Moving Network Protection from Software to Hardware: an Energy Efficiency Analysis

André França, Ricardo Jasinski, Volnei Pedroni
Federal University of Technology - UTFPR
Curitiba, Brazil
{*a.l.franca, jasinski*}*@ieee.org, pedroni@utfpr.edu.br*

Altair Olivo Santin
Pontifical Catholic University of Parana - PUCPR
Curitiba, Brazil
santin@ppgia.pucpr.br

***Abstract*—Software-based network security is constantly challenged by the increase in network speeds and number of attacks. At the same time, mobile network access underscores the need for energy efficiency. In this paper, we present a new way to improve the throughput and to reduce the energy consumption of an anomaly-based intrusion detection system for probing attacks. Our framework implements the same classifier algorithm in software (C++) and in hardware (synthesizable VHDL), and then compares the energy efficiency of the two approaches. Our results for a decision tree classifier show that the hardware version consumed only 0.03% of the energy used by the same algorithm in software, even though the hardware version operates with a throughput that is 15 times that of the software version.**

***Keywords*-intrusion detection system; anomaly-based detection; decision tree classifier; energy efficiency; FPGA.**

I. Introduction

Today's networking systems face two major concerns: throughput and energy efficiency. Cisco estimates the total internet throughput at 167 terabits per second, and growing at 29% a month [1]. At the same time, computing already accounts for 6% of worldwide electricity consumption [2], and mobile devices reinforce the need for making the most of the available energy.

While the network usage of an application can be measured with great precision, this is not true for energy consumption. At the current state of the art, there are very few tools to help build energy-efficient algorithms; one such example is the LEAP platform [3], which allows measuring the power consumption of certain elements (e.g., CPU, hard disk, and RAM) in an Intel Atom motherboard.

A promising approach to improve the throughput and energy efficiency of networking systems is to move basic algorithms from software to hardware. Using dedicated circuits, computations that require many instructions in software can be performed in a single clock cycle in hardware. Because there is no need to support general-purpose algorithms, the resulting hardware is leaner and operates at a fraction of the power consumption of a generic processor.

This paper describes a novel approach to move anomaly detection algorithms from software to hardware, assessing the impact on power consumption. Our end goal is to offload most network security tasks from the system CPU to a coprocessor or network card. Currently, we have implemented a decision tree algorithm to detect probe attacks, in both software and hardware. The two implementations are functionally identical; however, the hardware solution has a significantly higher throughput, and an energy cost that is orders of magnitude lower than the software approach.

II. Background

A. Anomaly-Based Intrusion Detection Systems

An intrusion detection system (IDS) monitors events in a network or system, and decides whether each event is legitimate or unauthorized [4] [5]. There are two basic kinds of IDS: signature-based and anomaly-based.

In signature-based IDS, the system maintains a list of known misuse patterns, and monitors network events in search of those patterns. One advantage of signature-based systems is that known threats can be detected efficiently and with a very low false-positive rate [5]. On the other hand, as the number of misuse patterns increases, the list may grow too large and some new patterns might be not included.

Anomaly-based IDS consists in creating a model of normal or anomalous behavior, and classifying each network event into these categories [5]. An advantage of such systems is that they can detect previously unknown attacks. On the other hand, they are susceptible to producing false alarms, and the model needs to be updated when there are changes in the definition of normal or attack behavior. Despite these disadvantages, anomaly detection is an alternative for IDS because the network is always exposed to new threats. The challenge lies in designing a classifier with a high accuracy and low false alarm rate.

B. Machine Learning for Anomaly Detection

One way to detect intrusions in the anomaly-based approach is to build a prediction model of what constitutes attack behavior, and then to test all network traffic using this model [6]. If the model predicts that the analyzed traffic is a threat, the detector may warn the user or host system.

Machine Learning (ML) techniques are commonly used to discover underlying models from a set of training data. Such

techniques are known for their adaptability, fault tolerance, and resilience against noise (information that confuses the classifier) [5]. They are also well suited for learning tasks where there is no a priori knowledge of the underlying patterns [7]. ML applications are commonly built around classifiers, which are functions that map data points to class labels. An IDS attempts to classify all traffic as either normal or attack [7].

The creation and use of a classifier are often two separate stages. The first stage, or training, uses a training dataset and a model generation algorithm to choose the features to be examined in the input data, and produces the attack model. In intrusion detection, the features are values obtained from packet data or network activity, and the possible output values are normal traffic or attack. This stage can be computed offline, and is often computationally intensive [5].

The second stage, executed in real-time, is classification. In this phase, the chosen features are extracted from an actual input vector, and the corresponding values are applied to the classifier. For intrusion detection, the mostly used classifiers are Support Vector Machines (SVM), Decision Tree, and K-Nearest Neighbors (KNN) [6].

C. Datasets for Intrusion Detection

The dataset used in the training phase is fundamental to the creation of a good attack model. The training algorithm iterates over a large number of input vectors, which must be previously labeled as either normal or attacks.

There has been intense research in creating representative datasets for security applications. The publicly available NSL-KDD dataset [8] is an improved version of KDD 99, and it is aimed at detecting four types of attacks: denial of service (DoS), probing, remote-to-local (R2L), and user-to-root (U2R).

D. Decision Tree Classifiers

Decision Trees (DTs) are appropriate for intrusion detection for several reasons: the input data can be discrete or continuous; it is not computationally intensive; and it can be easily implemented in real-time systems [9].

The basic principle behind DTs is a successive partitioning algorithm, which partitions the original dataset into nodes [10]; the C4.5 algorithm is a common choice for constructing the model [11]. DTs are implemented as sets of if-then rules, and they classify each input by traversing the tree from the root until it reaches a leaf node, with which a class label is associated. Each non-terminal node specifies a test of an attribute, and the branches correspond to the possible attribute values [11].

E. Implementation of Security Algorithms in Hardware

Machine learning techniques have shown limitations to achieve at the same time a high detection accuracy and fast processing time. Because of the increasing network speeds and number of attacks, the execution of security solutions purely in software is reaching its limit, and the sequential execution imposed by software can prevent real-time analysis at today's network speeds [5]. The inherent parallelism provided by hardware makes such algorithms ideal candidates for a hardware implementation.

Another reason to use hardware is immunity from software infections. Hardware circuits are very hard to modify unintentionally, and can be isolated from the software environment. On the other hand, an IDS implemented in hardware loses the software flexibility, so it must provide mechanisms for reconfigurability [12]. A feasible alternative for hardware updates is to use an FPGA, whose reconfiguration can be made as hard (to prevent malicious changes) or as easy as desired (to allow for end-user updates).

Systems-on-a-Chip (SoC) are another promising field for IDS. SoCs connected to the internet should total 50 billion by 2020 [13]. Because SoCs are associated with mobility and battery power, energy efficiency is a topic of concern.

F. Power Measurement and Estimation

Despite the great interest in energy efficiency, there are few tools to analyze the power consumption of individual algorithms or applications. One of the first platforms for energy measurement of an algorithm is LEAP [3], developed at UCLA's Laboratory for Advanced Systems Research.

The LEAP platform measures the power consumption of sections of code running in an Intel Atom motherboard. It uses the CPU timestamp counter and an external analog-to-digital converter to measure the power consumption at specific points in the motherboard. The system reports the energy consumed in individual modules such as CPU, RAM, hard disk, north bridge, USB, and power supply [3].

As for FPGAs, there are tools for both power estimation and power monitoring. Altera, for instance, provides the PowerPlay and Power Monitor tools. PowerPlay calculates the total power consumption of a circuit by summing up the estimates of each circuit element, considering the element's capacitance model and switching activity. However, this estimate may be off by up to 32%, or even 209% in similar tools from other manufacturers [14]. An alternative to estimation is to measure the actual consumption of the FPGA, using the Power Monitor tool. However, this tool requires a circuit board instrumented with several current sensors and analog-to-digital converters.

III. METHODOLOGY

A. General Workflow

In order to create the hardware and software for probing detection, we developed an automated process that starts with a dataset specification, and produces two classifier implementations, one in C++ and the other in VHDL. Fig. 1 shows the general workflow of our approach. This section describes the steps involved in this process.

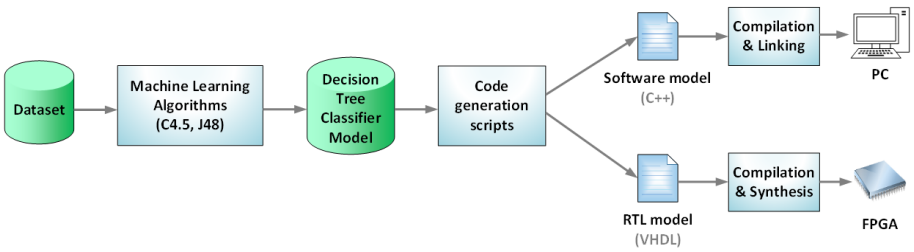

Figure 1. General workflow, from dataset specification to hardware and software implementations.

1) Dataset: The first step in creating the attack model is to choose a dataset. In our work, we have used two datasets: the publicly available NSL-KDD dataset, and a custom dataset aimed at detecting probe attacks.

The NSL-KDD dataset was used primarily to test our workflow and to develop our tools and methodology. This dataset uses 41 attributes, such as connection duration, protocol type, service type, and number of data bytes.

Our custom dataset was used in all of our energy measurements. It was created using an audit tool (Nessus) to produce probing attacks, and workload tools to generate different types of normal traffic (SSH, SNMP, IMAP, and HTTP). The resulting dataset has 66 attributes. The records were split in a training dataset (with 150,189 instances) and a test dataset (with 37,548 instances). Both datasets have a similar proportion of normal and attack traffic records.

2) Machine Learning Algorithms: For the training, evaluation, and test of our models, we used the Weka framework, an open-source collection of machine learning algorithms for data mining [15]. To generate the trees, we used the J48 algorithm, a Java implementation of the C4.5 DT algorithm.

To build our first DT, based on the NSL-KDD dataset, we used the partial training set (with 20% of the instances), and nine manually chosen attributes: protocol_type, flag, is_guest_login, srv_count, srv_serror_rate, rerror_rate, srv_rerror_rate, diff_srv_rate, srv_diff_host_rate. We chose a small confidence factor to reduce overfitting of the model on training data and to obtain a smaller tree. The resulting tree had 18 internal nodes and 40 leaves.

To build our second DT, which was based on our custom dataset, we used a genetic algorithm to select the 12 most relevant attributes. We also used the J48 algorithm with the standard value of 0.25 for the confidence factor. The resulting tree had 68 internal nodes and 70 leaves.

3) Attack Model: The output of the J48 algorithm is a model, which includes a list of the attributes used in the classification and a tree structure (Fig. 2).

4) Code generation scripts: To support the translation of our model into hardware circuits, we created a framework of VHDL code suitable for both simulation and synthesis. The simulation code ensures that the hardware and software versions have exactly the same behavior and accuracy. The synthesis code can be directly synthesized into an FPGA.

```
=== Classifier model (full training set) ===

J48 pruned tree
------------------
protocol_type = tcp
|   ip_len <= 104
|   |   tcp_fpush = 0: attack
|   |   tcp_fpush = 1
|   |   |   tcp_ffyn = 0: normal
|   |   |   tcp_ffyn = 1
|   |   |   |   tcp_ack <= 2825023790
|   |   |   |   |   tcp_seq <= 400185759: normal
|   |   |   |   |   tcp_seq > 400185759: attack
|   ip_len > 104
|   |   flag = REJ: normal
|   |   flag = RSTR: anomaly
|   ...
protocol_type = udp
|   ...
```

Figure 2. Example of a tree model output by the J48 algorithm.

Our custom scripts include a code generator which parses the model description obtained from Weka and outputs code in VHDL and C++. We use the VHDL code to synthesize the classifier hardware, and the C++ code for execution in an x86 CPU. This allows us to compare the two implementations of the same decision tree. Fig. 1 illustrates this process, with our custom script in the center of the diagram.

B. Classifier Implementation in Software

We compiled the generated C++ code on a desktop workstation and uploaded it to a separate Intel DN2800MT desktop board, which runs an Atom CPU at 1.8 GHz.

Our tests used a subset of 2,000 input vectors (1,000 attack vectors and 1,000 normal vectors) out of the total 37,548 entries. We limited the number of vectors in order to use the same entries in software and hardware, and this was the number possible with the RAM available in the FPGA. Our test program initially reads the input vectors from the disk and stores them in RAM. Next, it applies all input vectors to the classifier routine, repeating the whole process 100,000 times. This gives a total $2{\cdot}10^8$ classifying operations and a running time of approximately 15 seconds.

C. Classifier Implementation in Hardware

To evaluate the DT circuit in operation and to measure its power consumption, we created a test circuit composed of:

a) A configurable number of instances of the classifier circuit. Because the power consumption of a single classifier

was too small to measure, most of the experiments were run with 100 classifiers operating simultaneously (Fig. 3-d).

b) A ROM containing 2,000 vectors (the same vectors from software implementation) from our dataset (Fig. 3-b). Each vector has 12 attributes and is 218 bits wide.

c) Clock management (PLL) and timing circuits, to select the operating frequency and the number of vectors/second.

d) Additional circuitry, including a FIFO memory, to read the input vectors from the ROM, and to use them as inputs for the classifier instances (Fig. 3-c).

Because it was not possible to measure the power consumption of a single classifier instance, we designed a FIFO to store the input vectors and to apply them to 100 copies of the classifier circuit. The address increment circuit, the ROM memory, and the FIFO are clocked circuits; the DT classifiers are combinational circuits.

The test circuit implemented in the FPGA has the following characteristics: configurable number of decision tree classifiers (1 to 100), configurable operating frequency via a PLL (50, 100, 150, or 200 MHz), and configurable number of vectors classified per second (one operation every 1, 2, 5, or 10 clock cycles). The test vectors stored in the ROM are read continuously, repeating the test set every 2,000 vectors.

IV. Results

A. Decision Tree Accuracy

We used the training and test datasets of NSL-KDD to verify the accuracy of our first decision tree. The number of correct and incorrect classifications is shown in Table I.

Our tests showed that the hardware version of the classifier based on NSL-KDD has the same accuracy of the software version. To make certain that the software and hardware implementations are truly equivalent, we performed an instance-by-instance inspection, and all vectors were classified identically in software and in hardware.

We repeated the same tests for the second classifier, derived from our custom dataset. Using the test dataset, only three instances were classified incorrectly. This performance is significantly better than NSL-KDD's; however, it should be noted that our custom dataset is specifically tailored for the detection of a single type of attack (probing attacks).

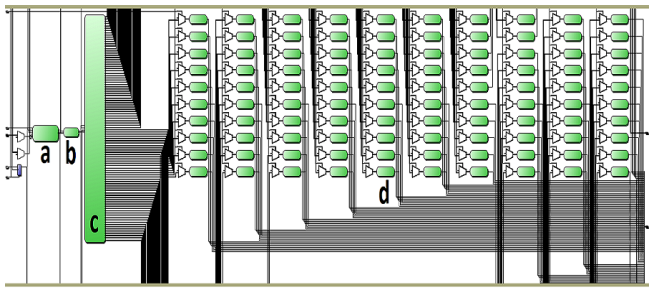

Figure 3. Hardware diagram of the classifier circuit: a) address increment circuit; b) ROM memory; c) FIFO memory; d) 100 decision tree classifiers.

Table I
Classifier Accuracy Using the NSL-KDD Dataset

Dataset	Instances	Correctly classified	Incorrectly classified
KDDTrain_20%.txt	25,192	24,288 (96.4%)	904 (3.6%)
KDDTrain.txt	125,973	121,508 (96.5%)	4,465 (3.5%)
KDDTest.txt	22,544	17,548 (77.8%)	4,996 (22.2%)

B. Circuit Area

The circuit synthesized for the classifiers is purely combinational, based on multiplexers and comparators. The NSL-KDD classifier uses 39 logic elements, representing 0.03% of the capacity of the EP4CGX150DF31C7 Cyclone IV FPGA. The circuit built from our probing dataset used 327 logic elements, or 0.2% of the FPGA.

C. Hardware Power Measurements

To measure the FPGA's power consumption, we used a Cyclone IV GX Development Kit and the Power Monitor tool from Altera. This tool measures the current consumption in the 8 tracks that supply power to the FPGA, and sends the results continuously to a PC via a JTAG interface.

We confirmed experimentally that the only power rail affected by changes in the hardware was VCC_{core}, which supplies power to the FPGA's core and hard IPs. Therefore, in the subsequent experiments, we used this value in all power consumption measurements.

Our first goal was to examine how the power consumption varies with the clock frequency and with the number of operations per second. The hardware for this experiment had 100 instances of the decision tree classifier. The results are shown in Table II. An active cycle of 100% means that one classification occurs on every clock cycle, whereas active cycles of 50%, 20% and 10% represent one classification at every 2, 5 and 10 clock cycles, respectively. The energy per classification (E_{TASK}) was calculated by multiplying the FPGA core voltage (1.2V) by the change in current when the circuit alternates between idle and normal operation, dividing this number by 100 (number of classifiers) and by the input vectors rate (number of vectors per second).

The first conclusion is that the energy to classify one input vector (E_{TASK}) is approximately constant, and independent of the operating frequency or the number of vectors classified per second. This is valid within the limits of our tests: between 50 and 200 MHz, and between $5{\cdot}10^6$ and $200{\cdot}10^6$ vectors per second. On average, it takes 23.8 pJ to classify a vector under these conditions, as shown in (1).

$$\mathcal{E}vector(J) = average(E_{\text{TASK}}) = 23.8pJ \qquad (1)$$

Fig. 4 shows in a 3D graph the energy needed to classify one vector, as a function of both the operating frequency and the active cycle. The graph is essentially flat, indicating that the energy per operation is practically constant. However, from the values in Table II, it is possible to observe a small

Table II
ENERGY PER CLASSIFICATION TASK AS A FUNCTION OF THE OPERATING FREQUNECY AND ACTIVE CYCLE.

F_{CLK} (MHz)	Active cycle (%)	Vectors/sec	ΔI_{core} (A)	E_{TASK} (pJ)
50	10%	$5{\cdot}10^6$	0.011	25.2
50	20%	$10{\cdot}10^6$	0.020	24.0
50	50%	$25{\cdot}10^6$	0.051	24.2
50	100%	$50{\cdot}10^6$	0.099	23.6
100	10%	$10{\cdot}10^6$	0.020	24.0
100	20%	$20{\cdot}10^6$	0.040	23.7
100	50%	$50{\cdot}10^6$	0.100	23.9
100	100%	$100{\cdot}10^6$	0.196	23.5
150	10%	$15{\cdot}10^6$	0.031	24.4
150	20%	$30{\cdot}10^6$	0.060	23.8
150	50%	$75{\cdot}10^6$	0.147	23.5
150	100%	$150{\cdot}10^6$	0.289	23.1
200	10%	$20{\cdot}10^6$	0.040	24.0
200	20%	$40{\cdot}10^6$	0.079	23.6
200	50%	$100{\cdot}10^6$	0.194	23.2
200	100%	$200{\cdot}10^6$	0.381	22.8

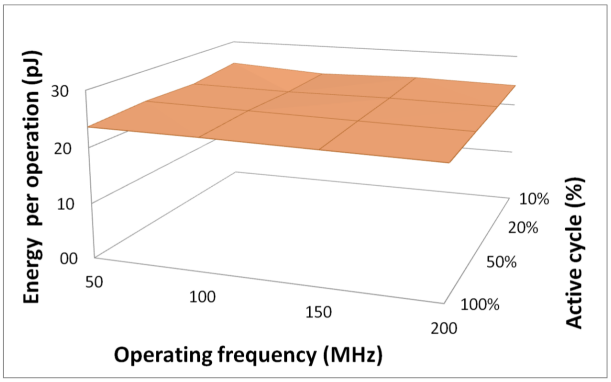

Figure 4. Energy spent to classify one vector as a function of the operating frequency and active cycle.

increase in the energy efficiency as the operating frequency increases and the active cycle approaches 100%.

The power measurements presented in Table II do not account for the total energy consumption of the classifier, because they only measure the difference in consumption when the circuit is idle from when it is operating. To fully characterize the energy consumption, we need to add the idle current because the FPGA draws a significant amount of current even when no operation is performed.

To measure this idle current, we disconnected the clock input from the test circuit. Different versions of our test circuit were compiled, using 1, 25, 50, 75, and 100 classifier instances. Fig. 5 shows the variation in idle current with the number of instances. The idle current consumption is directly proportional to the number of classifier circuits. Though this conclusion seems obvious at first (it means that the static power consumption is proportional to the circuit area), we must remember that in an FPGA this relation is

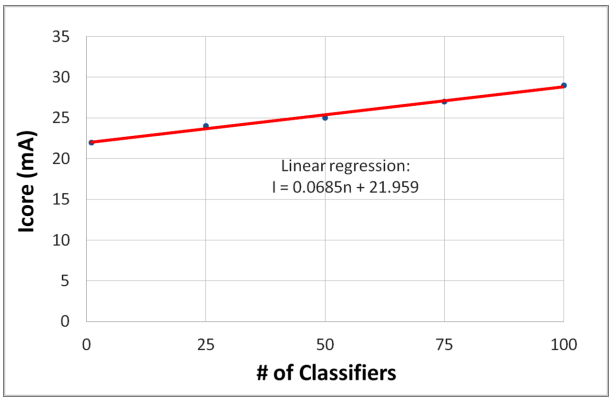

Figure 5. Idle consumption of the FPGA core, as a function of the number of classifier circuits.

not always straightforward, and needed to be checked.

A linear regression on the data shows that the idle current is approximately (21.96 + 0.0685n) mA, where n is the number of classifier circuits. The intercept value of 22 mA corresponds to the idle power consumption with zero classifier instances, and is due to the test harness circuits and the FPGA's base consumption. On top of that, each classifier instance adds an idle consumption of 68.5 μA.

This allows us to calculate the total power consumption for one instance of the classifier circuit, as a function of the number of operations performed per second. Adding the idle consumption (68.5 μA $\cdot$ 1.2 V = 82.2 μW) to the operating power consumption (23.8 pJ per vector/s), we obtain:

$$\mathcal{P}_{\text{classifier}}(W) = 82.2 \cdot 10^{-6} + 23.8 \cdot 10^{-12} \cdot \#vectors/s \quad (2)$$

As a final remark, the most important value obtained in these experiments is the energy required to classify one input vector (23.8 pJ, on average). In the final application, this will correspond to the energy cost of classifying a network packet as normal or attack. Even though the hardware and software implementations are completely different, the fact that they perform the same task gives us a number that can be compared across platforms, as long as we can calculate the energy required per classifying operation.

D. Software Power Measurements

To measure the power consumption of the classifier in software, we measured the total current drawn from the power supply by the Atom motherboard, using an Agilent 34401A 6 1/2 digit multimeter.

We first determined the idle power consumption of the Atom desktop board running the Ubuntu 12.10 operating system. The average current was 972 mA and the power supply was set to 15 VDC (a total power consumption of approximately 14.6 W).

After measuring the baseline power, we ran our test program, which took 15.4s to classify $2{\cdot}10^8$ vectors. During

this period, the average current was 1,040 mA. As in the FPGA experiment, we calculated the current and power differences (ΔI = 68 mA, ΔP = 1.02 W). With those values, we could calculate the energy expense to classify one vector:

$$\mathcal{E}vector(J) = \frac{68 \cdot 10^{-3} A \times 15V \times 15.4s}{200,000,000} = 78.5nJ \quad (3)$$

E. Comparison Hardware vs. Software

Because of the many differences between the hardware and software platforms, a direct comparison of most performance characteristics would be risky. However, since all implementations have the same ultimate goal, a direct comparison is possible if we focus on the energy required to classify one input vector. This approach abstracts away the influences of operating frequency, rate of vectors per second, power supply voltage, and even the hardware platform.

Table III compares the energy required to classify one vector in hardware and in software. The results indicate that the power consumption of the hardware classifier is 0.03% of the power consumption of the software classifier to perform the same task. Furthermore, it should be noted that the hardware version operates at a much higher rate than the software version. The FPGA classifies 200,000,000 vectors in one second, while the Atom CPU takes 15.4 seconds to classify the same number of vectors. In other words, the energy measurements were taken with the FPGA operating 15.4 times faster than the software implementation.

V. Conclusion and future work

We have created the entire infrastructure to implement simultaneously a hardware and a software version of the same decision tree classifier, for use in anomaly-based intrusion detection. Our experimental setup allows us to measure the power consumption of classifier algorithms in software and hardware. The energy measurements indicate that the power consumption of the hardware version is only 0.03% that of the software version, even though the hardware version was tested with a throughput 15.4 times higher.

We intend to improve our energy measurement methodology and to further evaluate the advantages of hardware versus software implementations. For this purpose, we plan to compare the energy efficiency of other machine learning algorithms, such as Naive Bayes, KNN, and SVM.

The next big step is to develop the hardware architecture for a complete network intrusion detection system (NIDS). We will use the PCIe bus to communicate between the FPGA and the CPU, and gradually offload computation (packet filtering, feature extraction, and vector assembly) from the Atom processor. This will allow us to evaluate which distribution of tasks between hardware and software is the most energy-efficient for network security applications.

Table III
Energy Cost to Classify One Vector - Software vs. Hardware.

Platform	Energy/vector (J)	Energy/vector (%)
Software (Atom CPU)	78.5 nJ	100 %
Hardware (Cyclone IV FPGA)	23.8 pJ	0.03 %

Acknowledgment

This work has been supported in part by the Intel Lab's University Research Office through the Intel Strategic Research Alliance (ISRA) program.

The first author would like to thank Coordination for the Improvement of Higher Education Personnel (CAPES) for his MSc grant.

References

[1] Cisco, "Visual networking index: Forecast and methodology, 2012-2017," Cisco, Tech. Rep., May 2013.

[2] P. Somavat and V. Namboodiri, "Energy consumption of personal computing including portable communication devices," *Green Engineering*, pp. 447–475, July 2011.

[3] P. Peterson, D. Singh, W. Kaiser, and P. Reiher, "Investigating energy and security trade-offs in the classroom with the atom leap testbed," in *Proceedings of the 4th conference on Cyber security experimentation and test*, 2011, pp. 1–9.

[4] J. M. Kizza, *Guide to computer network security*, 2nd ed. Springer, 2013.

[5] S. X. Wu and W. Banzhaf, "The use of computational intelligence in intrusion detection systems: A review," *Applied Soft Computing*, vol. 10, pp. 1–35, January 2010.

[6] 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, no. 10, pp. 11 994–12 000, 2009.

[7] S. T. Brugger, *Data Mining Methods for Network Intrusion Detection*, University of California, June 2004.

[8] NSL-KDD dataset. [Online]. Available: http://nsl.cs.unb.ca/NSL-KDD/

[9] J. Markey, *Using decision tree analysis for intrusion detection: a how-to guide*, SANS Institute InfoSec Reading Room, June 2011.

[10] A. Gregio, R. Santos, and A. Montes, "Evaluation of data mining techniques for suspicious network activity classification using honeypots data," in *Proceedings of the SPIE*, vol. 6570, April 2007.

[11] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[12] H. Chen, Y. Chen, and D. H. Summerville, "A survey on the application of fpgas for network infrastructure security," *IEEE Communications Surveys and Tutorials*, vol. 13, pp. 541–561, August 2011.

[13] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," Cisco, Tech. Rep., 2011.

[14] D. Meintanis and I. Papaefstathiou, "Power consumption estimations vs measurements for fpga-based security cores," in *International Conference on Reconfigurable Computing and FPGAs*, December 2008, pp. 433–437.

[15] Weka software. [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka