A Configurable-Hardware Document-Similarity Classifier to Detect Web Attacks

Craig Ulmer*, Maya Gokhale+
*Sandia National Laboratories, CA +Lawrence Livermore National Laboratory
cdulmer@sandia.gov, maya@llnl.gov

***Abstract*— This paper describes our approach to adapting a text document similarity classifier based on the Term Frequency Inverse Document Frequency (TFIDF) metric [11] to reconfigurable hardware. The TFIDF classifier is used to detect web attacks in HTTP data. In our reconfigurable hardware approach, we design a streaming, real-time classifier by simplifying an existing sequential algorithm and manipulating the classifier's model to allow decision information to be represented compactly. We have developed a set of software tools to help automate the process of converting training data to synthesizable hardware and to provide a means of trading off between accuracy and resource utilization. The Xilinx Virtex 5-LX implementation requires two orders of magnitude less memory than the original algorithm. At 166MB/s (80X the software) the hardware implementation is able to achieve Gigabit network throughput at the same accuracy as the original algorithm.**

I. Introduction

Document similarity is a well-known metric that can be used in many contexts. A literature search may require search and selection of documents similar to a specific document. A digital library may wish to categorize documents according to their content using a similarity measure. Market research applications analyze keyword occurrences in web pages to help target advertising.

In this work, we put the document similarity metric to an unconventional use. Following the algorithm of Gallagher and Eliassi-Rad [5], we use document similarity to detect cyber attacks in HTTP traffic. Our document collection consists of HTTP web page requests. The role of the classifier is to detect attacks embedded in the HTTP messages. An example of an SQL injection attack embedded in a request is shown in Figure 1. The request for an "odbc" connection has been modified to include an SQL command to delete a data table.

Gallagher's novel approach builds a classifier to recognize cyber attacks in the HTTP request "documents." Each document's Term Frequency Inverse Document Frequency (TFIDF) [11] vector is computed and compared to reference documents' TFIDF vectors according to the well known vector space model. The reference documents describe the characteristics of seven different types of cyber web attacks plus the normal category. The classifier assigns each incoming document to one of the eight classes.

Building on the Gallagher approach, we have devised a hardware, *streaming* algorithm to classify HTTP requests as they arrive in a text stream transmitted over a high-speed internetwork. Using our method, real-time classification of HTTP requests into attack vs. normal categories serves as an advanced intrusion prevention method capable of detecting and thwarting web attacks as they occur.

Our algorithm has been optimized for streaming computation in a hardware pipeline that exploits on-chip distributed RAM and Block RAM for high performance. The hardware building blocks are highly configurable, and their configuration parameters are generated semi-automatically through analysis of the training data set. The classifier demonstrates throughput of 166MB/s (80X speedup over the sequential algorithm) at the same accuracy as Gallagher's sequential algorithm.

II. Sequential Algorithm

A. Document Similarity

Document similarity using term weights is a well-understood Information Retrieval technique. It uses a term frequency $tf(t,d)$ to record the number of times a search term t appears in a document d normalized by the total number terms in d:

$$tf(t,d) = \frac{count(t,d)}{\sum_{v \in d} count(v,d)} \quad (1)$$

To compensate for frequently appearing terms, the term frequency is inversely weighted by the term's frequency in the entire document collection. The $tfidf$ of t is sim-

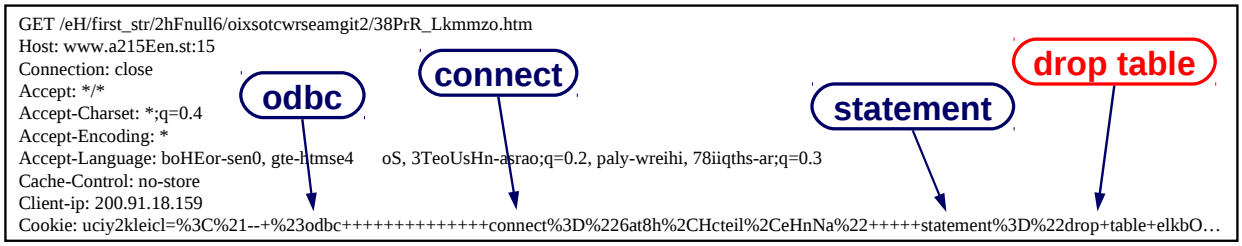

Fig. 1. Sample HTTP request with malicious database activity

ply t's tf multiplied by its inverse document frequency idf relative to a document collection D:

$$idf(t) = \frac{log\,|D|}{|\{d \in D : t \in d\}|} \tag{2}$$

$$tfidf(t,d) = tf(t,d) \cdot idf(t) \tag{3}$$

A document d is characterized by its $tfidf$ vector V_d. Each component i of V_d holds the $tfidf$ score of the i'th term in the document collection. In this vector space model, the similarity between two documents d and a is the cosine of the angle θ between the $tfidf$ vectors V_d and V_a:

$$sim(d,a) = cos(\theta_{V_d,V_a}) \tag{4}$$

i.e.

$$sim(d,a) = \frac{V_d \cdot V_a}{\|V_d\|\|V_a\|} \tag{5}$$

or equivalently

$$sim(d,a) = \frac{\sum_{t \in d \cap a} tfidf(t,d) \cdot tfidf(t,a)}{\sqrt{\sum_{t \in d} tfidf(t,d)^2}\sqrt{\sum_{t \in a} tfidf(t,a)^2}} \tag{6}$$

B. Data Set

Our experiments use a data set released by the European Conference on Machine Learning (ECML) and the 11th European Conference on the Principles and Practice of Knowledge Discovery in Databases (PKDD) in 2007 as part of the 2007 Discovery Challenge [8]. The data set contains more than 120,000 labeled HTTP requests, of which 50,000 are for training, and 70,000 constitute the testing data. 70% of the training requests are normal, valid requests, and 30% contain cybersecurity attacks. In the testing data, 60% are normal and 40% are attack.

There are seven different sorts of attacks: cross site scripting, SQL injection, LDAP injection, XPATH injection, directory path traversal, command execution, and Server Side Include (SSI) attacks. The 2007 challenge was to detect and characterize the attacks. The work of Gallagher [5] applies the TFIDF method to train and

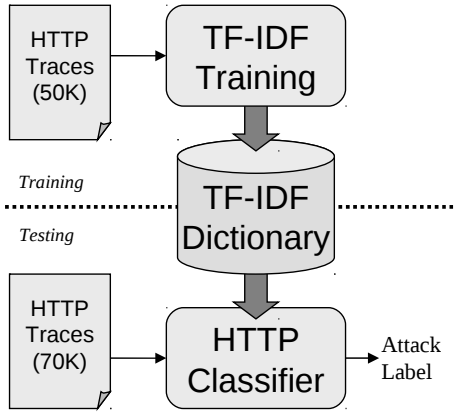

Fig. 2. Training and testing the classifier

run a classifier to detect attacks. To train the classifier, all requests of a specific attack type in the training set are combined into a single document, resulting in eight reference documents. $tfidf$ vectors are computed for each reference document.

In the testing phase, each HTTP request in the testing data set is considered to be a single document, and the cosine similarity between the incoming request and each reference attack document is computed. Once all similarity scores are computed, a threshold operation is applied to remove low values. This operation improves the quality of the results when comparing the attack scores to the normal score, most notably a large number of terms are utilized. This parameter is typically set during training to balance precision and recall [9] statistics. A lower value implies a more hostile environment, where it is important to filter anything that appears malicious, even if it means mistakenly filtering harmless traffic. The document is classified as belonging to class of the most similar reference document. The process is shown in Figure 2.

On the ECML/PKDD 2007 Discovery Challenge data

set, the Gallagher algorithm gave accuracy of 94% in distinguishing attack vs. normal and 91% in correctly distinguishing the type of attack. The single-threaded Java software implementation of the algorithm runs at about 2MB/s on a standard PC workstation (2.2GHz, quad core, 8GB memory).

III. Hardware Algorithm

A straightforward interpretation of Equation 6 requires access to the entire document collection a priori. Since our goal is a streaming, hardware approach, we study the equation for optimization opportunities. The major challenges for an efficient hardware implementation of the TFIDF classifier are to devise 1) a streaming, online method that 2) minimizes the number of multiplies and divides, and 3) minimizes memory usage.

Streaming: In a streaming environment, the data must be processed in a single pass with very limited buffering. We observe that according to Equation 6 a term's idf scores cannot be computed until the entire document collection has been scanned. In a streaming environment, the document collection is never complete, and therefore the idf we used is based on the training document collection, which is pre-computed and can simply be retrieved from the dictionary.

Simplification: In our application we generate the eight attack type scores for an input document and label the document based on the largest score. There is an opportunity to simplify Equation 6 in two places where normalization is applied to allow the scores of one document to be compared to another: (1) the numerator's $tfidf$ contains a normalization by the number of terms in the input document and (2) the denominator's $\sqrt{\sum_{t \in d} tfidf(t,d)^2}$. Both operations are constant across all attack types for an input and therefore are removed.

With these simplifications it is possible to translate Equation 6 into a form that can be facilitated through table lookups.

$$sim(d,a) = \frac{\sum_{t \in d \cap a} count(t,d) \cdot C_1 \cdot C_2}{C_3} \tag{7}$$

where

$$\begin{aligned} C_1 &= idf(t,a) \\ C_2 &= tfidf(t,a) \\ C_3 &= \sqrt{\sum_{t \in a} tfidf(t,a)^2} \end{aligned} \tag{8}$$

Given that all three constants are relative to a particular attack type, information can be combined to reduce table lookup size. In our approach we combine C_2 and C_3 and refer to the value as a $categoryWeight$. Thus the classifier model requires nine statistics for every term: a single C_1 value to indicate the term's idf and eight $categoryWeight$ values to indicate how relevant the term is to a particular attack type.

Memory Minimization: To maintain high throughput, the classifier model or $dictionary$ must be stored on-chip. In this application, the limited memory presents a formidable challenge, as the full dictionary for the ECLM training data set uses 47MB. We seek to encode the dictionary in a combination of logic and memory to approximately 128KB, a compression factor of two orders of magnitude. Three different optimizations are employed to help reduce the dictionary memory footprint. First we truncate the dictionary so that only the N most significant terms per attack type are utilized. Second, we quantize the $categoryWeight$ and idf data values in the dictionary in order to simplify numerical diversity and allow better information compression. Finally, we utilize a hashing technique that employs an array of Bloom filters to represent the dictionary data. Each of these optimizations is discussed in detail.

A. Truncate Dictionary

The classifier model or dictionary holds statistical $tfidf$ information (idf and $categoryWeights$) for each term encountered in the training data set. The number of terms in the dictionary ultimately dictates how much information is available during classification. A reasonable accuracy can be achieved with only a small number (e.g., 32) of high-value keywords. Supplementing these terms with a large number (e.g., hundreds to thousands) of less important but still relevant terms typically improves accuracy until the classifier becomes overtrained. After this point, accuracy may stay constant or degrade. Increasing the number of dictionary terms increases the amount of data a classifier must maintain.

A test program was constructed to evaluate the impact that truncation and algorithm modification have on accuracy. A dictionary was constructed based on the $training$ data set. We varied the number of terms available in the dictionary and then measured the accuracy of both the original and streaming versions of the algorithm. Evaluation was performed using the training and testing data sets as inputs to the classifiers. The results of the experiments are presented in Figure 3.

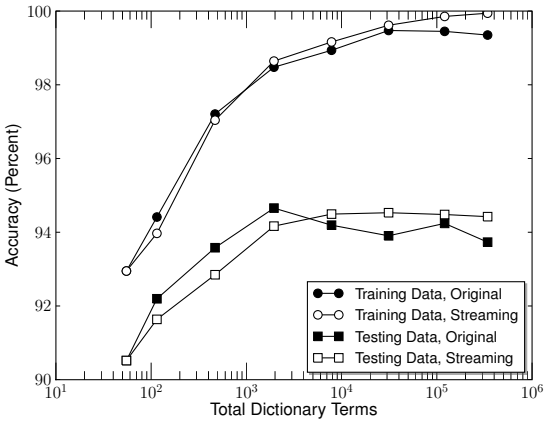

Fig. 3. Impact of number of terms on accuracy

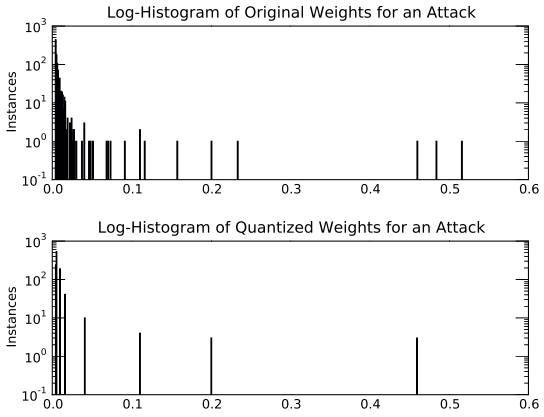

Fig. 4. Log histogram of $tfidf$ score values

As expected, both algorithms performed exceptionally well when evaluating the $training$ data set. Accuracy improved as more terms were included in the dictionary. In comparison, a loss of 2 to 5% in accuracy was observed in the $testing$ data set experiments. This drop is expected as the classifiers are evaluating data that was not available during training. The testing experiments plateau after two to eight thousand unique terms. This plateau indicates the point at which the classifiers begin to become overtrained.

In terms of the impact of the algorithm modifications on classifier quality, these experiments indicate that the original and streaming versions of the algorithm exhibit similar accuracy. Differences between the two algorithms can partially be attributed to a threshold operation in the classifiers that is used to fine tune filter sensitivity (see Section III-D for a description of this thresholding).

B. Quantize Term Scores

TFIDF training generates a large amount of statistical data that is encoded in a dictionary and utilized at runtime to determine the relevance of a document's terms to particular attack. This training typically exposes a small number of keywords that are assigned a high $categoryWeight$, while the majority of terms receive much lower values. The log histogram for one of the $categoryWeights$ in the ECML data set is illustrated in Figure 4 (upper). This histogram shows a great deal of numerical diversity (e.g., a few thousand unique data values for one category in the dictionary). However, is this diversity truly necessary for accurate classification? Our hypothesis is that it is not, given that our application may only need a gross estimate of a term's relevance (i.e., "high, medium, or low").

Based on this hypothesis, we constructed a program that resamples or $quantizes$ a dictionary's data values to a smaller number of unique data values. This approach employs a simple exemplar clustering algorithm that is weighted towards preserving larger data values. As illustrated in Figure 4 (lower), the number of unique data values is reduced to eight while maintaining a fair representation of the spectrum. Each of the nine vectors (idf and eight $categoryWeights$) in the dictionary is quantized individually. These data values are also transformed from a floating point representation to fixed point in order to simplify the hardware implementation.

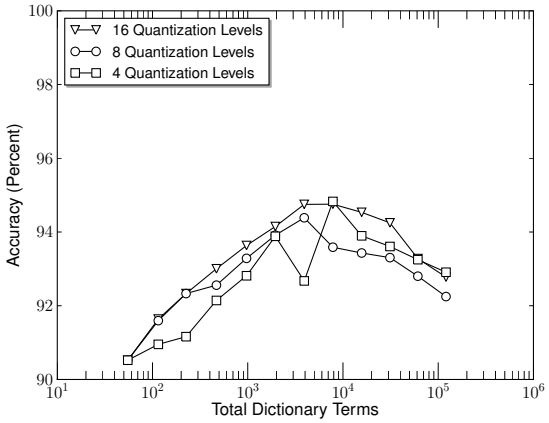

Fig. 5. Impact of quantization vs. term size on accuracy for competition data

In order to test our hypothesis we generated a wide range of truncated, quantized dictionaries from the training data set and evaluated accuracy when the classifier was applied to the testing data set. As illustrated Figure

5, reasonable accuracy can be achieved even when the dictionary is heavily quantized to contain just a few unique data values per vector. Over quantizing does result in instability and losses in accuracy. Based on this data set, we assert that a dictionary with roughly 4K terms and eight unique values per vector is sufficient.

C. Hash Methods

Terms in the original ECML training data set were on average 24 bytes long and in total were 19MB. Even with a reduced dictionary with only a few thousand terms, it is infeasible to store the original text in the dictionary. In addition to capacity issues, it is challenging to lookup an entry in the dictionary of this size with minimal memory accesses, necessitating hashing. A plain hash table for dictionaries of this size will still not likely fit entirely in an FPGA's internal Block RAM. It is necessary to consider more probabilistic hash functions that estimate whether an input belongs to a set. Bloom filters [2] are a common technique for compactly implementing a set membership test. A Bloom filter consists of several hash functions and a bit vector. All hash functions are applied to an input term and the resulting hash values index into the bit vector at multiple locations. The term is considered a member of the set if all selected bits are set. The Bloom filter is a probabilistic technique as collisions may result in false positives, although there will be no false negatives. The number of hash functions and size of the bit vector may be configured to optimize between memory constraints and desired accuracy. Reducing the false positive rate requires more memory for the filter.

Our approach to implementing a quantized dictionary is to employ a large array of Bloom filters, with each filter representing a particular data value in the dictionary. At runtime an incoming term is hashed according to the needs of the Bloom filters. The hashes are dispatched globally and each Bloom filter tests whether the input is a member of its set. If a Bloom filter identifies a hit, the data value associated with the filter is presented to the corresponding scoring unit. While this approach does not scale when there are a large number of quantization levels or attack types, it does provide a compact means of housing a dictionary with a large number of terms.

In our initial implementation, we focused on combining C_1, C_2, and C_3 to minimize the amount of data required by the dictionary. While this approach worked, it suffered in accuracy because of both false positive rates and the lack of numerical diversity. Instead, implementing two statistics, idf and $categoryWeight$, in the dictionary provides a larger numerical range (i.e., multiply two 8-value numbers) and can cause better Bloom filter accuracy (i.e., a false positive must occur in both the idf and $categoryWeight$ lookup to propagate).

D. Hardware Layout

The layout of the top-level hardware design is illustrated in Figure 6. This architecture has five components.

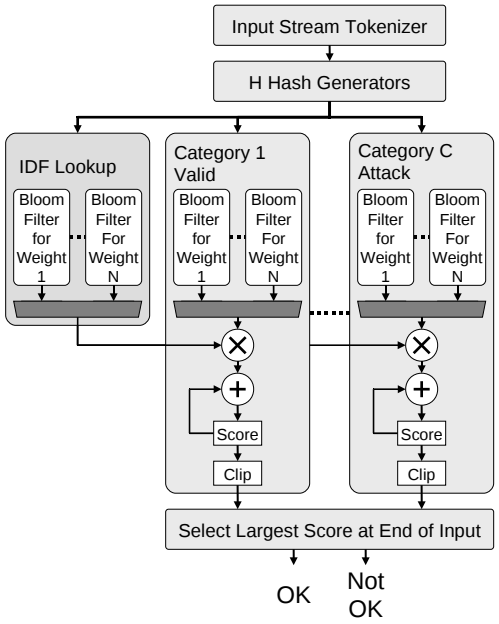

Fig. 6. Top level hardware design

Input Stream Tokenizer: The first unit parses an input message and generates a byte stream of lowercase tokens. This unit is the most complicated part of the design as tokens vary in length and can be delimited by multiple characters. Tokenization is a serial operation that operates on byte-sized data values and is therefore the bottleneck in the design.

Hash Generators: The second unit examines the incoming token byte stream and generates H different hashes for each token. A variety of hash functions were considered. We ultimately selected a Pearson [5] hashing approach that employs $4 \times H$ randomly-generated 256-entry lookup tables to hash each token. In order to avoid hash collisions between small tokens, we inserted a unit to append a token's bytestream with a 2-byte length. This unit adds two stall cycles per token to the byte stream, but greatly improves the quality of the hash functions.

IDF Lookup: A single set of Bloom filters is used to perform a dictionary lookup of the input term's idf value. If the term is not found in any of the Bloom filters, an output of zero is produced. The design only requires a single IDF Lookup Unit, as the idf value for an input token is the same for all categories.

Category Analysis Units: The bulk of the work in the design is performed by an array of category analysis units. Similar to the IDF Lookup unit, a category analysis unit employs an array of Bloom filters to lookup an input token's $categoryWeight$ for a particular attack type. This value is then multiplied by the idf value to compute the term's relevance, which is added to a cumulative score for the input message. When all tokens are processed, a threshold operation is applied to remove scores that do not meet a specified value. This threshold operation allows users to tune how sensitive the classifier is to malicious behavior.

Majority Vote: The last unit in the dataflow examines the final scores of the different categories when all tokens are processed and selects the category with the largest value as the winner. The message is labeled as "ok" or "not ok" based on whether the winning classifier is the "normal" category or an attack category.

E. Generating the Hardware Classifier

An important aspect of this work is being able to rapidly generate custom hardware designs based on different input training data sets and user-selected parameters. This feature is essential in network security applications where new attack vectors and categories are added on a regular basis. Our approach to making a customizable hardware implementation is based on two components. First, a general-purpose hardware design was developed that is parametrized and can be adapted to different classification work based on updates to the Bloom filter data. Second, we developed a tool chain for automatically building hardware. As illustrated in Figure 7, the tool flow is based on six steps:

1) **Training:** A TFIDF training program is used to analyze a labeled training data set and generate the full dictionary of term statistics. This data is exported to a SQLite database for later queires.
2) **Truncate:** For each classification category, the top T terms and their statistics are extracted. The user selects the parameter T.
3) **Quantize:** Each vector in the dictionary is run through a quantizer to reduce the number of unique data values in the dictionary. The number of quantization levels is a user-selected parameter chosen to trade between accuracy and memory footprint.
4) **Build Hashes:** Data from the modified dictionary is then converted into a series of Bloom filters. The user may tune a Bloom filter error rate parameter to scale the memory footprint of the filters and the number of number of hash functions that are utilized.
5) **Build Software:** The tool chain can export hash data to a C header file that is utilized by validation tools (e.g., verify all dictionary tokens hash properly) and a stand-alone evaluation program (e.g., classify all inputs in a file).
6) **Build Hardware:** Finally, the tool chain constructs a VHDL package file that includes all the data necessary to instantiate the Bloom filters.

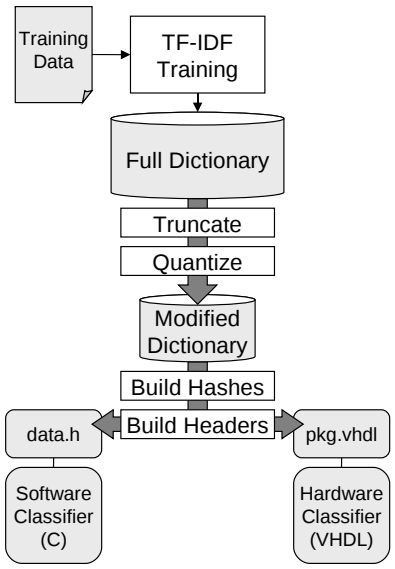

Fig. 7. Tool Flow for Building Hardware

While the current approach requires the hardware design be recompiled when a new model is applied, it would be straightforward to allow updates to be completed through writes to the Bloom Filter Block RAMs.

IV. Implementation Experiments

A number of experiments were conducted to validate both the hardware design and the tool chain. In all of the experiments we targeted a Xilinx Virtex 5-LX 50 part (XS5VLX50T-FG1136C-1) found on the Xilinx ML555 reference board. This part features sixty 36Kbit Block RAMs, allowing 240KB of 32b data values to be

stored internally. We employed the ISE 11.1 tools and the built-in synthesis tool XST. For verification, a special design was constructed that supplied a number of input documents to the classifier. ChipScope was utilized to verify the output results were correct.

A. Utilization Characteristics

In order to observe how different parameters affect the hardware implementation, we constructed a reference design that simply instantiates an input FIFO, the classification core, and routes all of the I/Os to the FPGA's pins. This design does not serve as a functional system, but provides a means by which realistic implementations can be observed. We supplied a large number of configurations and measured the amount of resources required by each implementation.

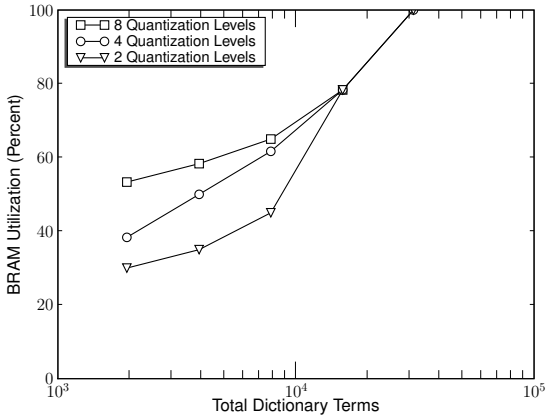

Fig. 8. Memory footprint for different build parameters

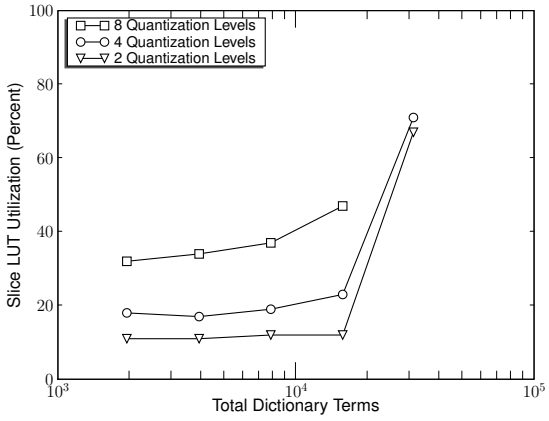

Fig. 9. Slice utilization for different build parameters

Resource utilization numbers are presented in Figures 8 and 9. As expected, more quantization levels translates to more Block RAM utilization in the smaller designs. However, these memory requirements become less distinct as more terms are included in the dictionary. This trait can be attributed to the fact that Block RAMs are allocated in large capacities (2KB), and that the lower-term dictionaries do not fully utilize their Block RAM allocations (e.g., a 128B Bloom filter is implemented with a 2KB BRAM). In terms of slice utilization, the different designs remain relatively constant until Block RAM resources are fully consumed. While the ISE tools are sophisticated enough to switch to using slices as memory when Block RAM is exhausted, doing so rapidly fills the FPGA.

TABLE I
THROUGHPUT COMPARISON

Classifier	Throughput
Original Java	2MB/s
Streaming C	34MB/s
Streaming HW	166MB/s

B. Performance Measurements

Performance of the hardware implementation depends on two factors: the average length of tokens in the input stream and the maximum rate at which the hardware can be clocked. For the former, each input contains a variable number of variable-length tokens. Our design processes data in a byte-stream manner and incurs two pipeline stalls at the end of each token encountered. For an input with C characters and T tokens, this delay results in a streaming efficiency of $\frac{C}{C+2 \cdot T}$. The design therefore has streaming efficiencies ranging from 0.5 in the worst case (when the input is a series of one character tokens) to nearly one in the best case (when the input is a single token). Inputs in the ECML testing data set were found to provide an average streaming efficiency of 0.85.

For clocking measurements, we generated a design that employed eight quantization levels and a dictionary with 3,919 terms of statistics. We found that the maximum clock rate for this design was 196MHz. Multiplying the streaming efficiency of the ECML testing data set by this clock rate results in a streaming rate of 166MB/s. This data rate is sufficient for Gigabit network speeds and greatly outpaces software implementations without impacting accuracy. Throughputs for different implementations are summarized in Table I. All three implementations have an accuracy of 94% in differentiating attack vs. normal.

V. Related Work

Reconfigurable computing researchers have utilized FPGAs in a variety of network security applications over the last decade. A significant amount of this work has focused on techniques for transforming the rules utilized by the well-known SNORT [10] network intrusion detection system into hardware that can be loaded into an FPGA. Hutchings et. al. [7] constructed tools that analyzed strings found in SNORT's ruleset and then transformed the resulting collection of regular expressions into synthesizable, string-matching hardware. Similarly, Gokhale et. al. [6] developed tools to convert SNORT's ruleset into content-addressable memory (CAM) tables that were then programmed into an FPGA that processed Gigabit Ethernet streams. While our work focuses on a different approach to classifying malicious traffic, it employs the same idea of using tools to translate a software application's data set to a custom hardware implementation that can be deployed in reconfigurable hardware.

In terms of architectural approaches, researchers have investigated a number of options for implementing SNORT rule checking in FPGA hardware [1]. These approaches typically compile rulesets into hardware in a way that leverages either the distributed memory resources of an FPGA or its ability to house large amounts of logic. Memory-based approaches include TCAMs [12], hash dictionaries [13], and parallel Bloom filters [4]. Logic-based approaches such as DFA or NFA [3] typically instantiate a large number of hardware state machines to compare many different patterns at the same time. Our approach is largely memory based as it utilizes many parallel Bloom filters to locate statistical weights for each input string.

VI. Conclusions

In this work, we have described an online, streaming, configurable hardware classifier capable of processing a text stream at 166MB/s. The classifier detects seven different attack types and differentiates between attack and normal HTTP web page requests with an accuracy of 94%. Optimizations were employed to enable streaming, reduce computation, and minimize memory usage. Even with a two order of magnitude reduction in the size of the dictionary, the hardware algorithm shows the same accuracy as the original software implementation. The sequential Java implementation has a throughput of 2MB/s, and a C implementation of the streaming algorithm has throughput of 34MB/s with the same accuracy. The performance of the hardware classifier allows it to be used as a real-time analysis component of an advanced intrusion prevention pipeline in network security applications.

Acknowledgments Brian Gallagher and Tina Eliassi-Rad invented the document similarity method of HTTP request classification and developed the software classifier. We are indebted to them for discussions concerning the classifier and for providing the sequential Java version. The ECML/PKDD data set was obtained from the ECML/PKDD 2007 Workshop and is administered by Dr. Mathieu Roche.

References

[1] T. AbuHmed, A. Mohaisen, and D. Nyang. A Survey on Deep Packet Inspection for Intrusion Detection Systems. *ArXiv e-prints*, Feb. 2008.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high speed networks. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–257, 2004.

[4] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *IEEE Micro*, pages 44–51. IEEE Computer Society Press, 2003.

[5] B. Gallagher and T. Eliassi-Rad. Classification of http attacks: A study on the ecml/pkdd 2007 discovery challenge. (TR-414570), 2009.

[6] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, pages 404–413, 2002.

[7] B. L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 111, 2002.

[8] Ludovic Denoyer and Hung Son Nguyen. *ECML/PKDD 2007 Discovery Challenge*. Available http://www.ecmlpkdd.org/, 2007.

[9] D. L. Olson and D. Delen. *Advanced Data Mining Techniques*. Springer, 2008.

[10] M. Roesch and S. Telecommunications. Snort - lightweight intrusion detection for networks. pages 229–238, 1999.

[11] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, (11):613–620, 1975.

[12] H. Song and J. W. Lockwood. Efficient packet classification for network intrusion detection using fpga. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 238–245, New York, NY, USA, 2005. ACM.

[13] S. Yoon, B. Kim, and J. Oh. High-performance stateful intrusion detection system. In *IEEE Computational Intelligence and Security*, 2006.