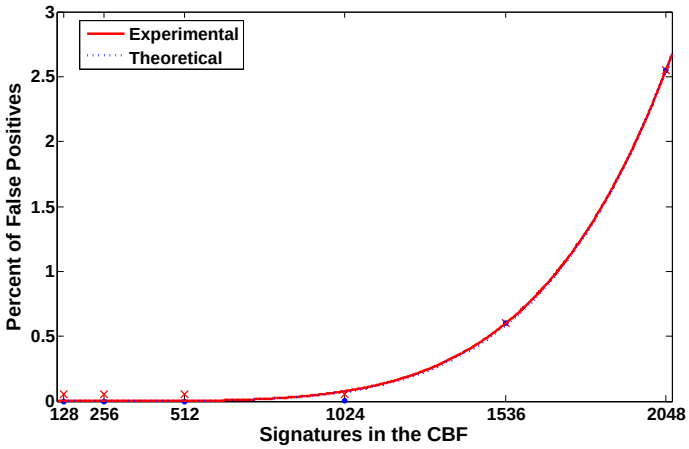
V. Application: Network Intrusion Detection

As mentioned earlier, networks face a large variety of threats from hackers and malware that can cause damage in the billions of dollars and breach national security. Assuming that end-users can protect their systems is both inaccurate and ineffective. Detecting malicious data or events at the network boundaries (Network Intrusion Detection) allows for preemptive and broader protection from threats and creates a single point to maintain network security (as opposed to at every machine in the network). The NID device is installed at the gateway to the network it is protecting.

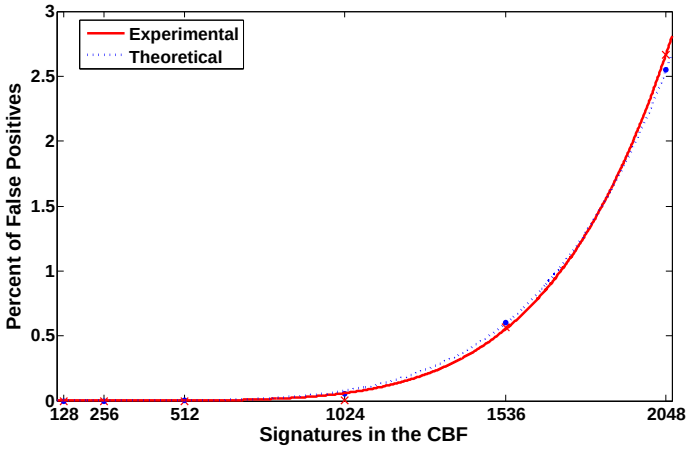
We have designed and implemented a Network Intrusion Detection (NID) device using Counting Bloom Filters to detect and mitigate threats before they enter a network.

Fig. 5 shows the structure of our NID device. The input data—Ethernet packets (frames)—is fed to the preprocessor which extracts the IP packet, stripping the headers and forwarding the host information to the PowerPC and the TCP/UDP payload to the Counting Bloom Filters. The filters take 2-, 4- and 8-byte data inputs respectively; this is because the threat patterns (such as `ILOVEYOU` for the *Love Bug* worm) can be of variable size; the preprocessor must be used to keep track of patterns longer than 8-bytes—we have not implemented this capability yet.

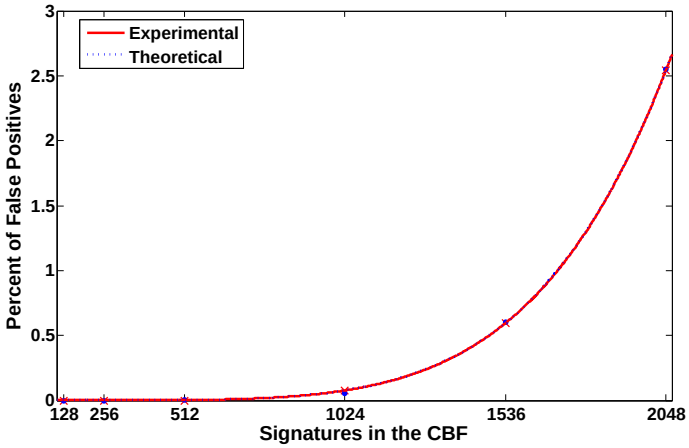
A flow chart showing how our NID device operates is shown in Fig. 7. After the CBFs, an intelligent processor is needed to analyze the output of the CBFs and decide if a threat is detected. Instead of offloading this to a separate computer or microprocessor (e.g., as is done in some commercial NID systems [5]) we decided to design the NID device as a complete system-on-chip by using the embedded PowerPC processor. Once the PowerPC detects a threat from the CBF outputs, it eliminates the possibility of a false positive by



(a) 2-byte Bloom filter results



(b) 4-byte Bloom filter results



(c) 8-byte Bloom filter results

Fig. 6. Experimental vs. Theoretical False Positive Rates for (a) a 2-byte Bloom filter, (b) a 4-byte Bloom filter and (c) a 8-byte Bloom filter ($m = 16384$-bits, $k = 8$ hash functions and $n = 128, \ldots, 2048$ elements)

verifying if that piece of data in question is present in the secondary hash table (stored in DDR RAM on the FPGA board). Note that this hybrid "bloom filter + small secondary hash table" approach is much faster and uses less memory than a full hash-table approach.

If the threat is confirmed, the NID device drops the data packet and sends an Internet Control Message Protocol (ICMP) datagram of type `DESTINATION UNREACHABLE` with explanation code "Host administratively prohibited" to the source of the malicious packet. Furthermore, a UDP packet is sent to a logging facility where the system administrator can further analyze the alerts. The NID device can also take more drastic steps such as IP blocking of known/persistent hosts to avoid denial of service attacks on the NID device. Additionally, since we use CBFs–as opposed to standard Bloom Filters–an administrator is able to modify the database (add or remove signatures) in real-time using the serial port on the ML403 development board to rapidly respond to emerging threats.

NID System-on-a-Chip: Testing and Performance

For our proof-of-concept NID implementation, we chose a realistically-sized $n = 2048$ element data set containing signatures for a variety of intrusions: viruses, trojans, worms, denial of service attacks, brute-force unauthorized access attempts and spam. The signatures are based on a subset of the malicious payloads included in the industry-standard Snort (ver. 2.4) rule-set [14].

Given $n = 2048$, with the additional constraint of a small false positive probability ($< 2.5\%$), to arrive at an optimal Bloom array length of $m = 16384$ bits and $k = 8$ hash functions (this was done using eqns. (5) and (6) found in the Appendix). Since the signatures vary in length, each Bloom Filter accepts a different-sized input: 2-, 4-, or 8-bytes respectively.

In evaluating the performance of the Bloom Filters we programmed each of the filters with an increasing number of elements, from $n = 128$ to 2048, based on the Snort rule-set and custom-crafted signatures, testing the false positive rate on simulated traffic. It is important to note that the custom-crafted signatures were used in cases where the number of Snort payload signatures was not sufficient (i.e. $< n$).

Based on real world data [15] and the testing methodologies presented in [16], we simulated random traffic on the order of $\approx$ 70,000 connections of which 2.58% were attacks (i.e., connections whose payloads contained signatures which were used to program the filters). The number of normal and attack connections in a *time interval* were distributed according to a Poisson process resulting in normal and "attack" intervals; in an attack interval the majority of the connections had payloads with malicious signatures [16].

Fig. 6 compares our Bloom Filter's false positive rate with the false positive rate of a theoretical Bloom Filter for signatures of length 2-bytes (6a), 4-bytes (6b) and 8-bytes (6c). As is apparent, our experimental false positive rates closely track the theoretical values.

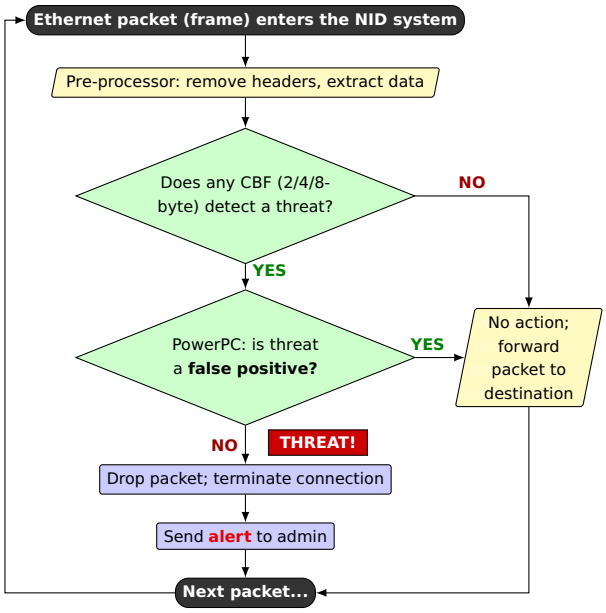


Fig. 7. Flowchart of Network Intrusion Detection using CBFs on an FPGA

VI. Conclusion

We presented a single-chip, FPGA-based hardware Network Intrusion Detection (NID) system using Counting Bloom Filters. Our design identifies and can neutralize threats such as hackers and viruses at the network boundary before they can attack end-user computers. As network data rates increase, such real-time preemptive action may prove impractical for software NID solutions.

Counting Bloom Filters are efficient, randomized data structures that are much faster and use less memory than the hash tables usually used in software applications. Consequently, our NID device has a throughput of $\sim$3.3 Gbps—over an order of magnitude higher than commercial software-based NID systems.

Future work includes increasing the number and variety of threat-signatures that the system can detect as well as full scale testing on a live network. We hope that by increasing the flexibility and speed of effective Network Intrusion Detection we can help secure computers against malicious attacks, reduce associated financial losses and prevent the compromise of national security.

Appendix
False Positive Probability Derivation

Here we derive the probability of getting a false positive when querying a Bloom Filter.

Consider an m-bit array; since the output of each hash function is uniformly and independently distributed over the range $\{1,\ldots,m\}$, the probability that a bit is set (i.e., `ONE`) after a single hash of one element is $\frac{1}{m}$. The probability that a bit is *not* set (i.e., `ZERO`) is, therefore, $(1-\frac{1}{m})$.

To program a Bloom Filter using k hash functions on a set with n entries, a total of $n \cdot k$ hashing operations are

performed. The probability p that a bit is *not* set after the filter is completely programmed is therefore

$$p = Prob(\text{bit is not set}) = \left(1 - \frac{1}{m}\right)^{n \cdot k} \quad (2)$$

Assuming m is large (which is true in practice), p can be approximated as

$$p \approx \lim_{m\to\infty} \left(1 - \frac{1}{m}\right)^{n \cdot k} = \left[\lim_{m\to\infty} \left(1 + \frac{1}{-m}\right)^{-m}\right]^{\frac{n \cdot k}{-m}}, \quad (3)$$

where the expression inside the square brackets is the definition of the exponential (e) function. Thus, the probability p that a bit is not set is approximately

$$p = Prob(\text{bit is not set}) \approx e^{\frac{-n \cdot k}{m}} \quad (4)$$

The complementary probability (p') that a bit *is set to* ONE is simply $p' = (1-p)$. A false positive occurs when all k bits for the entry being hashed are set to ONE, thus the false positive probability, $p(FP)$, is

$$\begin{aligned} p(FP) &= Prob(k \text{ bits are set}) \\ &= (p')^k = (1-p)^k \\ &\approx (1 - e^{-\frac{n \cdot k}{m}})^k \end{aligned} \quad (5)$$

Given eq. (5), the false positive probability $p(FP)$ is minimized when

$$\frac{n \cdot k}{m} = \ln(2), \quad (6)$$

which also gives the optimal value for one of the three variables $\{m, n, k\}$ if the other two are kept constant; e.g., the optimal number of hash functions k for a given m-bit Bloom array that will hold n entries is $k = \frac{m}{n} \cdot \ln(2)$.

REFERENCES

[1] R. Lemos, "Counting the cost of slammer," CNet, Feb. 2003.

[2] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks, "Internet worm and virus protection in dynamically reconfigurable hardware," in *Proc. of Mil. and Aero. Programmable Logic Device (MAPLD)*, Sep. 2003.

[3] J. Swartz, "Chinese hackers seek U.S. access," USA Today, Mar. 2007.

[4] B. Achoido and M. Kessler, "Worm, virus threat grows," USA Today, Apr. 2003.

[5] "Installing Cisco intrusion prvention system appliances and modules 5.1," Cisco.

[6] M. Attig and J. Lockwood, "SIFT: SNORT intrusion filter for TCP," in *Proc. 13th Symp. on High Performance Interconnects*, Aug. 2005, pp. 121–127.

[7] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," in *Hot Interconnects*, Aug. 2003, pp. 44–51.

[8] I. L. Dalal and F. L. Fontaine, "A reconfigurable FPGA-based 16-channel front-end for MRI," in *40th Asilomar Conf. on Sigs., Sys. and Comps, 2006*, Oct./Nov. 2006, pp. 1860–1864.

[9] A. Broder and M. Mitzenmacher, "Network applications of Bloom Filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[10] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: an aid to network processing," in *SIGCOMM '05*, 2005, pp. 181–192.

[11] M. Ramakrishna, E. Fu, and E. Bahcekapili, "A performance study of hashing functions for hardware applications," in *Int'l Conf. Computing and Information*, 1994, pp. pp. 1621–1636.

[12] M. Mitzenmacher and S. Vadhan, "Why simple hash functions work: exploiting the entropy in a data stream," in *Proc. of the 19th annual ACM-SIAM SODA*, 2008, pp. 746–755.

[13] M. Attig, S. Dharmapurikar, and J. Lockwood, "Implementation results of bloom filters for string matching," in *12th Symp. on Field-Programmable Custom Computing Machines*, Apr. 2004, pp. 322–323.

[14] M. Roesch, "SNORT-lightweight intrusion detection for networks," in *LISA 1999: USENIX 13th Systems Administration Coinferecne*, Nov. 1999, pp. 229–238.

[15] R. Lippmann, R. Cunningham, D. Fried, I. Graf, K. Kendall, S. Webster, and M. Zissman, "Results of the DARPA 1998 Offline Intrusion Detection Evaluation," in *Recent Advances in Intrusion Detection*, vol. 99, 1999, pp. 829–835.

[16] J. Cabrera, B. Ravichandran, and R. Mehra, "Statistical Traffic Modeling for Network Intrusion Detection," in *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE Computer Society Washington, DC, USA, 2000, pp. 466–473.

Jared Harwayne-Gidansky (S'06–GS'08) received the B.E. degree in Electrical Engineering from the Cooper Union for the Advancement of Science and Art. He is currently a Graduate Electrical Engineering student at The Cooper Union, a Graduate Research Fellow with the *S*ProCom*^{2} research laboratory at Cooper Union, and a reviewer for *IEEE Potentials*. His interests include Control Theory, Stochastic Processes, Signals Processing, Programming, and Applied Mathematics.

Deian Stefan (S'06) will receive the B.E. degree in Electrical Engineering in 2009 from the Cooper Union for the Advancement of Science and Art. He is currently an Undergraduate Research Fellow in the *S*ProCom*^{2} research laboratory at Cooper Union; his interests include signal processing, reconfigurable systems, cryptography, and operating systems.

Ishaan Dalal (S'02–GS'07) received the B.E. degree in Electrical Engineering from the Cooper Union for the Advancement of Science and Art in 2006. Since then, he has been a Graduate Research Fellow with the *S*ProCom*^{2} research laboratory at Cooper Union and an Adjunct Instructor of Electrical Engineering. His interests include statistical signal processing, information theory and system design for communications. He is currently pursuing a doctorate in Electrical Engineering at the University of Texas-Austin.