The Energy Cost of Network Security: A Hardware vs. Software Comparison

André L. França*, Ricardo Jasinski*, Paulo Cemin*, Volnei A. Pedroni* and Altair Olivo Santin†
{a.l.franca, jasinski, p.r.c}@ieee.org, pedroni@utfpr.edu.br, santin@ppgia.pucpr.br
*Federal Technological University of Parana (UTFPR), Curitiba, PR, Brazil
†Pontifical Catholic University of Parana (PUCPR), Curitiba, PR, Brazil

***Abstract*—The increasing network speeds, number of attacks, and need for energy efficiency are pushing software-based network security to the limit. A common kind of threat is probing attacks, in which an attacker tries to find vulnerabilities by sending many probe packets to a target machine. In this paper, we evaluate three machine learning classifiers (Decision Tree, Naive Bayes, and k-Nearest Neighbors), implemented in hardware and software, for the detection of probing attacks. We present detailed results showing the tradeoffs between energy consumption, throughput, and accuracy of the three classifiers. The fastest hardware implementation is 926 times as fast as its software counterpart, and its energy consumption per classification is 0.05% that of the software version.**

Keywords—Network Security; Probing Attack; Decision Tree; Naive Bayes; k-Nearest Neighbors; Energy Efficiency

I. Introduction

Protection against threats is a key part of any network-enabled device. In the first semester of 2014, McAfee reported 229 million malware samples, with 3.9 million targeting mobile devices [1]. Such devices are typically battery-powered, and must provide effective network protection with minimum energy consumption. Moving protection algorithms from software (SW) to hardware (HW) is a promising approach to achieve both goals.

Probing attacks are a common kind of threat. An attacker sends probe packets to a target machine to find open ports, identify services, and exploit vulnerabilities. Probing attacks are carried out using software tools such as network sniffers and port scanners [2]. Because probing often precedes other kinds of attacks, detecting attempts at this early stage can prevent later intrusions. Network attacks can be detected by creating a prediction model of the attack behavior, and then testing all network traffic using this model [3].

This paper evaluates three Machine Learning (ML) classifiers designed for the detection of probing attacks: Decision Tree (DT), Naive Bayes (NB) and k-Nearest Neighbors (kNN). These classifiers were chosen among others because of their wide use and, at the same, relatively low but increasing implementation complexity. The algorithms were implemented in SW and HW to compare the energy efficiency of the two approaches. The HW and SW versions implement the same algorithms and exhibit exactly the same classification behavior. Besides the energy consumption, we have also investigated throughput and accuracy results for the three classifiers.

This work has been supported in part by the Intel Lab's University Research Office and by the Brazilian National Council for Scientific and Technological Development (CNPq).

II. Machine Learning for Network Security

ML techniques can classify an input sample based on similar records from a dataset. Functions that map input data to one of possible classes are called classifiers. In a network security application, a classifier can label incoming traffic as normal or attack, using feature values (attributes) extracted from network packets and communication history [4]. The training stage of a classifier uses a dataset and an ML algorithm to produce the attack model. The classification stage extracts features in real-time and uses them as inputs for the classifier.

To develop our classifiers we created a dataset, using an audit tool (Nessus) to produce probing attacks, and workload tools to generate normal traffic. Table I shows the features selected from this dataset and used to create the classifiers. The features include values obtained directly from the packet header as well as communication state variables maintained by the detector engine. An example of a feature extracted from the header is the IP "*Don't Fragment*" flag (*ip_DF*). Features obtained from the communication between hosts are counters that depend on the packet direction. In Table I, the suffix *c2s* indicates the client to server direction, and *s2c*, the opposite direction. For instance, feature *total_bytes_c2s* counts the total number of bytes sent from the client to the server.

TABLE I. FEATURES USED IN THE CLASSIFIERS DEVELOPMENT

Category	Features
Protocol header	ip_length, ip_id, protocol_type, ip_checksum, tcp_ack, frame_length
Port numbers	udp_sport, udp_dport, tcp_sport, tcp_dport
IP/TCP Flags	ip_DF, ip_ MF, tcp_frst, tcp_fpush, tcp_fack
Frame counters	total_frames_s2c, service_frames_s2c
Byte counters	total_bytes_s2c, service_bytes_c2s, service_bytes_s2c
Flag counters	fin_c2s, syn_fin_s2c, ack_c2s, ack_s2c, syn_c2s, syn_s2c, rst_c2s, rst_s2c
Connection state	conn_status

A. Decision Tree Classifier

The DT is implemented as sets of *if-then* statements that classify a sample by traversing a tree until a leaf associated with a class is reached. Each branch tests an attribute of the input sample, and each leaf corresponds to a possible outcome [5]. Fig. 1 shows part of our DT model. The *ip_DF* attribute is tested first. If it is 0, the attribute *service_frames_s2c* is tested next. If it is greater than 6, the classifier indicates that this is a normal packet. The DT classifier performs only simple computations, so it is not computationally intensive.

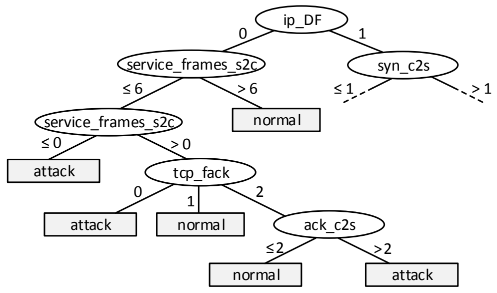

Fig. 1. Partial view of the Decision Tree classifier.

B. *Naive Bayes Classifier*

The NB is a probability-based algorithm that assumes the features are conditionally independent. Thus, the probability of an instance belonging to a class depends on the probabilities of each feature inferring that class [5]. The prior probability of each class, which does not depend on the features, is multiplied by the individual feature probabilities. The class with the greatest probability is chosen as the output (see Algorithm 1).

An advantage of NB is that the probabilities are constant for each attribute range, and in a table-based implementation it may be easily updated without changing the classifier structure.

Algorithm 1 Naive Bayes Classifier

```
1:  $P_{NORMAL} \leftarrow P_{NORMAL\_PRIOR}$
2:  $P_{ATTACK} \leftarrow P_{ATTACK\_PRIOR}$
3:  for each attribute attr(i)
4:      $P_{NORMAL}$_attr ← table_lookup( TABLE_$P_{NORMAL}$, attr(i) )
5:      $P_{ATTACK}$_attr ← table_lookup( TABLE_$P_{ATTACK}$, attr(i) )
6:      $P_{NORMAL} \leftarrow P_{NORMAL} \times P_{NORMAL}$_attr
7:      $P_{ATTACK} \leftarrow P_{ATTACK} \times P_{ATTACK}$_attr
8:  end for
9:  if $P_{ATTACK} \geq P_{NORMAL}$ then class ← attack
10:                        else class ← normal
11: end if
```

C. *k-Nearest Neighbors Classifier*

The kNN classifier works directly with examples stored in memory, meaning that it does not require a model. An input packet is classified based on similarity with the training samples [5]. The kNN pseudocode is shown in Algorithm 2. First, all attributes are normalized in the range -1 to +1 to fit the same scale. Next, it calculates the Euclidean distance (or its square) from the packet to each of the training samples. Then, the *k* nearest samples are selected. If the majority of the neighbors are attacks, the classifier output indicates an attack.

The kNN algorithm is appropriate for attack detection because it provides a nonlinear boundary between normal and attack classes. The classifier is also easy to update, since its training instances may be stored in a volatile memory.

Algorithm 2 k-Nearest Neighbors Classifier

```
1: input_sample ← normalize (input_sample)
2: for each training_sample
3:     distance = $\sum_i$(training_sample_attr(i) − input_sample_attr(i))$^2$
4:     if distance < nearest_neighbors.distance.max
5:         insert(nearest_neighbors,[training_sample.distance])
6:     end if
7: end for
8: class ← majority(nearest_neighbors, classes)
```

III. Implementation

A. *Classifiers in Software*

The SW implementations of the classifiers are direct translations (in C++) of the algorithms described in the previous section. They were all tested using the same input vectors as the HW versions (the latter were implemented using VHDL).

B. *Decision Tree Classifier in Hardware*

The DT classifier (Fig. 2) is a straight implementation of the algorithm of Fig. 1, using 32-bit unsigned values. The comparators check the range of the features. Then, the outputs are combined in a sum of products that will be high when the packet is classified as an attack. The circuit is entirely combinational [6].

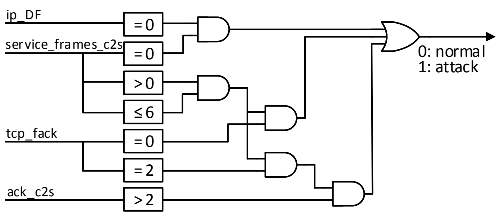

Fig. 2. Hardware implementation of the DT (~1/6 of the circuit shown).

C. *Naive Bayes Classifier in Hardware*

We used two versions of the NB classifier. The combinational version (Fig. 3a) is a straight implementation of the Algorithm 1. The used model has 10 attributes. For each attribute, two table lookups are performed: one to get the probability that the attribute denotes an attack, and another for the probability of a normal packet. This implies two floating-point multiplications of 10 factors each. The larger probability value determines whether the packet is normal or an attack.

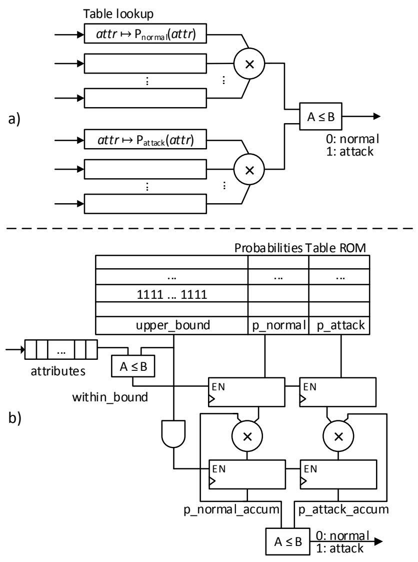

Fig. 3. Hardware implementation of the NB: a) combinational; b) sequential.

The sequential version (Fig. 3b) serializes the table lookup for the probability values and calculates the probabilities one attribute at a time. As a result, it uses only two 2-input multipliers, but it takes 73 clock cycles to classify one vector. The classification process starts when the features are loaded into the *attributes* register. For each attribute, the probabilities table is read one row at a time, from bottom to top. The table lookup consists in comparing the attribute value with the upper bound in the table. While the attribute is within this bound, the table output registers are overwritten with the corresponding P_{normal} and P_{attack} values. The last upper bound value for each attribute is a vector with all bits set to one, which is used to detect the end of an attribute. At this point, the attribute probability values are multiplied by the accumulated values. When the probabilities of all 10 attributes have been multiplied, a comparator decides whether the input packet is normal or an attack.

D. *k-Nearest Neighbors Classifier in Hardware*

The kNN classifier (Fig. 4) keeps 1,000 training samples in a ROM. It works iteratively, calculating the distance from each training sample to the input packet. The circuit keeps the three closest distance values (k = 3) and the corresponding class labels. After all distances have been calculated, the label with the most occurrences is selected as the output.

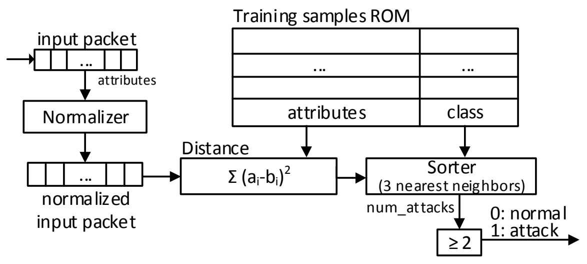

Fig. 4. Hardware implementation of the kNN classifier.

E. *Measurement Circuit and Setup*

To evaluate the HW classifiers in operation and to measure their power consumption, we created a test circuit (Fig. 5) composed of: a configurable number of instances of any classifier; a ROM with 2,000 vectors and the expected class (each vector has 50 attributes and is 1601 bits wide); a PLL to select the operating frequency; a FIFO memory, to read the test vectors from the ROM, and to use them as inputs for the classifiers; and an error detector that signals when a vector was incorrectly classified. The vectors are read continuously from the ROM.

The HW and SW implementations were evaluated in a Cyclone IV GX FPGA development kit and in a DN2800MT Atom motherboard, respectively. To measure the FPGA's power consumption, we used the Power Monitor tool from Altera. This tool measures the FPGA's current consumption and sends the results continuously to a PC via JTAG. To measure the motherboard's power consumption, we used a 6½-digit multimeter controlled by a script that initiates and stops the sampling as the SW classifiers are executed. We used the Linux *time* command to obtain the processing time.

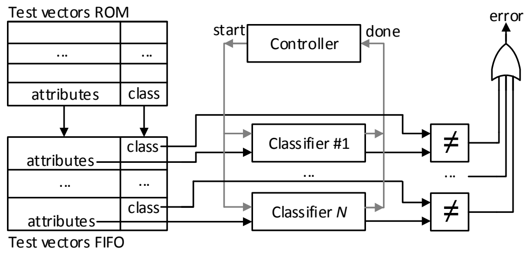

Fig. 5. Power measurement circuit for the HW classifiers.

IV. Results

A. *Circuit Area (HW)*

Table II presents the area used by the classifiers. The implementations are named with a prefix (*dt*, *nb*, or *knn*) denoting the algorithm. The NB classifiers include an infix (*comb* or *seq*) denoting the combinational or sequential version. The classifiers that use floating-point numbers include a suffix (32, 16, or 10) indicating the number of bits of the floating-point representation. The DT is the most compact of all implemented classifiers, occupying only 167 logic cells (LCs). The combinational version of the NB classifier uses the most area, with 11,488 LCs. The kNN classifier requires the most memory bits, used to store the training samples used in the algorithm.

TABLE II. Circuit Area of the Implemented Classifiers

Algorithm	Classifier Implementation	Logic Cells	Memory Bits	9-bit Multipliers
DT	dt	167	0	0
NB	nb_comb_32	11,488	0	126
	nb_comb_16	5,480	0	36
	nb_comb_10	2,867	0	17
	nb_seq_32	1,733	10,112	14
	nb_seq_16	1,000	6,656	4
	nb_seq_10	705	5,120	2
KNN	knn_32	7,489	464,896	14
	knn_16	4,074	229,376	4
	knn_10	2,641	125,952	0

B. *Throughput (HW vs. SW)*

Table III presents the throughput of the classifiers. For the combinational ones (*dt* and *nb_comb*), the throughput was calculated from the propagation delay between the attributes input and the class output. For the sequential ones (*nb_seq* and *knn*), the throughput is based on the maximum operating frequency and the number of cycles to classify one packet. The last column in the table shows the throughput of the SW classifiers. For DT and NB, the HW version is faster (by a factor of 926, 37.4, and 6.1 for *dt*, *nb_comb*, and *nb_seq,* respectively). For kNN, the SW version is faster. The main reason is the large number of clock cycles due to the number of attributes (19) and training samples (1,000), which are multiplied by each other. If we compare the throughput values with the maximum theoretical limit of a Fast Ethernet link, 150,000 packets/s [7], we can see in the rightmost column that the software classifiers are unable to reach this value. On the other hand, in hardware, all but the kNN classifier easily exceed this limit.

TABLE III. Throughput of the Implemented Classifiers

Classifier	# Cycles	Fmax (MHz)	Classification Time (ns)	#Packets/s (HW)	#Packets/s (SW)
dt	-	-	13.7	72,865,054	78,684
nb_comb_32	-	-	357.8	2,794,811	74,649
nb_comb_16	-	-	240.5	4,157,295	-
nb_comb_10	-	-	181.6	5,507,942	-
nb_seq_32	73	33.46	2,181.7	458,356	74,649
nb_seq_16	73	56.12	1,300.8	768,767	-
nb_seq_10	73	71.70	1,018.1	982,192	-
knn_32	24,020	14.49	1,657,695	603	7,246
knn_16	24,020	18.60	1,291,397	774	-
knn_10	24,020	26.56	904,367	1,106	-

C. *Energy Consumption (HW vs. SW)*

Fig. 6 shows the power consumption of the measurement circuit itself vs. the number of instances of the HW classifiers, at 25 MHz. Because the consumption increases linearly with the number of classifiers, we obtained the power consumption of a single instance from the angular coefficient of each trend line.

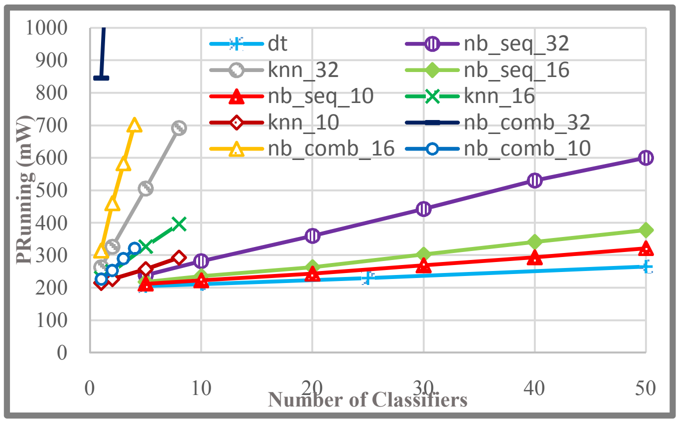

Fig. 6. Measurement circuit's consumption vs. #HW classifiers at 25 MHz.

Table IV summarizes the energy consumption and relative accuracy of the classifiers. $P_{Running}$ is the average power consumption of a single HW classifier, obtained by linear regression from Fig. 6. P_{Idle} is the power consumption of a single HW classifier while it is disabled. To obtain the energy per classification in HW, we calculated the weighted average of the energy spent during idle and running intervals, considering the time needed for classification and a throughput chosen within the limits of each classifier.

TABLE IV. ENERGY CONSUMPTION OF THE IMPLEMENTED CLASSIFIERS

Classifier	P_{idle} (mW)	$P_{running}$ (mW)	HW Energy per op. (nJ) [a]	Relative Accuracy (%)	SW Energy per op. (nJ)
dt	0.71	1.35	4.76	100.00	8,929.4
nb_comb_32	1.93	583.79	36.14	100.00	8,954.1
nb_comb_16	1.86	128.68	17.47	99.66	-
nb_comb_10	1.29	31.87	9.82	97.51	-
nb_seq_32	2.06	8.10	31.37	100.00	8,954.1
nb_seq_16	1.57	3.55	16.25	99.66	-
nb_seq_10	1.26	2.42	11.79	97.51	-
knn_32	53.26	61.01	113,966.2	100.00	76,099.0
knn_16	18.46	24.07	42,310.1	99.97	-
knn_10	10.13	11.04	21,134.3	98.47	-

a. Throughput of 150,000 packets/s for dt and nb and 500 packets/s for knn.

To obtain the energy consumption in SW, we multiplied the application running time by the difference in power consumption when the classifier is running and when the OS is not running any user application [8]. We subtracted this baseline consumption to allow a fair comparison between the HW measurements, which consider only the FPGA chip, and the SW measurements, which consider the motherboard. The DT in HW is the most energy-efficient of all implemented classifiers, spending 4.76 nJ to classify one vector – only 0.05% of the SW version. The relative accuracy indicates the percentage of operations performed by the HW classifier that match the output of the corresponding SW classifier. For the 32-bit float versions, the relative accuracy is 100% because HW and SW produce the same results. As we reduce the number of bits in HW, the accuracy is also reduced. The accuracy loss was less than 2.5% in all cases, but the energy savings obtained by reducing the floating-point size from 32 to 10 bits were 73%, 62%, and 81% for *nb_comb*, *nb_seq*, and *knn*, respectively.

Fig. 7 shows side-by-side the energy to classify one packet by each classifier. The difference between the best HW and SW implementations (*dt_hw* and *dt_sw*) is greater than three orders of magnitude.

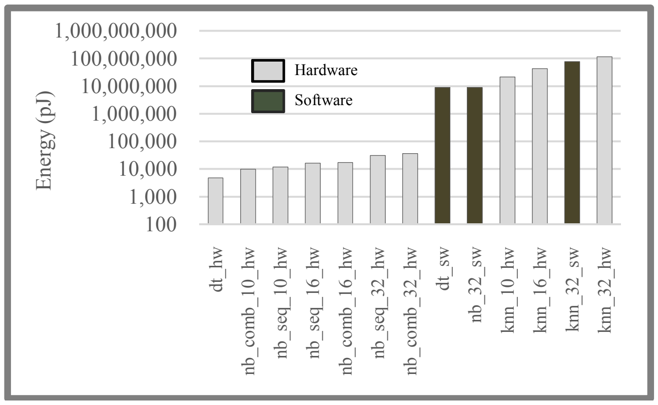

Fig. 7. Energy spent to classify one packet, HW vs. SW.

V. Conclusion and Future Work

We have implemented and evaluated three machine learning classifiers (Decision Tree, Naive Bayes, and kNN) aimed at detecting probing attacks in HW and SW. We presented a comparison of energy consumption, throughput and accuracy for the three algorithms. The fastest HW implementation, the Decision Tree, is 926 times as fast as its SW counterpart, and its energy consumption per classification is 0.05% that of the SW version. Future work includes the development of Support Vector Machine (SVM) and Linear Discriminant Analysis (LDA) classifiers and implementation of the feature extraction engine in hardware.

References

[1] McAfee Labs, "Threats Report", McAfee, Tech. Rep., June 2014.

[2] S. Pukkawanna, H. Hazeyama, Y. Kadobayashi and S. Yamaguchi, "Investigating the Utility of S-Transform for Detecting Denial-of-Service and Probe Attacks", IEEE International Conference on Information Networking, 2014.

[3] C.-F. Tsai, Y.-F. Hsu, C.-Y. Lin, and W.-Y. Lin, "Intrusion detection by machine learning: A review", Expert Systems with Applications, vol. 36, n. 10, pp. 11994-12000, 2009.

[4] A. Das, D. Nguyen, J. Zambreno, G. Memik, and A. Choudhary, "An FPGA-Based Network Intrusion Detection Architecture", IEEE Transactions on Information Forensics and Security, vol. 3, n. 1, pp. 118-132, 2008.

[5] S. Sharma, J. Agrawal, S. Agarwal and S. Sharma, "Machine Learning Techniques for Data Mining: A Survey", IEEE International Conference on Computational Intelligence and Computing Research, 2013.

[6] A. L. França, R. Jasinski, V. A. Pedroni and A. O. Santin, "Moving Network Protection from Software to Hardware: an Energy Efficiency Analysis", IEEE Computer Society Annual Symposium on VLSI, 2014.

[7] Cisco, "Bandwidth, Packets Per Second, and Other Network Performance Metrics", Cisco, Tech. Rep., 2009.

[8] K. Tsoi and W. Luk, "Power Profiling and Optimization for Heterogeneous Multi-Core Systems", ACM SIGARCH Computer Architecture News, vol. 39, n. 4, pp. 8-13, 2011.