Towards an Energy-Efficient Anomaly-Based Intrusion Detection Engine for Embedded Systems

Eduardo Viegas, Altair O. Santin, *Member, IEEE*, André França, Ricardo Jasinski, Volnei A. Pedroni, and Luiz S. Oliveira

Abstract—Nowadays, a significant part of all network accesses comes from embedded and battery-powered devices, which must be energy efficient. This paper demonstrates that a hardware (HW) implementation of network security algorithms can significantly reduce their energy consumption compared to an equivalent software (SW) version. The paper has four main contributions: (i) a new feature extraction algorithm, with low processing demands and suitable for hardware implementation; (ii) a feature selection method with two objectives—accuracy and energy consumption; (iii) detailed energy measurements of the feature extraction engine and three machine learning (ML) classifiers implemented in SW and HW—Decision Tree (DT), Naive-Bayes (NB), and k-Nearest Neighbors (kNN); and (iv) a detailed analysis of the tradeoffs in implementing the feature extractor and ML classifiers in SW and HW. The new feature extractor demands significantly less computational power, memory, and energy. Its SW implementation consumes only 22 percent of the energy used by a commercial product and its HW implementation only 12 percent. The dual-objective feature selection enabled an energy saving of up to 93 percent. Comparing the most energy-efficient SW implementation (new extractor and DT classifier) with an equivalent HW implementation, the HW version consumes only 5.7 percent of the energy used by the SW version.

Index Terms—Low-power design, energy-aware systems, machine learning, classifier design and evaluation, feature evaluation and selection, hardware description languages, network-level security and protection, security and privacy protection

1 Introduction

According to yearly internet threat reports [1], there were 6,549 new vulnerabilities identified in 2014. The yearly average since 2006 is over 4,600, or 12 new vulnerabilities per day [2]. An embedded system connected to the Internet is potentially exposed to a large number of vulnerabilities that can be exploited by an attacker.

An intrusion detection engine can be used to protect an embedded system from attacks over a network. An anomaly detection engine (ADE) is an effective solution when the number of vulnerabilities is large and increasing. An ADE is composed of two main parts: algorithms implementing well-known inference techniques (e.g., machine learning) and a model (an attack profile).

There are two important phases in the development and use of an ADE: model building and classification. The model-building phase is usually performed offline, based on a set of known inputs and outputs. This task is computationally intensive, and it is usually performed purely in software. In the classification phase, network traffic is analyzed, and a set of features is extracted from the exchanged packets. These features are used by a classifier and an anomaly model to predict whether the analyzed traffic is normal or an attack. This phase is traditionally performed in software; however, different studies [3], [4] indicate that a hardware implementation can reach better throughput and lower energy consumption.

For the purposes of this work, a software (SW) implementation is a solution that runs on general purpose hardware (a commodity computing platform), and a hardware (HW) implementation is a circuit designed for a dedicated purpose. An embedded system [5] is a dedicated computer system, running application-specific SW and implemented in HW using microcontrollers, Systems-on-a-Chip (SoCs), or Field-Programmable Gate Arrays (FPGAs).

A hardware implementation of an algorithm can be custom-tailored to a specific problem, yielding a solution that is more optimized if compared to a SW implementation. The circuit can be described using a hardware description language (HDL) such as VHDL (Very High Speed Integrated Circuits Hardware Description Language). Traditionally, hardware-based intrusion detection uses signature-based algorithms, which perform bit-pattern matching in the network traffic. This approach usually provides good accuracy and performance [6], [7], [8]; however, it may not detect attacks in more complex scenarios [9], [10], [11], demanding the use of anomaly-based techniques [12].

- *E. Viegas and A.O. Santin are with the Pontifical Catholic University of Parana, Curitiba, Parana 80215-901, Brazil. E-mail: {eduardo.viegas, santin}@ppgia.pucpr.br.*
- *A. França, R. Jasinski and V.A. Pedroni are with the Federal Technological University of Parana, Curitiba, Parana 80230-901, Brazil. E-mail: andref@alunos.utfpr.edu.br, rjasinski@ieee.org, pedroni@utfpr.edu.br.*
- *L.S. Oliveira is with the Federal University of Parana, Curitiba, Parana 81531-980, Brazil. E-mail: lesoliveira@inf.ufpr.br.*

Manuscript received 28 Oct. 2015; revised 17 Mar. 2016; accepted 17 Apr. 2016. Date of publication 28 Apr. 2016; date of current version 19 Dec. 2016. Recommended for acceptance by D. Marculescu. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2016.2560839

In the literature, most works that discuss anomaly detection using hardware circuits do not take into account the resulting accuracy [13], [14], [15]. Usually, the developers create a direct implementation of the algorithm using an FPGA [13], [16], [17], making changes to account for the restrictions of a hardware implementation, but without considering the impact in the system accuracy [18]. In this naïve conversion from SW to HW, it is not possible to identify implementation errors that may appear due to modifications in the classifier, and it is not possible to guarantee that the detection engine in the FPGA is functionally equivalent to and has the same detection accuracy as the software version. Moreover, the update of the detection engine or model requires reprogramming the entire FPGA chip.

In battery-powered systems, energy efficiency is another important requirement, in addition to detection accuracy. Feature extraction is an important part of an anomaly-based detection engine, performing a constant network data stream (packets) analysis and consuming a significant amount of energy; therefore, in both SW and HW implementations, this module must be highly energy efficient.

This paper presents a new set of techniques to improve the energy efficiency in anomaly-based packet classification, using machine learning algorithms. We focus on the practical details of feature selection and extraction, and in the implementation of classifiers in SW and HW for embedded systems. We compare the energy efficiency of several SW implementations with their HW counterparts.

The paper is organized as follows. Section 2 provides background in intrusion detection and machine-learning classifiers. Sections 3 and 4 present the development of classifiers and feature extraction engines in SW. Section 5 describes the hardware implementation of the extractors and classifiers. Section 6 compares the experimental results in SW and HW. Section 7 summarizes the related works. Finally, Section 8 presents conclusions and future work.

2 Background

2.1 Anomaly-Based Intrusion Detection

The goal of an Intrusion Detection System (IDS) is to identify security violations in a computing system. A Network-Based Intrusion Detection System (NIDS) monitors the traffic by analyzing packets, hosts, and service flows in search of attacks [19].

Denial-of-Service (DoS) is a current and dangerous attack [1] in which attackers try to render a host or its services unavailable to the legitimate users. This is normally done by sending an amount of requests that a host cannot handle properly, or by sending a malformed request that causes a system or a service to terminate abnormally [20].

In order to detect DoS attacks, a NIDS can use a signature-based engine. This approach usually produces low false-alarm rates; however, it is unable to detect many attack variants. Moreover, as the number of attacks increases, the number of signatures also increases, making the usage of the whole set of signatures impractical for online detection [21].

Anomaly-based detection consists in creating a behavior model that is used to detect deviations from normal behavior. An anomaly-based classifier assigns a class (e.g., normal or attack) to each event (e.g., packet or flow). This approach can often detect attack variations, but it tends to produce higher false-alarm rates than the signature-based approach [22].

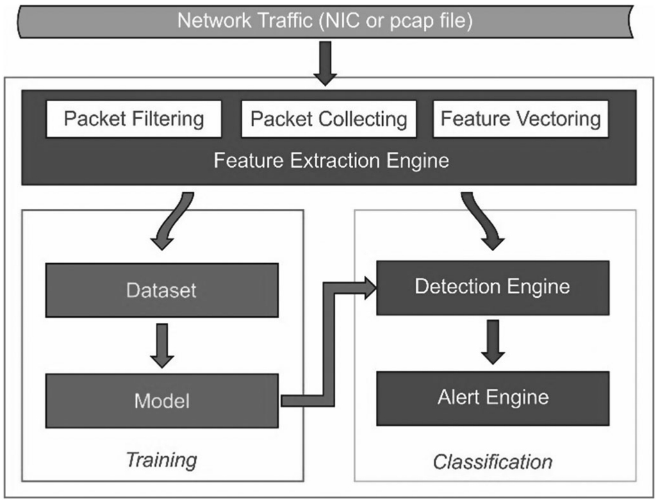

Fig. 1. Overview of a typical anomaly-based intrusion detection process.

Machine learning (ML) is often employed to implement anomaly-based intrusion detection (Fig. 1). The network traffic is collected from the Network Interface Card (NIC) or from a *pcap* (packet capture) file containing previously captured network traffic. The packets are then filtered and sent to a feature extraction engine, which computes flow-based and header-based attributes. These attributes are assembled into a feature vector, which provides the input data for the training or classification phases of a classifier.

To perform a classification, the ML algorithms use a behavioral model that can predict certain classes of network traffic (e.g., normal or attack). Creating a model requires a set of feature vectors (a dataset) whose classes are known for each vector. The ML algorithm learns the behavior model, which is then tested with a set of samples that were not used in the model creation. The process is repeated until the desired accuracy is achieved or cannot be further improved. If the nature of the network traffic changes, it may be necessary to regenerate the model—for instance, when a different kind of attack arises and cannot be detected.

2.2 Feature Sets for Network Intrusion Detection

A NIDS uses only the information available in network packets to distinguish attacks from legitimate (normal) requests. The feature set used in this paper was based on previous works existing in the literature [23], [24], [25]. In total, 50 features were extracted from each network packet. Table 1 show the way in which the features are organized: (i) *header-based* (features extracted directly from each packet header); (ii) *host-based* (extracted from the general communication history or data flow between two hosts); and (iii) *service-based* (extracted from the communication history between two hosts, and specific to a single service).

2.3 Feature Extraction

The feature extraction phase (Fig. 2) starts with the selection of the desired kind of traffic by filtering the network packets based on protocol fields or flags, patterns of bits, or packet content. This filtered packet set contains the data that will

TABLE 1
Categories of Extracted Features

Feature category	Number of Features	Example
Header-based: features extracted directly from a single packet header.	27	SYN flag from TCP protocol header.
Host-based flow: features extracted from data exchanged between two hosts, involving various packets.	17	Number of bytes sent from a client to a server in the last 2 seconds.
Service-based flow: features extracted from multi-packet data exchanged between two services.	6	Number of bytes sent from a client to a server service in the last 2 seconds.

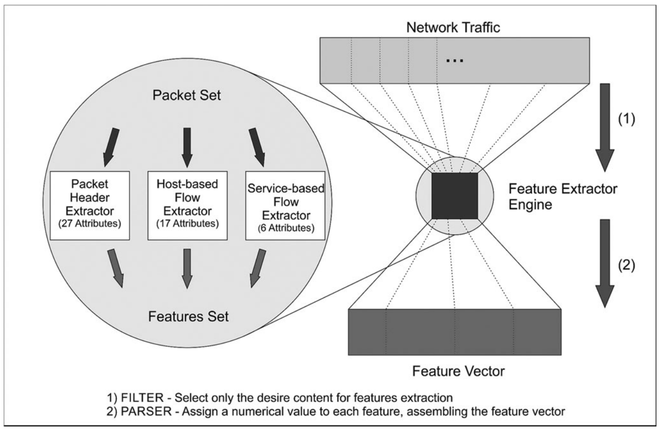
Network Traffic
...
Packet Set
Packet Header Extractor (27 Attributes)
Host-based Flow Extractor (17 Attributes)
Service-based Flow Extractor (6 Attributes)
Features Set
(1)
Feature Extractor Engine
(2)
Feature Vector
1) FILTER - Select only the desire content for features extraction
2) PARSER - Assign a numerical value to each feature, assembling the feature vector

Fig. 2. Overview of the feature extraction process.

be processed or used directly to obtain the desired features (Table 1). The features are then parsed, scaled to a uniform range, and assembled into a feature vector. The feature extraction engine (Fig. 2) typically performs an important part of this process.

The analysis and extraction of host-based and service-based flows can be obtained by a flow extraction engine or by a netflow-based [26], [27] approach.

2.4 Feature Selection

The feature selection process identifies the most relevant features from a feature set (Fig. 2), aiming at improving the classifier accuracy and reducing the computational load during classification. Feature selection algorithms can be classified into two categories, based on whether the selection is performed independently from the learning algorithm used to construct the classifier. If feature selection is independent from the learning algorithm, the technique is said to follow a *filter* approach. Otherwise, it follows a *wrapper* approach [28].

While the filter approach is generally computationally more efficient, its major drawback is that an optimal selection of features may require the inductive and representational biases of the ML algorithm used to build a classifier. On the other hand, the wrapper approach has the computational overhead of evaluating candidate feature subsets by executing the ML algorithm on the dataset, using each subset under consideration.

Because a complete search over all possible subsets of a feature set (2^N, where N is the number of features) may not be computationally feasible, several authors have explored heuristics for feature subset selection. Genetic Algorithms (GA) are an interesting alternative because they do not assume restrictive monotonicity and can use multiple-objective optimization (for example, classification accuracy and energy consumption).

A general multi-objective optimization problem consists of a number of objectives associated with inequality and equality constraints. Solutions to a multi-objective optimization problem can be expressed mathematically in terms of non-dominated points (a solution is dominant over another only if it has better performance in all criteria). A solution is said to be Pareto-optimal if it cannot be dominated by any other solution available in the search space.

Conflicting objectives are a common difficulty in multi-objective optimization. In general, none of the feasible solutions allows simultaneously optimal solutions for all objectives. Thus, mathematically, the most favorable Pareto-optimum is the solution offering the least conflict between objectives. In order to find such solutions, classical methods scalarize the objective vector into a single objective.

The simplest of all classical methods is the weighted sum, which aggregates the objectives into a single and parameterized objective through linear combination. However, setting up an appropriate weight vector also depends on the scaling of each objective function. Therefore, the solution obtained through this strategy largely depends on the underlying weight vector.

To overcome such difficulties, Pareto-based evolutionary optimization has become an alternative to classical techniques such as the weighted sum. Goldberg [29] first proposed this approach, which explicitly uses Pareto dominance to determine the reproduction probability of each individual. In essence, it consists of assigning rank 1 to non-dominated individuals and removing them from contention, then finding a new set of non-dominated individuals, ranked 2, and so forth.

A popular multi-objective genetic algorithm that has been successfully applied to multi-objective feature selection [30] is the non-dominated sorting genetic algorithm (NSGA-II) [31]. The idea behind NSGA-II is to use a ranking selection method to emphasize good points and a niche method to maintain stable subpopulations of good points. It varies from simple genetic algorithms only in the way the selection operator works.

3 Development of Energy-Efficient Classifiers in Software

This section outlines the development process of the classifiers used in our experiments, including the model training, software implementation, and experimental measurement phases. For our experiments, we have built a dataset aimed at DoS attacks.

3.1 Power Measurement in Software

Despite the high demand for energy-efficient systems, there are very few tools available to measure and evaluate the

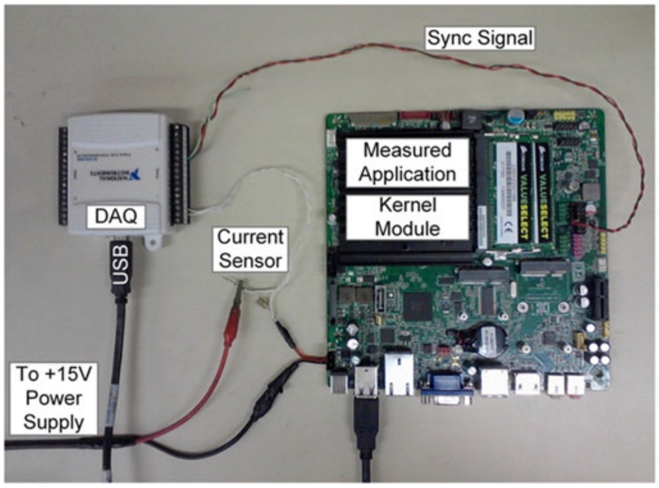

Fig. 3. Power measurement platform for software applications.

power consumption of software applications. Moreover, the currently available solutions are unable to isolate the power consumption of an application running in a multitask environment [32], [33], [34], [35]. To overcome these problems, we have developed a custom power measurement platform composed of a hardware environment, a measurement application, and a kernel-level probe module (KPM). The KPM detects when the monitored application is running and triggers a signal in the motherboard's parallel port; while this signal is asserted, the monitored application is running. The measurement application periodically samples this signal and the voltage across a current sensor (Fig. 3), a shunt resistor in series with the board power supply. The current samples are integrated over the time the monitored application was actually executing in the microprocessor, providing the energy consumption of the application.

Fig. 3 shows the general setup of the measurement platform. The hardware environment is composed of a DN2800MT Atom motherboard with an Intel N2800 CPU, 4 GB DDR3 RAM, and a 500 GB hard drive. A 270 mΩ resistor is used as current sensor for the main power line. The system is powered by a 15 VDC power supply, and the current consumption is sampled by a National Instruments USB-6008 data acquisition device (DAQ), with a 12-bit resolution, at 5,000 samples/s, in differential mode.

The platform was used to measure the power consumption of all SW algorithms described in this paper. The energy consumed per operation (entire extraction or classification of one packet) was calculated with equation (1), where $P_{running}$ denotes the motherboard's power consumption while the algorithm is running, and P_{idle} denotes the motherboard baseline power consumption. We discount the baseline consumption because our goal is to isolate the consumption of the measured application in order to compare it with its HW counterpart.

$$Energy\ per\ operation\ (J) = \frac{(P_{running} - P_{idle}) * processing\ time}{number\ of\ packets} \tag{1}$$

The developed power measurement platform can be used with virtually any operating system, and it is independent of the hardware used in the motherboard.

TABLE 2
Services Used for Normal Traffic Simulation

Service	Description
HTTP	The 1,000 most visited worldwide websites were mirrored (www.alexa.com/topsites) and hosted in the honeypot. Each HTTP client requests a random website.
SMTP	A script for each SMTP client sends a mail with a 50-400 bytes subject and 100-4,000 random bytes in the body.
SSH	A script for each SSH client logs into the honeypot host and executes a random command from a list from 100 possibilities.
SNMP	A script for each SNMP client walks at random through a predefined MIB (Management Information Base) from a predefined list of possible MIBs.
DNS	A script is run for every name resolution, making same request to the honeypot service.

3.2 Data Generation

Rather than using publicly available data, we have created our own dataset in a controlled environment, in order to prevent the problems exposed in [36] when traffic recorded from a real environment is used. The proposed method aims at ensuring the desired properties of a NIDS dataset [58]. Among other characteristics, the dataset should be publicly available and contain realistic network traffic, it should be similar to production environments, its packets should be well-formed and correctly labeled (previous class assignment), it should have normal and attacker profile diversity, and the attacks should be correctly implemented and easy to update and reproduce.

The deployed scenario uses a single host as a server when generating the background traffic (normal traffic samples). This server hosts a number of services and responds to all received requests. It generates real service responses using a honeypot tool (HoneyD [37]), resulting in a database with real and valid traffic. HoneyD was configured as a high interactivity honeypot, hosting five different services (Table 2). To create client requests, we used 100 hosts running automated scripts. The interval between successive requests from the same client varies randomly between zero and four seconds, following a uniform distribution. All the network traffic produced in this scenario was captured and stored in a *pcap* file.

The attack traffic was created using well-known exploit tools to guarantee a correct attack characterization. The scenario simulates four different DoS attacks: synflood, udpflood, icmpflood, and slowloris (an attack that opens many connections to a webserver and keeps them open). Each attack was originated from a different virtual machine, with varying packet send rates and attack durations.

The class label for each sample (normal or attack) was automatically assigned, as we know the source IP address of each machine generating normal traffic or attacks. This eliminates the possibility of wrong labeling that could generate inaccurate models during the learning process. Every feature that could identify a specific machine by means other than its behavior (for instance, IP or MAC addresses) was removed from the feature vector.

The scenario ran for 30 minutes. Legitimate clients made requests during the entire time, ensuring the existence of

TABLE 3
Number of Captured Packets in the Generated Scenario

Traffic	Generated Behaviors	Number of Packets	Packets Representativeness (%)	
HTTP	65,786	20,238,802	73.61	97.24
SMTP	35,110	2,298,222	8.36	
SSH	2,579	1,048,482	3.81	
SNMP	10,111	3,017,731	10.97	
DNS	-	135,188	0.49	
SYNFLOOD Attack	8	471,288	1.71	2.76
UDPFLOOD Attack	6	121,645	0.44	
ICMPFLOOD Attack	6	130,698	0.47	
SLOWLORIS Attack	5	37,814	0.14	
TOTAL	**113,611**	**27,499,870**	**100.00**	

background traffic for the entire period. The periods from zero to seven minutes and from twenty-three to thirty minutes are attack-free, allowing us to analyze the system behavior with normal packets only. During the period between 7 and 23 minutes, the attacks are deployed. Table 3 shows the traffic distribution observed during the database generation and the number of different behaviors for each traffic type.

The vast majority of events (network packets) in the database are normal (97.24 percent). Because the two classes do not occur with the same frequency, we must use additional metrics (such as the false-positive and false-negative rates) when evaluating the classifiers performance. To reduce the complexity during the model-training phase and to allow an equal representation from both classes on the datasets, we stratified our database (secplab.ppgia.pucpr.br/eeids). We used 25 percent of the attack events for model training, 25 percent for model validation (feature selection process evaluation), and 50 percent for model testing (final accuracy rate). To guarantee that both classes have the same representation during the model training, we randomly choose the same number of normal and attack events from the database. The remaining normal events are used for model testing.

3.3 Preprocessing

After building the samples database, a preprocessing phase may be necessary for some classifier algorithms. In this work, we have evaluated the DT (Decision Tree), NB (Naive-Bayes), and kNN (k-Nearest Neighbors) classifiers. The DT classifier does not require any pre-processing. The NB classifier requires a discretization step. Every feature admitting a real number as value was discretized, allowing the classifier to compute the individual probability for each class (normal or attack). The kNN classifier requires a normalization step; every feature was normalized to the range -1.0 to $+1.0$, to avoid disproportional influences of different features during the distance calculation. For every nominal (enumeration) feature, such as the TCP connection status, a corresponding numerical value was assigned.

3.4 Feature Selection

Different features require different amounts of processing to be computed, which implies different energy consumption

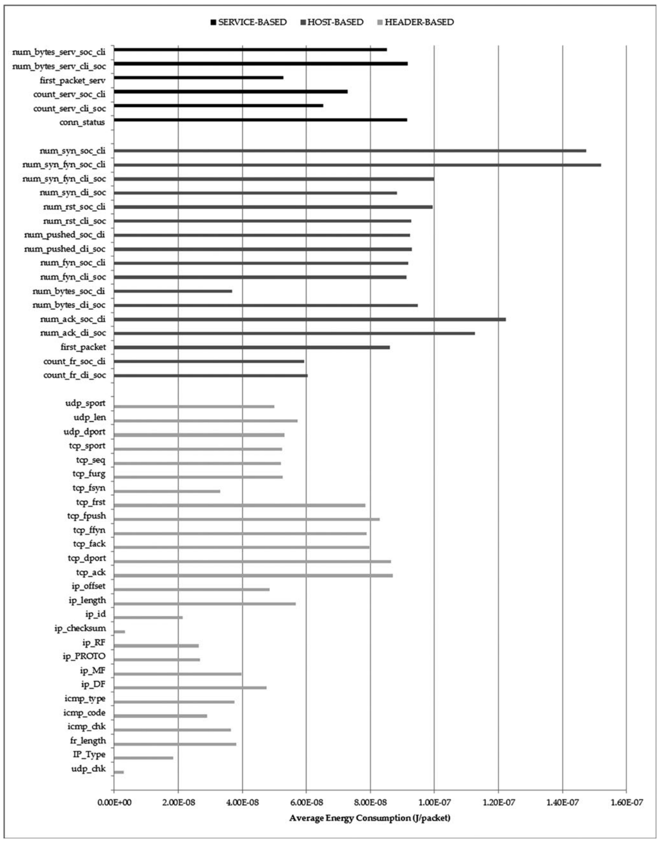

Fig. 4. Average energy consumption for the extraction of each feature.

levels. We have measured the energy consumption of each individual feature and used these values in the feature selection process. A total of 100 measurements, with 1 million packets each, were performed for each of the 50 features. The energy consumption to extract each feature individually is the difference between the energy consumed to extract all 50 features and the energy consumed to extract all but the feature of interest. The average energy consumption for each feature, using the hash-based extractor (Section 4.2), is shown in Fig. 4.

In the subsequent energy measurements for the extractor and classifiers, we have considered three feature selection scenarios: no feature selection, single-objective (accuracy only) selection, and dual-objective (accuracy and energy consumption) selection. In our dual-objective scenario, the first objective is to minimize the error rate of the attack model, and the second objective is to minimize the energy consumption for feature extraction. Giving both objectives the same weight, the relative energy consumption is calculated by summing the energy consumption for each used feature, divided by the total energy consumption when using all features. The error rate is evaluated using the validation dataset (Section 3.2) and defined as the misclassification rate of the obtained model. The relationship between the error rate and the estimated energy consumption is shown in Fig. 5. Each point represents a population of the last generation for each classifier.

The power measurements were performed using the platform described in Section 3.1. The feature extraction module was modified to extract only a subset of features in each run.

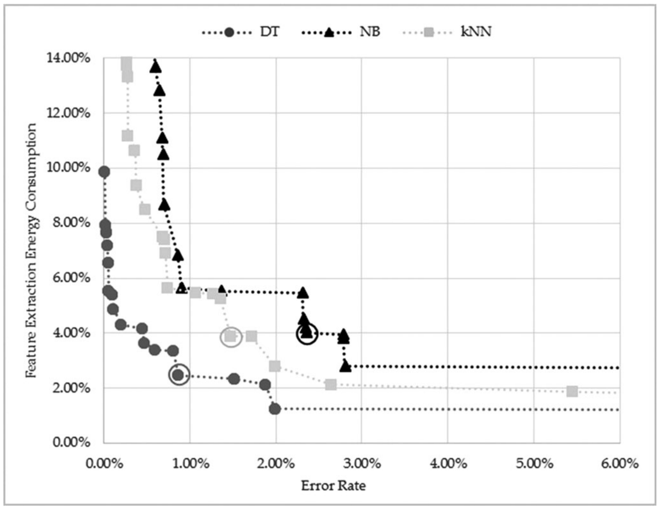

Fig. 5. Relationship between accuracy and energy consumption for the three classifiers.

The GA (single-objective) and NSGA-II (dual-objective) were used with 100 generations and 100 populations for each generation, a mutation probability of 3.3 percent, and a 60 percent crossover probability. The classifiers were developed using the Weka framework [38].

The results obtained for the DT, kNN, and NB classifiers are discussed in Section 3.5. The accuracy value was obtained using the test dataset, which contains only packets that are not present in the training or validation datasets.

3.5 Energy Consumption

Following the feature selection stage, a model was generated for each combination of classifier (DT, NB, and kNN) and feature selection method. The SW implementations of the classifiers are direct translations of their algorithms and were obtained by translating the output of the Weka framework to the C++ language.

Table 4 compares the accuracy and energy consumption of the three classifiers and three feature selection methods. The results indicate that feature selection can provide a significant reduction in energy consumption, while maintaining approximately the same accuracy rate.

We can evaluate the energy savings achieved in the *feature extraction* stage by using the “no selection” method as a baseline. The relative savings were 44.5 percent (single-objective) and 51.5 percent (dual-objective) for DT, 36.5 percent (single-objective) and 52.0 percent (dual-objective) for kNN, and 45.5 percent (single-objective) and 51.5 percent (dual-objective) for NB. On average, feature selection provided an energy saving of 42.2 percent (single-objective) and 51.7 percent (dual-objective), in the extraction stage. Therefore, dual-objective feature selection provided an average 9.5 percent of extra savings in feature extraction, when compared to single-objective selection.

Similarly, we can evaluate the energy savings achieved in the *classification* stage using the “no selection” method as a baseline. The relative savings were 44.4 percent (single-objective) and 44.4 percent (dual-objective) for DT, 60.2 percent (single-objective) and 80.7 percent (dual-objective) for kNN, and 85.8 percent (single-objective) and 92.9 percent (dual-objective) for NB. On average, feature selection provided an energy saving of 63.5 percent (single-objective) and 72.7 percent (dual-objective). Therefore, dual-objective feature selection provided an average 9.2 percent of extra savings, when compared to single-objective selection.

TABLE 4
Comparison of Energy Consumption for Feature Extraction Engine And ML Classifiers

Classifier (feature selection technique)	Energy Cons. for Feature Extraction (μJ)	Energy Cons. for Classification (μJ)	Total Energy Cons. (μJ)	Accuracy (%)
DT (no-selection)	2.00	0.09	2.09	99.94
DT (single-objective)	1.11	0.05	1.16	100.00
DT (dual-objective)	0.97	0.05	1.02	99.14
kNN (no-selection)	2.00	169.74	171.74	98.98
kNN (single-objective)	1.27	67.57	68.83	99.80
kNN (dual-objective)	0.96	32.79	33.75	98.53
NB (no-selection)	2.00	2.53	4.53	99.02
NB (single-objective)	1.09	0.36	1.45	99.98
NB (dual-objective)	0.97	0.18	1.15	99.41

On average, the energy savings in the entire extraction and classification process (total energy consumption) were of 57.5 percent (single-objective) and 68.7 percent (dual-objective). In summary, dual-objective feature selection provided an average energy savings of 9.3 percent while incurring an average 0.90 percent accuracy loss, when compared with single-objective feature selection.

4 Development of Feature Extractors in Software

As indicated in Table 4, a large part of the energy cost is due to the feature extraction stage. However, we did not find previous work in the literature measuring the energy cost for extracting each feature individually. In our work, we have paid special attention to reducing the energy consumption of this module. In this section, we present two approaches to the implementation of feature extraction engines in software, which we named *table-based* and *hash-based*. We then compare the two approaches with a third alternative called *netflow-based*, a table-based implementation found in commercial networking equipment.

4.1 Table-Based Extraction Method

The computation of feature values requires storing information about the data exchanged between hosts or about the services (Table 1, Section 2.3). The table-based implementation approach uses a table indexed by the IP addresses of the communicating hosts. This table, called the host lookup table, indexes two other tables: the host flow table and the service lookup table. The values of host-based features can be accessed directly in the host flow table. For service-based features, another lookup is needed to reach the flow table containing the features values (Fig. 6). The service lookup table uses the client port number as index.

The host and service flow tables store pieces of information corresponding to packets read from the network; this

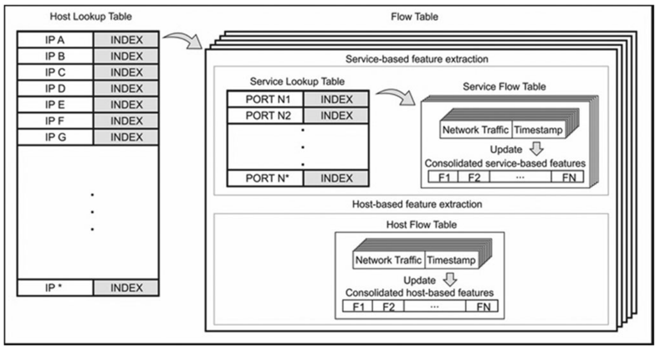

Fig. 6. Table-based feature extractor implementation.

information must be consolidated to provide the feature values over a period of time (time window). Therefore, for each table entry, it is necessary to keep the network traffic corresponding to the chosen time window, as well as its time of occurrence. During extraction, the feature values are computed by adding the contribution of each packet within the time window; packets with a timestamp older than the time window are discarded.

The overall operation can be described as follows: each time a packet arrives at the feature extraction module the host lookup table is accessed using the client IP address. The next step is to calculate the updated feature values. Depending on the kind of feature being extracted, the algorithm follows one of two possible paths.

For host-based features, only the host flow table is accessed. This table contains the previous traffic exchanged with a single host. The entire table is scanned, and the data contributing to the current feature is consolidated into the new feature value. Only packets within the configured time window are used, and older traffic is discarded.

For service-based features, the service lookup table is accessed using the client port number as index. This table contains the traffic exchanged with a single host and belonging to a single service. The entire table is scanned and the values consolidated. Packets outside the current window are also discarded.

In the table-based extractor, there are at most 2^{32} entries in the host lookup table (corresponding to all possible numeric values of IPv4 addresses) and 2^{16}entries in the service lookup table (corresponding to all possible numeric port values). Each entry in the service lookup table contains pointers to tables with the packet data used to calculate the consolidated service-based features. All tables and entries are allocated dynamically. We use a two-second time window to calculate the feature values, an interval commonly used in the literature [23], [24], [25].

The most demanding task in table-based extraction is to update the feature values by summing the contribution of each packet when a new packet is received. For each feature, the corresponding flow table (host or service) must be scanned, and all previous packets within the current time window must be read to calculate the updated feature value. To implement the time window, each packet is tagged with a timestamp indicating the moment it was received. This provides a very fine-grained control, but it incurs a high computational cost. The window is effectively a *sliding time window*, because only the past two seconds are taken into account when calculating a consolidated value.

TABLE 5
Classifier Accuracy for Different Sizes of the Sliding Time Window

Sliding Time Window size (sec.)	Energy Consumption per packet (μJ)	Processing Time per packet (μs)	Classifier Accuracy (%)		
			DT	5NN	NB
0.5	366.31	344.50	99.86	97.30	98.19
1.0	456.11	412.74	99.90	97.75	98.32
1.5	538.74	478.86	99.90	97.80	98.42
2.0	603.78	525.15	99.91	97.89	98.62

One approach to reduce the computational demand is to reduce the window size. This results in lower processing and memory costs; however, it may negatively affect the classifier accuracy. To evaluate this tradeoff, we have measured the impact of the time window reduction in classifier accuracy. We used the method introduced in Section 3.1 and the classifiers with no feature selection (the worst case for accuracy rate – see Table 4, Section 3.5). We used four different time ranges for the sliding time window, from 0.5 to 2 seconds (Table 5). The largest accuracy reductions were observed for a 0.5 seconds time window: 0.05 percent for DT, 0.59 percent for kNN (5NN), and 0.43 percent for NB. On the other hand, the reduction from 2 to 0.5 seconds provided an average energy savings of 39.3 percent. This indicates that a sliding time windows of 0.5 seconds may be useful, for instance, when the embedded system reaches a critical battery level.

Some important issues regarding the implementation of a table-based extractor or a netflow-based extractor (a standard data collection method available in many networking products) are discussed next.

An important shortcoming of the table-based approach is the need for storing the received packets. Despite any possible reduction with the shortening of the time window, if the throughput is high, the required memory increases, as well as the processing demands. As will be discussed in Section 4.4, there is a high computational cost to maintain and update this information in the table-based approach. The table-based extractor is also difficult to implement in hardware, because it requires dynamic memory management and the iterative scanning and checking of previous network packets. More details about the table-based extractor in hardware are provided in Section 5.1.

A netflow-based implementation also has a series of drawbacks. A *netflow-enabled device* simply keeps a cache of IP flows that have traversed that device. The first implication is that a second program, a *netflow collector*, must analyze the reported flows and extract the NIDS features. This program operates similarly to *tcptrace* (www.tcptrace.org). Second, there is some delay between the moments when the packets are captured and the netflow collector receives the flow information. Normally, the flow is reported when one of three conditions occur: (i) the flow is inactive for a certain period (CISCO devices usually adopt a 15-second idle timeout [39]), (ii) the flow has been active for a long time, or (iii) a TCP flag indicates the flow has ended. Additionally, each

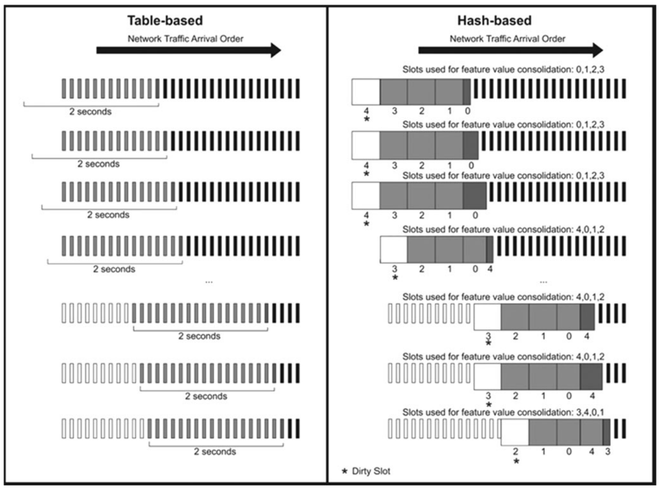

Table-based
Network Traffic Arrival Order
2 seconds
Hash-based
Network Traffic Arrival Order
Slots used for feature value consolidation: 0,1,2,3
Slots used for feature value consolidation: 4,0,1,2
Slots used for feature value consolidation: 3,4,0,1
4 3 2 1 0
* Dirty Slot

Fig. 7. Comparison of the table-based (sliding time window) and hash-based (time slots) approaches.

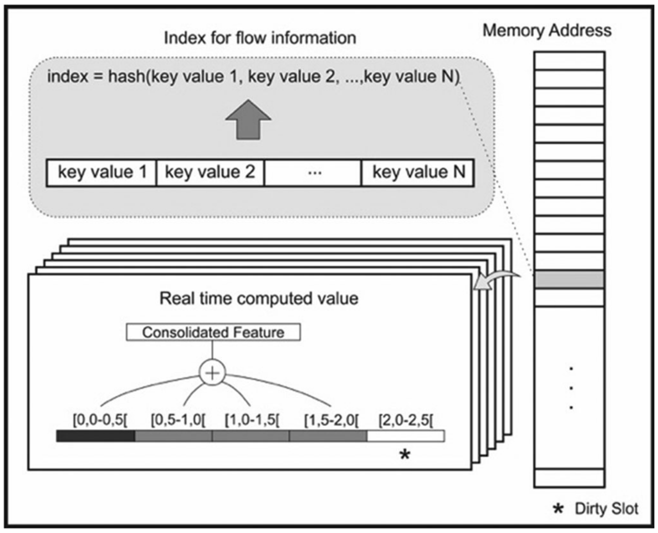

Fig. 8. Overview of the hash-based feature extractor.

reported flow must be consolidated in a netflow collector, and further processing is required for each feature being extracted. This makes netflow-based feature extraction a costly task.

4.2 Hash-Based Extraction Method

One of the problems with the table-based approach is the need for keeping the previous history of flow information within the chosen time window, as explained in Section 4.1. To solve this problem, we propose a new approach based on time slots. This approach has two main advantages: (i) it makes it easier to remove the influence of older packets from the feature calculation, and (ii) allows a simple resizing of the time window, which can be reduced from 2.0 to 0.5 sec., for instance, when the device battery level is low.

Our implementation uses five time slots, numbered from 0 to 4. At any given moment, four of the slots are used to compose the feature value; the remaining slot (the “dirty slot”) is not considered in the calculation and is scheduled for reset (Fig. 7). Each slot accumulates flow information corresponding to a time window of 0.5 sec.; the slot that is currently accumulating flow information (the *active slot*) is shown with a darker shade in Fig. 7. To compute a consolidated feature value, one must add up the values of the active slot and the three most recently updated slots.

Every 0.5 seconds, the next slot (in decreasing numerical order) becomes active. When the active slot number reaches zero, it starts again from four, in a circular countdown fashion. The same happens with the number of the dirty slot; every 0.5 seconds, a periodic maintenance task resets all the dirty slots in memory.

In the table-based approach, the time considered in the feature calculation corresponds exactly to the past two seconds. In contrast, in the time slots approach, the feature may correspond to an interval between the last 1.5 and 2.0 seconds. As shown in Table 5, this variation does not incur a significant accuracy loss. A diagram comparing the calculation of a feature value in the table-based and hash-based approaches is shown in Fig. 7.

In the time-slot-based approach described above, a hash function provides the address for the time slots corresponding to a feature (Fig. 8). Contrary to the table-based approach, the hash-based approach does not require a series of indirections and lookup operations.

Our approach uses two indexing schemes to obtain an address for the time slots. For host-based features, a key is created from a unique feature identifier (feature ID) and the client IP address. For service-based features, the key is composed of the client IP address, feature ID, and client port address (Fig. 8). In both cases, the key goes through the same hash function. Each memory position contains the five slots corresponding to the flow information for a single feature. Each slot is 2 bytes wide, yielding 10 bytes per flow. The output of the hash function is 16 bits long; therefore, our current implementation is able to handle 2^{16} flows with 10 bytes each, requiring in total 5 Mbits. This memory size is within the limits of the embedded system used in our experiments, which has a total of 6.6 Mbits.

Our implementation uses the well-known FNV (Fowler–Noll–Vo) hash function – www.isthe.com/chongo/tech/comp/fnv. This function allows a variable number of output bits, making it possible to adapt to different table sizes and gracefully degrading the hash properties in a controlled way. Our flow table is stored in a contiguous memory region (flat memory model) with a fixed size. This has two implications. First, because the address is provided by the hash function, there may be unused memory locations. Second, it is possible that different keys have the same hash value (causing hash collisions), and two different features may be allocated to the same memory address. The impact of hash collisions in the classifier accuracy is evaluated in Section 4.3.

4.3 Impact of Hash Collisions

The occurrence of hash collisions (Fig. 9, left y-axis) and the total memory size (right y-axis) depend on the number of entries available in the table (x-axis): the bigger the number of possible entries, the lower the number of collisions, and the higher the memory requirements. The graphs shown in

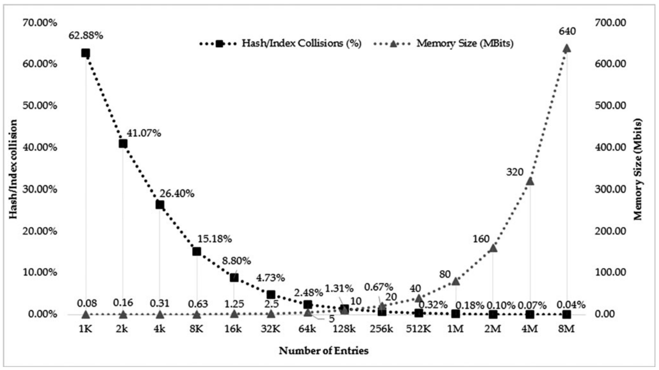

Fig. 9. Correlation between hash collisions occurrence and the number of index entries.

Figs. 9 and 10 were obtained with the dataset introduced in Section 3.2, composed of over 27 million packets.

For 2^{16} (or 64 k) memory entries, we obtained a collision rate of 2.48 percent, using 5 Mbits of memory. For 2^{24} memory entries, the collision rate drops to 0.04 percent, but the required memory is 640 Mbits. To evaluate this tradeoff, Fig. 10 shows the effect of collisions on classification accuracy. For these measurements, a version of each classifier with no feature selection was used. For a memory size of 2^{16} entries, the results show a slight decrease in classifier accuracy: 0.21 percent for DT, 0.66 percent for kNN, and 0.69 percent for NB. This indicates that the 2^{16} memory entries used in our implementation are enough to provide a good accuracy, even in the presence of hash collisions.

4.4 Comparison of Extraction Approaches

Using the power measurement platform described in Section 3.1, we measured the average energy consumption and processing time required to extract one packet, using the three extractors (Table 6). For the netflow-based implementation, the *fprobe* [40] library was used, exporting "ready" flows every 5 seconds, with an inactive lifetime of 1 minute and an active lifetime of 5 minutes. Only the processing time used for flow extraction (flow caches) was considered in Table 6; for an intrusion detection application, it would be necessary to assemble each flow into a feature vector, as observed in Section 4.1.

The results show that the hash-based extractor is 189 times faster and consumes only 0.33 percent of the energy used by the table-based implementation. Moreover, it is 8.7 times faster and consumes only 22 percent of the energy used by the netflow-based implementation. The hash-based extractor was implemented with 2^{16} table entries, as described in Section 4.2. These results indicate that the hash-based approach is a promising candidate for implementation in embedded systems.

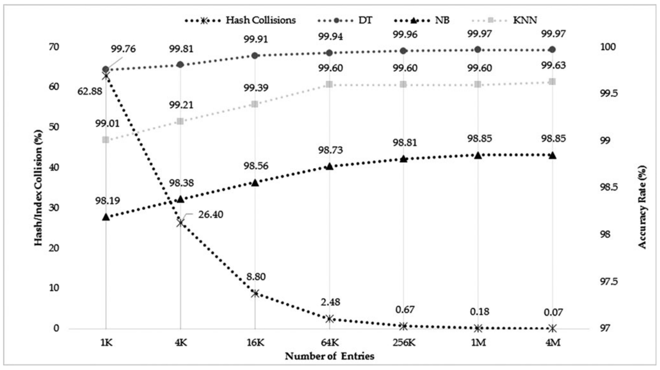

Fig. 10. Relationship between classifier accuracy, hash collisions, and number of index entries.

Table 6
Energy Consumption and Processing Time for Each Extractor

Measurement item	Table-based	Netflow-based (flows cache)	Hash-based
Energy Consumption (uJ/packet)	603.78	9.05	2.00
Processing Time (us/packet)	525.15	24.19	2.78

5 Development of Extractors and Classifiers in Hardware

5.1 Table-Based Extractor Issues

The direct implementation of the feature extraction algorithm described in Section 4.1 would impose a series of challenges in hardware. One of the main limitations is the amount of memory required to keep the previous history of flow information. This amount is proportional to the number of known hosts and flows; in general, each feature requires at least one 32-bit accumulator. Another implication is that the feature computation must consider only the data exchanged within the sliding time window; therefore, a maintenance task would be necessary to remove packets outside the current window.

In a software implementation, memory is allocated from an application-wide memory pool that is accessed uniformly. In contrast, a typical HW implementation uses a number of localized storage elements and on-chip memory structures. This helps eliminate bottlenecks in memory accesses, but it requires that the number of storage elements be fixed at compile time, when the circuit is synthesized. This restriction makes the table-based feature extractor ill-suited for a hardware implementation. For these reasons, in our work, only the hash-based extractor was implemented in HW.

5.2 Hash-Based Extractor Implementation

Fig. 11 shows the block diagram of the hash-based feature extractor implemented in HW, which uses the same

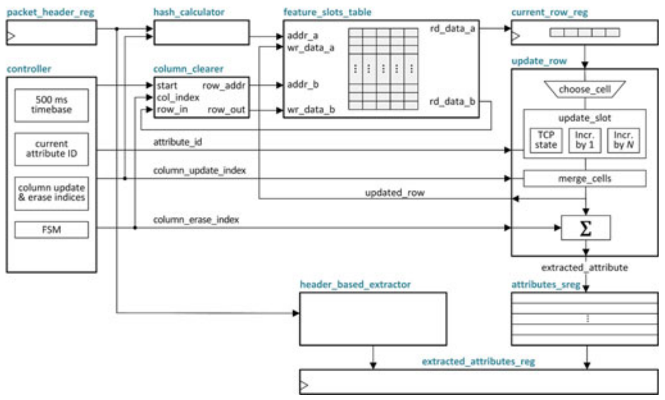

Fig. 11. Block diagram of the hash-based feature extractor in HW.

TABLE 7
Area of the Implemented HW Extractors

Extractor for	Attributes	Logic Cells (LCs)	Memory Bits
ANY (no selection)	50	2,785	5,242,880
DT (single objective)	6	2,085	5,242,880
DT (dual objective)	2	1,439	5,242,880
NB (single objective)	9	2,253	5,242,880
NB (dual objective)	2	1,723	5,242,880
kNN (single objective)	14	2,377	5,242,880
kNN (dual objective)	2	1,723	5,242,880

algorithm as its SW counterpart and presents exactly the same extraction behavior and accuracy hit rate.

The hash function is implemented as combinational logic. Its inputs are the remote IP address, remote port number (for service-based features), and the feature ID. The output is a 16-bit value, which will be used as a row address in the flow table.

The feature flow table is a dual-port RAM with 2^{16} memory locations (rows) and 5 slots per row. Each slot is 16 bits wide, thus the total memory capacity is $2^{16} \times 5 \times 16$, or 5,242,880 bits (5 Mbits).

For each incoming packet, the corresponding row is found and read from the flow table. Then the active slot is updated, according to the update logic for the current feature. For example, the number of received packets is updated by incrementing it by one. The number of bytes received is incremented by the packet's payload size, and so on.

After the updated slot value is calculated, it is replaced in the row read from the table. Only the active slot is changed when a feature is updated; the other four slots keep their previous values. This updated row is one of the two main outputs of the *update_row* block; the other is the actual feature value, used to compose the extracted feature vector that will be the output of the extractor module when the extraction is completed. The extracted feature value is calculated by adding up the value of the four slots currently in use (i.e., excluding the dirty slot).

Once a row has been updated, it is written back to the flow table, and the extraction of a new feature begins. The process repeats until all features have been extracted.

Besides the feature extraction proper, the extractor module includes a timer that manages periodic events. Every 0.5 seconds, the dirty slot column (i.e., all dirty slots in the flow table) must be reset. During this process, the extractor *ready* output is driven low, and one slot is reset every two clock cycles. In total, a column reset takes $2^{16} \times 2 \div (50 \cdot 10^6)$ seconds, or 2.62 ms. During this time, any incoming packet are placed in a buffer for later processing.

The VHDL code of the HW extractor is configurable in terms of which attributes to extract. Therefore, it can be custom-tailored for each classifier to provide only the required attributes, preventing unnecessary computations.

Table 7 presents the resources used by the seven extractors implemented in HW, synthesized for an Altera Cyclone IV GX FPGA. The implementations are named with a prefix (DT, NB, kNN, or ANY) denoting the classifier used with the extractor. The extractor labeled *ANY (no selection)* can be used by any of the three classifiers, because it extracts all 50 features. The *DT (dual objective)* extractor uses the least resources: 1,439 logic cells (LCs) and 5,242,880 memory bits. The extractor for all features, *ANY (no selection)*, uses the most resources: 2,785 LCs and 5,242,880 memory bits. All extractors use the same number of memory bits.

TABLE 8
Area of the Implemented HW Classifiers

Classifier	Logic Cells (LCs)	Memory Bits	9-bit Multipliers
DT (no selection)	106	0	0
DT (single objective)	81	0	0
DT (dual objective)	52	0	0
NB (no selection)	2,471	18,688	14
NB (single objective)	1,327	4,480	14
NB (dual objective)	1,132	0	14
kNN (no selection)	11,327	986,112	14
kNN (single objective)	6,103	279,552	14
kNN (dual objective)	3,891	44,032	14

5.3 Classifiers Implementation in Hardware

Each classifier requires a specific version of the feature extractor, configured to extract only the necessary features. For example, the DT (single objective) extractor (Table 7) extracts the features used by DT (single objective) classifier, and so on. All classifiers were implemented in HW for the three feature selection approaches: no selection, single objective, and dual objective.

The DT classifiers are direct translations of their SW counterparts [41]. Comparators check the feature ranges, and the outputs are combined in a sum-of-products; if the sum is a logic one, the packet is classified as an attack [42].

The NB classifier performs two table lookups per attribute: one to get the probability that the attribute denotes an attack, and another for the probability that it is a normal packet. The classifier serializes the table lookups and calculates the probability of one attribute at a time. When all probability values have been multiplied, a comparator decides whether the input packet is normal or an attack, based on the higher probability value [42].

The kNN classifier keeps 1,000 training samples in a ROM. The circuit calculates the five closest distance values (i.e., $k = 5$) and the corresponding class labels. After all distances have been calculated, the label with the most occurrences is selected as the output [42].

Table 8 presents the HW resources used by the three classifiers. The dual-objective DT is the most compact of all classifiers, requiring only 52 logic cells (LCs). The no-selection kNN classifier uses the most resources: 11,327 LCs, 986,112 memory bits (used to store the kNN training examples), and fourteen 9-bit multipliers. All the classifiers in HW exhibit exactly the same classification behavior and have the same accuracy as their SW counterparts (Table 4).

6 Experimental Results

Because the HW and SW implementations of the extractors and classifiers are functionally equivalent (both implementations always produce the same output for the same input values), it is possible to compare their processing time and energy consumption. The measurements for the SW implementations were performed as described in Section 3.1.

TABLE 9
Throughput of the Extractors

Extractor	SW throughput (packet/s)	HW throughput (packets/s)	HW/ SW ratio
ANY (no selection)	359,531	534,815	1.49
DT (single objective)	913,399	2,925,756	3.20
DT (dual objective)	1,222,514	9,947,571	8.14
NB (single objective)	936,833	2,925,756	3.12
NB (dual objective)	1,265,741	9,947,571	7.86
kNN (single objective)	706,181	1,344,266	1.90
kNN (dual objective)	1,218,642	9,947,571	8.16

TABLE 10
Energy Consumption of the Extractors

Extractor	Energy per extraction in SW (nJ)	Energy per extraction in HW (nJ)	HW/SW (%)
ANY (no selection)	1,999.47	1,078.25	53.9
DT (single objective)	1,110.81	230.48	20.7
DT (dual objective)	965.00	57.90	6.0
NB (single objective)	1,094.96	242.56	22.2
NB (dual objective)	969.17	80.56	8.3
kNN (single objective)	1,267.43	476.10	37.6
kNN (dual objective)	962.70	80.59	8.4

The measurements related to the HW implementations are described in the following topic.

6.1 Power and Throughput Measurements in Hardware

To evaluate the energy consumption and throughput of the HW implementations, we developed a measurement setup based on an FPGA development board (a Cyclone IV GX FPGA Development Kit). To measure the FPGA power consumption, we used Altera's Power Monitor tool, which measures the FPGA consumption using onboard ADCs and sends the results continuously to a PC via JTAG.

$$Energy\ per\ operation\ (J) = \left(P_{running} - P_{idle}\right) * processing\ time, \tag{2}$$

$$Throughput\ (packets/s) = \frac{number\ of\ packets}{processing\ time}. \tag{3}$$

To calculate the energy consumed per operation, we used (2), where $P_{running}$ denotes the FPGA power consumption while the circuit is operating, and P_{idle} denotes the FPGA baseline power. To calculate the throughput, we used (3), which is also valid for SW implementations. The processing time is calculated from the clock frequency (50 MHz for all circuits) and the number of clock cycles required to complete an extraction or classification.

6.2 Comparison of the Implemented Feature Extractors

One way to compare different feature extractors (in SW or HW) is to evaluate their throughput running at maximum speed and to count the number of packets processed per second. This can be accomplished by keeping a number of sample packet headers in memory, and providing the extractor with a new input as soon as it is ready to process the next packet.

Table 9 shows the throughput achieved by the HW and SW implementations of the hash-based extractors, using the minimum feature set required by each classifier. Even though the SW and HW implementations operate at widely different frequencies (1.86 GHz versus 50 MHz), all HW versions are faster than their SW counterparts by factors that vary from 1.49 to 8.16. The main reason for such difference is that the HW implementations were designed for a specific task, and they have less overhead than a general-purpose processing platform. For example, a HW implementation may perform relatively complex operations, such as normalizing or updating a feature value, in a single clock cycle; in SW, however, this operation must be broken down into a series of low-level instructions in the Atom CPU.

To compare the energy consumption in HW and SW, we measured the energy required by each implementation to perform the same basic operation. In the case of feature extraction, we measured the average energy consumed to extract the features from a network packet. Table 10 presents the energy spent by each extractor; all HW extractors use less energy than their SW counterparts. The HW extractor created for use with the dual-objective DT classifier is the most energy-efficient of all implemented extractors, requiring 57.9 nJ to extract one packet—only 6 percent of the energy consumed by the corresponding SW version. Fig. 12 shows another view of the energy consumed (in nJ) by each extractor. The grey bars are HW implementations; the black bars denote SW implementations.

6.3 Comparison of the Implemented Classifiers

We have measured the throughput of all implemented classifiers, in both SW and HW. Because of the differences in classifier algorithms and HW implementations, the classification throughput varies greatly—from around 900 packets/s to 79 million packets/s. Table 11 presents the throughput of the implemented classifiers. Unlike the extractors, not all classifiers are faster in HW: for DT (no selection), DT (single objective), and DT (dual objective), the HW version is faster, whereas for all others the SW version is faster. The main

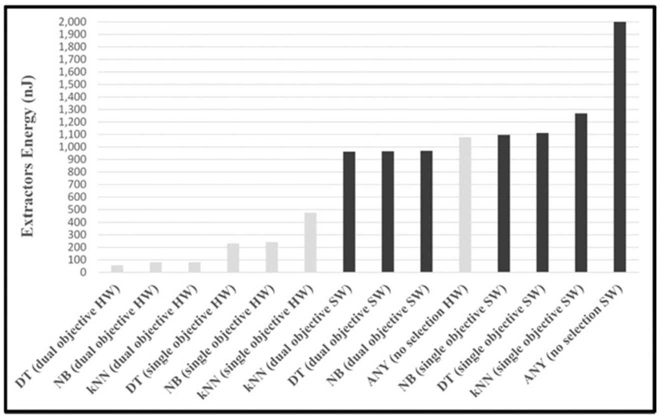

Fig. 12. Energy consumed by the extractors to process one packet, in SW and HW.

TABLE 11
Throughput of the Implemented Classifiers

Classifier	SW throughput (packet/s)	HW throughput (packets/s)	HW/ SW ratio
DT (no selection)	7,687,074	79,170,295	10.30
DT (single objective)	42,854,618	72,632,190	1.69
DT (dual objective)	62,517,739	63,653,723	1.02
NB (no selection)	213,860	213,675	1.00
NB (single objective)	1,761,352	1,190,476	0.68
NB (dual objective)	4,485,460	4,166,666	0.93
kNN (no selection)	3,041	904	0.30
kNN (single objective)	8,318	2,621	0.32
kNN (dual objective)	17,088	7,131	0.42

TABLE 12
Energy Consumption of the Classifiers

Classifier	Energy per classification in SW (nJ)	Energy per classification in HW (nJ)	HW/ SW (%)
DT (no selection)	94.31	0.047	0.05
DT (single objective)	49.32	0.031	0.06
DT (dual objective)	47.85	0.015	0.03
NB (no selection)	2,526.37	100.78	4.0
NB (single objective)	358.90	16.78	4.7
NB (dual objective)	180.30	7.40	4.1
kNN (no selection)	169,743.52	152,094.95	89.6
kNN (single objective)	67,566.69	30,709.27	45.5
kNN (dual objective)	32,787.18	6,440.44	19.6

reason is that the other classifiers are sequential and iterative circuits, operating in a lower clock frequency (50 MHz) than the corresponding SW versions (1.86 GHz).

We have also measured the energy consumption of each classifier in SW and HW. All HW classifiers consume less energy to classify a packet, compared to their SW counterparts. Table 12 presents the energy consumed by each classifier, for a single classification operation. The DT algorithm with dual-objective feature selection in HW is the most energy-efficient of all, requiring 15 pJ to classify one packet—only 0.03 percent of the corresponding SW version. The kNN classifier in HW also requires less energy than its SW counterpart, unlike the result found in our previous work for probing attacks [42]. The main reason is that now we assume that the HW classifiers operate with the maximum throughput from Table 11; this is a reasonable assumption, because if the classifiers were run with a lower throughput, we could lower the clock frequency as well, and the energy consumption of the classifiers would also decrease, roughly proportionally to the operating frequency [41].

Fig. 13 shows another view of the energy consumed (in pJ) by each classifier, for one classification operation. The grey bars are HW implementations; the black bars are SW implementations. The difference between the best HW and the best SW implementations – DT HW (dual objective) and DT SW (dual objective) – is greater than three orders of magnitude.

Considering the entire packet processing (extraction plus classification), the best case in SW – *DT SW (dual objective)* – spends 1,012.85 nJ, whereas the best case in HW – *DT HW (dual objective)* – spends 57.92 nJ. The energy savings in this case were of 94 percent.

7 Related Work

Research on power-efficient network intrusion detection is still in its beginnings, and publications on the subject are still rare. Here we describe the main works available in the literature so far.

In commercial products, features for NIDS are usually extracted using netflow-based solutions. Fprobe [40] is a libpcap-based tool and NProbe [43] a netflow port for embedded systems. It is implemented as a linked list; when a collision happens, another entry is stored in the list. An independent thread checks every flow for inactivity, with a frequency of approximately 1 to 5 minutes for Fprobe and 1 minute for Nprobe.

A netflow-based feature extractor implemented using specialized hardware is used in the Cisco Catalyst 6,500 Series Switch [44]. IP addresses, port numbers, and the protocol type are used as an index into a first lookup table, which provides an address into a second flow data table.

Tran et al [57] developed a block-based neural NIDS in SW and HW using a Cyclone III FPGA. The authors used the DARPA dataset, converting it to the netflow format to obtain a classifier able to analyze packets directly from Cisco routers. The HW implementation was 1,300 times as fast as its SW counterpart, but it was unclear whether the two versions were functionally equivalent. Moreover, the used version of the DARPA dataset was outdated. There are several documented attempts to use netflow to extract intrusion related features [45], [46], [47].

Das et al. [48] developed an FPGA composed of a feature extraction module, that uses a hash table to keep the attributes, and a detection module, using the Principal Component Analysis technique to detect port scan (probing) and syn flood (DoS) attacks. The throughputs achieved for feature extraction and detection were 21 and 23 Gbps, respectively, using a Xilinx Virtex-II XC2V1000. However, it is not clear whether the modifications for implementing the algorithms in HW produced different results from a SW implementation. In addition, the authors used the outdated KDD'99 dataset to validate the proposal.

In the work of Gómez et al. [49] about feature selection, the authors compare a single aggregate objective, using weights for each objective, against multi-objective feature

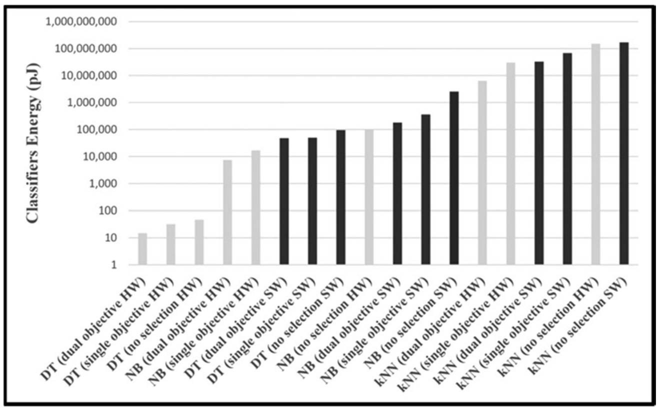

Fig. 13. Energy consumed by the classifiers to process one packet, in SW and HW.

selection. A signature-based IDS was used for the tests, aiming at minimizing the number of features and showing that it can reduce the false-positive and false-negative rates when classifying the DARPA 1998 [50] dataset. It is worth noting that the authors used an outdated DARPA dataset, which is not recommended for use with signature-based IDS due to several limitations [50], [51].

Hoque et al. [52] use NSGA-II as a filter-based feature selection method. The tests showed that as the number of used features increases, the execution time also increases for the kNN and DT classifiers. Feature selection improved the classifier accuracy; however, the authors did not consider the impact that a large number of training instances used in the kNN classifier would have in energy consumption.

Regarding the use of ML classifiers for anomaly-based detection in hardware, Vijayasarathy et al. [53] developed a DoS detection system with an NB classifier in a Virtex 4 FPGA. The authors used the outdated KDD'99 and real-world traffic captured from the "Society for Electronic Transactions and Security" website for training, which does not allow reproducibility. The classifier was first modeled in SW and then implemented in HW, but the authors did not take any measures to ensure that the two versions were functionally equivalent.

In our previous work, we have evaluated DT, NB, and kNN classifiers to detect probing attacks in software and hardware [42]. To allow a direct comparison of the energy efficiency of the two approaches, we ensure that the HW and SW versions of each algorithm have exactly the same classification behavior. The results showed that the most energy-efficient classifier (without considering the feature extraction) is the DT, in both SW and HW. The hardware version of this classifier consumed only 0.05 percent of the energy used in SW version.

There are very few works comparing software and hardware implementations of intrusion detection engines, with an emphasis on energy consumption. Moreover, most HW-based works use Snort rules [54], [55], [56] rather than anomaly-based detection. We have found no previous work in the literature addressing all aspects of the SW and HW implementation of energy-efficient anomaly detection systems.

8 Conclusion

Intrusion detection is usually implemented in SW, making accurate power measurements difficult because they require specialized techniques as described in Section 3.1. Moreover, when comparing SW and HW implementations, it is indispensable to prove that they are functionally equivalent, so that the throughput and power consumption of the two alternatives can be compared directly. In this paper, we have proposed and evaluated three new approaches to improve the energy efficiency of network security algorithms and applications: a new feature extraction algorithm suitable for HW implementation, a feature selection method based on two simultaneous objectives (accuracy and energy consumption), and the implementation of the feature extractor engine and ML classifiers in HW. We have presented detailed energy consumption measurements for all algorithms, in both SW and HW. The new feature extractor consumes only 22 percent of the energy used by a commercial tool, when implemented in SW, and 12 percent when implemented in HW.

The dual-objective feature selection method enabled energy savings of up to 92.9 percent (in the best case, for the NB classifier) in comparison with a classifier without feature selection. Dual-objective feature selection provided an average 9.3 percent energy savings, while incurring an average 0.90 percent accuracy loss, when compared with single-objective feature selection. Overall, comparing the most energy-efficient software implementation (using the proposed feature extraction engine and the Decision Tree classifier) with an equivalent hardware implementation, the hardware version consumed only 5.7 percent of the energy used by the software version.

Acknowledgments

The authors would like to thank to Paulo Cemin for developing the power measurement platform used in the experiments, and Intel Labs Univ. Research Office and the Brazilian National Council for Scientific and Technological Development, grant 440850/2013-4, for the financial support.

References

[1] Symantec Lab. ISTR20: Internet Security Threat Report, https://www.symantec.com/security-center/threat-report, May 2016.

[2] Kaspersky Lab. Kaspersky Security Bulletin 2014: Overall statistics for 2014, securelist.com/analysis/kaspersky-securitybulletin/68010/kaspersky-security-bulletin-2014-overall-statistics-for-2014/, May 2016.

[3] S. Yusuf, W. Luk, M. Sloman, N. Dulay, E. C. Lupu, and G. Brown, "Reconfigurable architecture for network flow analysis," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 16, no. 1, pp. 57–65, Jan. 2008.

[4] P. Lambruschini, M. Raggio, R. Bajpai, and A. Sharma, "Efficient implementation of packet pre-filtering for scalable analysis of IP traffic on high-speed lines," in *Proc. 20th Int. Conf. Softw. Telecommun. Comput. Netw.*, 2012, pp. 1–5.

[5] Embedded Intel solutions, Intel's Hybrid CPU-FPGA, http://www.embeddedintel.com/commentary.php?article=2143, May 2016.

[6] N. B. Guinde and S. G. Ziavras, "Efficient hardware support for pattern matching in network intrusion detection," *Comput. Security*, vol. 29, no. 7, pp. 756–769, 2010.

[7] S. Kim and J.-Y. Lee, "A system architecture for high-speed deep packet inspection in signature-based network intrusion prevention," *J. Syst. Archit.*, vol. 53, no. 5–6, pp. 310–320, 2007.

[8] J. Harwayne-Gidansky, D. Stefan, and I. Dalal, "FPGA-based SoC for real-time network intrusion detection using counting Bloom filters," in *Proc. IEEE Southeastcon*, 2009, pp. 452–458.

[9] K. Hwang, M. Cai, Y. Chen, and M. Qin, "Hybrid intrusion detection with weighted signature generation over anomalous internet episodes," *IEEE Trans. Dependable Secure Comput.*, vol. 4, no. 1, pp. 41–55, Jan.-Mar. 2007.

[10] L. Khan, M. Awad, and B. Thuraisingham, "A new intrusion detection system using support vector machines and hierarchical clustering," *Int. J. Very Large Data Bases*, vol. 16, no. 4, pp. 507–521, 2007.

[11] D. S. Kim, H.-N. Nguyen, and J. S. Park, "Genetic algorithm to improve SVM based network intrusion detection system," in *Proc. IEEE 19th Int. Conf. Advanced Inform. Netw. Appl.*, 2005, pp. 155–158.

[12] C.-F. Tsai, Y.-F. Hsu, C.-Y. Lin, and W.-Y. Lin, "Intrusion detection by machine learning: A review," *Expert Syst. Appl.*, vol. 36, no. 10, pp. 11994–12000, 2009.

[13] Z. Trabelsi and R. Mahdy, "An anomaly intrusion detection system employing associative string processor," in *Proc. Int. Conf. Netw.*, 2010, pp. 220–225.

[14] T. QuangAnh, F. Jiang, and H. Quang Minh, "Evolving block-based neural network and field programmable gate arrays for host-based intrusion detection system," in *Proc. Knowl. Syst. Eng.*, 2012, pp. 86–92.

[15] T. Tuncer and Y. Tatar, "FPGA based programmable embedded intrusion detection system," in *Proc. Int. Conf. Security Inform. Netw. ACM*, 2010, pp. 245–248.

[16] M. Papadonikolakis and C. Bouganis, "A novel FPGA-based SVM classifier," in *Proc. Field-Programmable Technol.*, 2010, pp. 283–286.

[17] A. Das, D. Nguyen, J. Zambreno, G. Memik, and A. Choudhary, "An FPGA-based network intrusion detection architecture," *IEEE Trans. Inform. Forensics Security*, vol. 3, no. 1, pp. 118–132, Mar. 2008.

[18] M. Papadonikolakis and C.-S. Bouganis, "Novel cascade FPGA accelerator for support vector machines classification," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 23, no. 7, pp. 1040–1052, Jul. 2012.

[19] I. Corona, G. Giacinto, and F. Roli, "Adversarial attacks against intrusion detection systems: Taxonomy, solutions and open issues," *Inf. Sci.*, vol. 239, pp. 201–225 Aug. 2013.

[20] R. P. Lippmann, D. J. Fried, I. Graf, J. W. Haines, K. R. Kendall, D. McClung, D. Weber, S. E. Webster, D. Wyschogrod, R. K. Cunningham, and M. A. Zissman, "Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation," in *Proc. DARPA Inform. Survivability Conf. Expo.*, 2000, pp. 12–26.

[21] P. Kabiri and A. A. Ghorbani, "Research on intrusion detection and response: A survey," *Int. J. Netw. Security*, vol. 1, no. 2, pp. 84–102, 2005.

[22] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Proc. IEEE Symp. Security Privacy* , 2010, pp. 305–316.

[23] C. Komviriyavut, T. Sangkatsanee, P. Wattanapongsakorn, and N. Charnsripinyo, "Network intrusion detection and classification with decision tree and rule based approaches," in *Proc. 9th Int. Symp. Commun. Inform. Technol.* , 2009, pp. 1046–1050.

[24] P. Gogoi, M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Packet and flow based network intrusion dataset," *Contemporary Comput.*, pp. 322–334, 2012.

[25] J. J. Davis and A. J. Clark, "Data preprocessing for anomaly based network intrusion detection: A review," *Comput. Security*, vol. 30, no. 6–7, pp. 353–375, 2011.

[26] IETF, Cisco Systems NetFlow Services Export Version 9, http://www.ietf.org/rfc/rfc3954.txt, May 2016.

[27] CISCO, Introducing to CISCO IOS Netflow, http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.pdf, May, 2016.

[28] G. John, R. Kohavi, and K. Pfleger, "Irrelevant features and the subset selection problems," in *Proc. Int. Conf. Mach. Learn.*, 1994, pp. 121–129.

[29] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA, USA: Addison-Wesley, 1989.

[30] L. S. Oliveira, R. Sabourin, F. Bortolozzi, and C. Y. Suen, "A methodology for feature selection using multiobjective genetic algorithms for handwritten digit string recognition," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 17, no. 06, pp. 903–929, 2003.

[31] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[32] D. Bedard, M. Y. Lim, R. Fowler, and A. Porterfield, "PowerMon: Fine-grained and integrated power monitoring for commodity computer systems," in *Proc. IEEE SoutheastCon*, 2010, pp. 479–484.

[33] J. H. Laros, P. Pokorny, and D. DeBonis, "PowerInsight—A commodity power measurement capability," in *Proc. IEEE Int. Green Comput. Conf.*, 2013, pp. 1–6.

[34] V. M. Weaver, M. Johnson, K. Kasichayanula, and J. Ralph, "Measuring energy and power with PAPI," in *Proc. IEEE Int. Conf. Parallel Process. Workshops*, 2012, pp. 262–268.

[35] J. Yan, C. K. Lonappan, A. Vajid, D. Singh, and W. J. Kaiser, "Accurate and low-overhead process-level energy estimation for modern hard disk drives," in *Proc. IEEE Int. Conf. Green Comput. Commun.*, 2013, pp. 171–178.

[36] P. Mell, V. Hu, R. Lippmann, J. Haines, and M. Zissman, "An overview of issues in testing intrusion detection systems an overview of issues in testing intrusion detection," 2003. [Online]. Available: http://csrc.nist.gov/publications/nistir/nistir-7007.pdf. Accessed Oct. 2015.

[37] Honeyd, Honeyd Development, http://www.honeyd.org/, May 2016.

[38] Weka, Weka 3 Data Mining Software in Java, http://www.cs.waikato.ac.nz/ml/weka/, May 2016.

[39] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better Netflow," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, p. 245, 2004.

[40] Fprobe, Fprobe Project, https://sourceforge.net/projects/fprobe/, May 2016.

[41] A. L. França, R. Jasinski, V. A. Pedroni, and A. O. Santin, "Moving network protection from software to hardware: An energy efficiency analysis," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2014, pp. 456–461.

[42] A. L. França, R. Jasinski, P. Cemin, V. A. Pedroni, and A. O. Santin, "The energy cost of network security: A hardware vs. software comparison," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2015, pp. 81–84.

[43] NProbe, An Extensible NetFlow v5/v9/IPFIX Probe for IPv4/v6, http://www.ntop.org/products/netflow/nprobe/, May 2016.

[44] CISCO, Cisco Catalyst 6500 Supervisor Engine 2T: Netflow Enhancements, http://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/white_paper_c11-652021.pdf, May 2016.

[45] Z. Jadidi, V. Muthukkumarasamy, E. Sithirasenan, and M. Sheikhan, "Flow-based anomaly detection using neural network optimized with GSA algorithm," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2013, pp. 76–81.

[46] T. Peng and W. Zuo, "Data mining for network intrusion detection system in real time," *Int. J. Comput. Sci. Netw. Security*, vol. 6, no. 2, pp. 173–177, 2006.

[47] M. Y. Su, G. J. Yu, and C. Y. Lin, "A real-time network intrusion detection system for large-scale attacks based on an incremental mining approach," *Comput. Security.*, vol. 28, no. 5, pp. 301–309, 2009.

[48] A. Das, S. Misra, S. Joshi, J. Zambreno, G. Memik, and A. Choudhary, "An efficient FPGA implementation of principle component analysis based network intrusion detection system," in *Proc. Design, Automation and Test in Europe*, 2008, pp. 1160–1165.

[49] J. Gómez, C. Gil, R. Baños, A. L. Márquez, F. G. Montoya, and M. G. Montoya, "A Pareto-based multi-objective evolutionary algorithm for automatic rule generation in network intrusion detection systems," *Soft Comput.*, vol. 17, pp. 255–263, 2012.

[50] J. McHugh, "Testing Intrusion detection systems: A critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory," *ACM Trans. Inform. Syst. Security*, vol. 3, no. 4, pp. 262–294, 2000.

[51] M. V. Mahoney and P. K. Chan, "An analysis of the 1999 DARPA/Lincoln laboratory evaluation data for network anomaly detection," in *Proc. Int. Symp. Recent Adv. Intrusion Detect.*, vol. 2820, pp. 220–237, 2003.

[52] N. Hoque, D. K. Bhattacharyya, and J. K. Kalita, "MIFS-ND: A mutual information-based feature selection method," *Expert Syst. Appl.*, vol. 41, no. 14, pp. 6371–6385, 2014.

[53] R. Vijayasarathy, S. Raghavan, and B. Ravindran, "A system approach to network modeling for DDoS detection using a naive Bayesian classifier," in *Proc. 3rd Int. Conf. Commun. Syst. Netw.*, 2011, pp. 1–10.

[54] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," in *Proc. 13th Int. Symp. Field-Programmable Gate Arrays*, 2005, pp. 238–245.

[55] T. Katashita, Y. Yamaguchi, A. Maeda, and K. Toda, "FPGA-based intrusion detection system for 10 gigabit ethernet," *IEICE Trans. Inform. Syst.*, vol. E90-D, no. 12, pp. 1923–1931, 2007.

[56] S. Pontarelli, G. Bianchi, and S. Teofili, "Traffic-aware design of a high-speed FPGA network intrusion detection system," *IEEE Trans. Comput.*, vol. 62, no. 11, pp. 2322–2334, Nov. 2013.

[57] Q. A. Tran, F. Jiang, and J. Hu, "A real-time netflow-based intrusion detection system with improved BBNN and high-frequency field programmable gate arrays," in *Proc. IEEE Int. Conf. Trust, Security Privacy Comput. Commun.*, 2012, pp. 201–208.

[58] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection," *Comput. Security.*, vol. 31, no. 3, pp. 357–374, 2012.

Eduardo Viegas received the BS degree in computer science from PUCPR in 2013 and is currently working toward the MS degree at PUCPR. His research interests include machine learning and security.

Altair Olivo Santin received the BS degree in computer engineering from the PUCPR in 1992, the MSc degree from the UTFPR in 1996, and the PhD degree from the UFSC in 2004. He is a full professor of computer engineering at PUCPR. He is a member of the IEEE, ACM, and the Brazilian Computer Society.

André França received the BS degree in electrical engineering from the Federal University of Parana in 2013 and the MSc degree in electrical and computer engineering from the Federal Technological University of Parana in 2015.

Ricardo Jasinski received the BS, MS, and PhD degrees in electrical engineering from the Federal Technological University of Parana in 2000, 2004, and 2014, respectively.

Volnei A. Pedroni received the BSc degree in electrical engineering from UFRGS, in 1975, and the MSc and PhD degrees from Caltech, in 1990 and 1995, respectively. He has been since with the Electronics Engineering Department of Federal Technological University of Paraná State, in Brazil.

Luiz S. Oliveira received the BS degree in computer science from UP, the MSc degree from UTFPR, and PhD degree in computer science from École de Technologie Supérieure, Université du Quebec in 1995, 1998 and 2003, respectively. From 2004 to 2009 he was professor of PUCPR. In 2009, he joined the UFPR, where he is professor at the Department of Informatics.