Piccett: Protocol-Independent Classification of Corrupted Error-Tolerant Traffic

Florian Schmidt, Martin Henze, Klaus Wehrle
Communication and Distributed Systems
RWTH Aachen University, Aachen, Germany
Email: {schmidt,henze,wehrle}@comsys.rwth-aachen.de

***Abstract*—Bit errors regularly occur in wireless communications. While many media streaming codecs in principle provide bit error tolerance and resilience, packet-based communication typically drops packets that are not transmitted perfectly. We present PICCETT, a method to heuristically identify which connections corrupted packets belong to, and to assign them to the correct applications instead of dropping them. PICCETT is a receiver-side classifier that requires no support from the sender or network, and no information which communication protocols are used. We show that PICCETT can assign virtually all packets to the correct connections at bit error rates up to 7–10%, and prevents misassignments even during error bursts. PICCETT's classification algorithm needs no prior offline training and both trains and classifies fast enough to easily keep up with IEEE 802.11 communication speeds.**

I. Introduction

Media content, especially audio and video streaming, has been an increasingly important part of Internet communications. These days, it produces a large part of data traffic worldwide. Many audio and video codecs can, at least in theory, cope with bit errors in the received data, and are more sensitive to packet loss than to (partially) erroneous data [1], [2].

Such bit errors are a typical property of wireless connections, which are much more susceptible to them than the classic wired connections. In recent years, the use of wireless communications has greatly increased. The ubiquitous use of notebooks, smartphones and tablets has increased the portion of devices that communicate wirelessly at least on the last hop. As a result, the problem of bit errors within packets (even after PHY decoding) has become more widespread.

The classical approach to Internet communications, however, has not taken these developments into account. Data is still secured by checksums, and even single bit errors lead to a complete discard (and potential retransmission) of a packet, a very inefficient approach in case of error-tolerant data. Partial solutions to single protocols, such as UDP-Lite [3], have tried to mitigate the problem by introducing tolerance to errors in the application payload. However, these insular approaches are not well-adapted because they interact badly with existing solutions: they need support from both sender and receiver to switch to a new protocol, and their advantages are typically rendered void by the interaction with other protocols within the network stack, which still enforce complete packet integrity.

Finally, even if such a solution worked well, and allowed error-tolerant applications to receive erroneous data instead of discarding and retransmitting, only errors in the payload of packets could be coped with. However, especially in the case of audio streaming, headers form a sizable part of a packet, sometimes more than 50%. A solution that also tolerates header errors can produce further performance improvements.

In this paper, we present PICCETT, a **p**rotocol-**i**ndependent **c**lassifier for **c**orrupted **e**rror-**t**olerant **t**raffic that takes all these considerations into account and has the following properties: (1) It refrains from discarding packets with checksum mismatches, heuristically identifies which connection a packet belongs to, and repairs header errors, thus introducing error tolerance for both packet headers and payload. (2) It allows for concurrent error-tolerant and error-sensitive traffic, that is, it does not produce negative side-effects to error-intolerant traffic such as HTTP or FTP. (3) It resides on the receiving host and does not need support from the sender. (4) It is a general approach that neither focuses on specific protocols nor needs detailed domain knowledge about the protocols it handles.

To reach these goals, we created a classification algorithm that inspects the header and payload content of correctly received packets (as indicated by a correct checksum). It creates patterns from these contents which it then uses to classify packets with bit errors (indicated by checksum mismatches) to find the connection a packet most likely belongs to (property 4). Applications can, on connection setup, indicate that they tolerate bit errors (property 2). If the algorithm chooses such a connection, it will repair packet contents to fit learned values (property 1). All information to do this is locally acquired, and no interaction with the sender is needed (property 3).

We will show that PICCETT assigns erroneous packets to the correct connections at bit error rates (BERs) in excess of 7% while preventing misassignments to wrong connections, all while being fast enough to do both training and classifying in real-time (within a few microseconds) as packets are received, and without needing any prior off-line training.

The rest of the paper is structured as follows. We describe the packet classification algorithm in Section II. In Section III, we explain how this algorithm can be used in a system by implementing it in the Linux kernel. We show evaluation results in Section IV. We discuss potential extensions of the classifier in Section V and related work in Section VI before concluding in Section VII.

II. Packet Classification Algorithm

We will start the description with some basic properties and constraints that the classifier must follow. First, the algorithm needs to produce a high classification accuracy. As many packets as possible should be classified correctly. Even more important, as few packet as possible should be classified incorrectly (misassigned), even at very high bit error rates (BERs). While misassigning a packet to an error-tolerant application is not instantly fatal (it will merely appear to the application as if the packet had an extremely high error rate), it is nevertheless undesirable. Second, to reach this goal, the algorithm should have a discard threshold. Contrary to standard classification algorithms used in Machine Learning, which always assign the input to a class, our algorithm should have a threshold that drops packets that are too different from previously seen ones. Otherwise, extremely corrupted packets which do not well match any connection have a high risk of being misassigned. Furthermore, packets for other network participants could be overheard; they need to be sorted out. Third, the algorithm must be fast. It must be able to both train new classes from new connections (i.e., learn) and classify incoming corrupted packets (i.e., predict) in real-time without slowing down the system. As such, the demands are similar to data stream mining algorithms. As a rough estimate of the speed requirement, a fully utilized 802.11a/g connection at $54\,\mathrm{Mbit/s}$ produces several packets per millisecond, with which the classifier needs to keep up. Fourth, the algorithm should run inside the OS kernel and therefore not use floating-point operations. This is because in many kernels, among them the Linux kernel [4], the floating-point unit is reserved for user-space calculations.

These constraints suggest that popular classifiers such as support vector machines [5] and random forests [6] are unsuitable. Hence, we create a novel classifier tailored to our problem. PICCETT comprises two main parts: the learner and the predictor. The learner sits between the network stack and an application and processes correct packets. At this point, it has access to the full packet content (header and payload), packet size and reception time, and most importantly, knows which application the packet belongs to. It then uses this information to train the classifier, creating one class per connection.

The predictor sits at the bottom of the network stack and processes corrupted packets before protocol handlers process the packet. It compares the (potentially erroneous) content of the packet to the learned classes. It then makes one of three decisions: (1) It assigns a class label of an error-tolerant connection. The packet is then repaired as far as possible (see Section III-C) and passed up the network stack. (2) It assigns a class label of an error-sensitive connection, and drops the packet. Note that the learner processes all correct packets for all applications, not merely for those applications that indicated error tolerance. This is to train a class for each connection and filter out erroneous packets belonging to error-sensitive traffic. (3) It assigns no label, indicating no close resemblance to any ongoing connections, and drops the packet.

To get an idea of the variability of certain bits in network connections, we captured several kinds of data streams, among them audio and video streams with and without RTP, web browsing via HTTP, and ICMP pings. We then analyzed how static bits tended to be across packets, that is, how often within any single stream a certain bit position was 1 and how often it was 0. Our results showed that within the first 46 byte (we truncated the results to the shortest packet lengths in our data set), 66% of the bits either were a static 0 or a static 1, that is, never changed over the course of the connection. A further 23% were almost completely random, with a close to 50%–50% distribution between 1s and 0s in those positions.

training packets	1. packet	0	1	1	0	1	0	0	0
	2. packet	0	1	1	0	1	0	1	1
	3. packet	0	1	1	0	0	1	0	1
learned pattern	$mask$	1	1	1	1	0	0	0	0
	$value$	0	1	1	0	-	-	-	-
scoring packets	score 0.0	0	1	1	0	*1*	*1*	*1*	*1*
	score 0.25	0	**0**	1	0	*0*	*1*	*0*	*1*
	score 0.75	**1**	**0**	**0**	0	*0*	*0*	*0*	*0*

Fig. 1. Example of the scoring algorithm. After learning from three packets, *mask* shows the first four bits to be static, with the static values in *value*. Incoming packets can then be assigned a score. The score of a packet is the ratio of non-matching (static) bits to all (static) bits. Non-matching bits for to-be-scored packets are bold, masked (ignored) bits italic.

This motivated our design of a very lightweight classification algorithm: For each class (i.e., connection), bits are placed into one of two categories. They are either considered static or random. Each class contains a bit field (the *mask*) that denotes which bit positions are static and which are random. A second bit field (the *value*) contains values for the static positions, to keep track of whether those are static 1s or 0s. The initial mask after the first packet processed by the learner for a connection is all 1s, and the initial value equal to the packet. With each additional packet, the packet is compared to the value bitfield, and differing positions are set to 0 in the mask.

The predictor calculates the score by comparing the received bits to the unmasked (i.e., static) value bits. All these operations can be done efficiently with basic binary operations. The score is the ratio of non-matching (static) bits to all (static) bits. An example is given in Figure 1. Whenever a corrupted packet is received, the predictor calculates the score for every class trained by the learner. The packet is assigned to the class with the lowest score, unless that score is above the discard threshold, in which case it is dropped because the risk of misassignment is deemed to high. We will investigate the effects of threshold setting in Section IV-B.

This scoring algorithm already worked very well in most situations. However, we noticed that in dynamic environments, with connections opening and closing, the accuracy decreased whenever a new connections was opened. This is because it takes several packets for the mask to stabilize. After the first packet, the mask would be set to 1 for every bit. Positions with random bits can only be recognized as such on subsequent correctly received packets that the learner uses for training. Those random bits inflate the score even on packets belonging to the connection: the class cannot recognize its own packets properly. The effect is that erroneous packets for new connections have a high risk to be misassigned to older connections. Only the first few packets of each connection are susceptible to this effect (in our experiments, masks typically stabilized within the first five learned packets), but in highly dynamic scenarios with many opening and closing connections, this can lead to undesirably high misassignment rates.

To mitigate this problem, we introduced *combined scoring*. The idea is to compare two classes by specifically testing for those bits that are static and different between them. A combined mask for two classes is created by $mask_{1,2} = mask_1 \text{ AND } mask_2 \text{ AND } (value_1 \text{ XOR } value_2)$. This mask is then used instead of the class's own mask to calculate the

connection 1	mask1	1	1	1	1	0	1	0	0
	value1	1	0	1	1	-	1	-	-
connection 2	mask2	1	1	1	1	1	1	1	1
	value2	1	0	1	1	0	0	1	1
combined mask	cmask	0	0	0	0	0	1	0	0
incoming packet		1	0	1	1	1	0	0	0
score 1	0.2	1	0	1	1	*1*	**0**	*0*	*0*
score 2	0.375	1	0	1	1	**1**	0	**0**	**0**
combined score 1	1.0	*1*	*0*	*1*	*1*	*1*	**0**	*0*	*0*
combined score 2	0.0	*1*	*0*	*1*	*1*	*1*	0	*0*	*0*

Fig. 2. Example of the combined scoring algorithm. A new connection (2) has not learned any random bits yet. The next incoming packet shows a high score towards that connection, preferring connection 1. Applying the combined mask shows a combined score that suggests the packet should not be assigned to connection 1, either.

combined score. This gives a comparative measure between two streams. Figure 2 shows an example of combined scoring. One idea behind this is that, in many protocols, static bits that identify a connection (e.g., ports in TCP/UDP) occur at the same positions, so for connections with (at least some) shared protocols, this is especially beneficial. To reduce the $\mathcal{O}(n^2)$ complexity of the combined scoring (pairwise combination of all streams), we limit its use. Analysis during evaluation showed that a good tradeoff between accuracy and complexity is considering only classes for combined scoring that had scores of 0.2 or less or fewer than 10 learned packets.

The predictor combines both scoring algorithms into a three-step algorithm to decide on how to label a packet:

1) Calculate packet's score for each connection. Remove all connections with a score above a threshold θ_s from consideration.
2) Order connections by increasing score and iterate over them. If for a connection, combined scores with all other connections are below a threshold θ_{cs}, label packet with that connection.
3) If packet has no label or a label of an error-sensitive connection, discard.

III. Kernel Implementation

In order to assess the feasibility of PICCETT in a real-world setting, we prototypically implemented our proposed approach for handling corrupted network packets into the Linux kernel (version 2.6.32). This illustrates (in addition to the feasibility of implementing the classifier in a kernel context) which parts of the network stack need to be modified to facilitate handling of corrupted packets. For this prototype, we decided to start classification at the network layer. This is not due to any fundamental limitations, and in fact, PICCETT could include the link layer protocol into classification. Rather, starting at the network layer in this prototype was done for practical reasons: the MAC protocol of IEEE 802.11 (which, as a widespread wireless technology, is our main focus) requires strict timing, so that time-critical parts of the protocol are done in firmware on consumer hardware. Thus, interception of packets below the MAC layer would require (hardware-specific) firmware programming. In addition to the classification algorithm described in Section II, the prototypical implementation comprises three changes to the network stack: signal packet corruption from the link layer upwards (Section III-A), create packet interception points for learner and predictor (Section III-B), and enabling the passing of corrupted packets up the stack (Section III-C).

A. Link Layer Signaling

To enable recovery of corrupted packets, we have to, as a first step, instruct the network hardware to not discard them. This feature is available in many consumer cards (e.g., by Atheros and Broadcom) and can be activated by setting a flag via the Linux kernel's `mac80211` hardware abstraction interface. While the information whether a packet is corrupted or not is passed with the packet from the hardware, we also need this information later on, in order for the learner, which sits at the very top of the stack at the socket interface with the application, to know which packets to learn from. Hence, we added a flag to the kernel's `sk_buff` structure, which is used to store all information about network packets, to signal whether the link layer checksum matched or not.

B. Interception Points and Signaling

As discussed in Section II, we have to intercept network packets at two distinct points in the network stack: all correct packets have to be intercepted right before they are passed to the application in order to learn packet patterns and all corrupted network packets have to be intercepted before they are passed to the corresponding network layer protocol handler.

For the learner, which resides between the transport layer and the application layer, we leverage the concept of Linux Socket Filters which was derived from Berkeley Packet Filters [7]. The `sk_filter()` method is called by all transport layer protocols after the processing of a packet has finished and it is ready to be passed to the application. By hooking into this filter, we are able to intercept all incoming network packets right before they are passed to the (user space) application in order to feed the learner component of PICCETT.

In order to realize the predictor, we have to intercept all corrupted packets before they are passed to the handler of the specified Internet layer protocol. In the Linux kernel, the task of identifying the correct Internet layer protocol handler is performed by the `__netif_receive_skb()` method. We decided to integrate the predictor here so it can use its repair feature (see Section III-C) to recover from errors in the protocol identifier field of the packet header. Additionally, our requirements laid out in Section I demand coexistence of error-tolerant and error-sensitive traffic. If we simply used our classifier on all incoming packets, corrupted data would be assigned to applications which cannot tolerate this, such as file transfers. To prevent this, we require error-tolerant applications to signal this capability. This means that PICCETT is backwards-compatible: if the application does not signal error tolerance, no corrupted packets will reach it, and packets that the classifier assigns to those applications will be discarded instead of delivered. For this signaling, we use an approach we presented in an earlier paper [8]. For applications to signal error tolerance, we extend the socket interface that is used to open and close connections by an additional flag `SO_BROKENOK` that can be set when the connection is opened. Conversely, we also signal to the application if a packet is corrupted. This information can then be used by the application if desired, for example, to use error concealment algorithms on the contained data. When an application uses the `recvmsg` system call to receive incoming data, it also receives additional ancillary information via message flags. We added the `MSG_HASERRORS` flag to signal data from corrupted packets.

C. Leveraging Predicted Information

With the above changes, we are able to receive corrupted packets and to predict which connection they most probably belong to. However, if we simply pass those corrupted packets to the network stack, the upper layers will most likely discard them as their checksums fail. Hence, additional measures have to be taken in order to leverage the predicted information.

It might seem as if the easiest solution were to simply skip the protocol handlers and directly assign the packet to the predicted socket. However, this comes with two large downsides: First, stateful protocols which react to packet receptions, for example, for flow control or statistics gathering, cannot work properly if some received packets are not processed. Second, the protocol handlers remove the headers from the packet, so that the application receives the payload part of the packet. For static-size headers, it would be possible to learn this size and chop off the headers on corrupted packets. However, for protocols with variable-size headers, this is impossible.

We therefore take a more complicated approach. As a first step, we repair header fields by settings all static bits to the values learned by the classifier (see Section II). However, random bits are not repaired, so checksums in network and transport layer protocols can still be expected to fail. Hence, we disable checksum checks in those protocol handlers. Thus, we can repair errors in some header fields, and have the handlers accept corrupted packets. How the handlers react to residual errors in bits that were learned as random and therefore not repaired depends on the protocols used. Errors in such fields may lead to packet drops; however, we have shown in previous work [8] for IPv4 and UDP that many header fields are unused and their contents ignored even for standard protocol handlers. Hence, we consider this a workable solution, especially due to the fact that many important fields (e.g., IP addresses in IP, port numbers in UDP or TCP) stay static over the course of the connection and are therefore repaired.

IV. Evaluation

The main goal of PICCETT is to classify packets to the correct applications and to prevent misassignments. The performance stems from the classification algorithm as described in Section II, while the implementation details presented in Section III are mostly inconsequential. Hence, we will focus on the classification algorithm in the evaluation. Its classification needs to be fast and lean enough to allow real-time processing in an operating system network stack. From this, the two questions we want to answer in this section follow: how accurate is our prediction, and how fast is it?

A. Experimental Setup

To evaluate the performance of our classification algorithm, we again captured a number of connections of different types. To create a data set (data set 1), we surfed the web, ran a Debian `aptitude` system update, listened to several web radios (HTTP and UDP-based), and watched a number of YouTube videos (via HTTP and RTP). Ancillary traffic, such as DNS, was also captured. In total, this resulted in 214 connections. All data traffic was transmitted over an Ethernet interface, so we can safely assume this data set to be error-free. To achieve comparability, repeatability, and to exactly control bit error rates (BERs), we decided to use the data set as trace instead of manually creating live traffic for every experiment. Thus, we can track assignments and misassignments exactly. Since in our data set the association of every packet with a connection is known, we can easily calculate the algorithm's classification accuracy. In a real system with live testing, this information has to be derived from the packet contents; however, the motivation of our work is to classify packets in which this information is corrupted and unreliable.

We fed the pre-captured traffic with injected errors into our classification algorithm to evaluate its output. For error injection we used a Bernoulli process. Note that this leads to an independent error distribution. While it is well known that bit errors in many real-world wireless systems, such as 802.11, show a bursty tendency [9], we will show performance values for the complete BER range from 0% to 50% (i.e., completely corrupted). The latter in effect models the worst case of an error burst over the whole length of the packet. We examine this wide range to investigate the number of misassignments our classifier produces in worst-case situations.

Especially at high BERs, virtually no packet is error-free. As a point of reference, a BER of 1% produces more than 99.9% packet errors at packet sizes of only 100 bytes. Since we rely on correct packets to learn from, at high BERs no classes could be trained, and it would be impossible to present evaluation results. Hence, we decided to only inject errors into 70% of the packets. The remaining 30% were fed into the classifier without any injected errors. As a side effect, this introduces a certain packet-level error burstiness. Each data point presented in the following graphs is the result of repeating the experiment 20 times. The error bars denote the minimum and maximum observed results.

We also split the data set into streams and randomized the starting time for each one, emulating different surfing behaviors and increasing the number of combinations of concurrent connections. Finally, we also used the same data set as described in Section II (data set 2) to compare performance between the two data sets.

B. Classification Accuracy

The main performance parameter of a classification algorithm is its accuracy in assigning data to the right class. While a typical classification algorithm knows two outcomes (correct and incorrect assignment), our algorithm produces three: correct and incorrect assignment, and no assignment at all. Since we want to make sure that misassignments occur preferably never or at least as rarely as possible, the algorithm has to be parameterized to balance these outcomes. If the discard threshold is too lenient, many misassignments occur. If it is too strict, misassignments are prevented, but many packets that would have been correctly assigned are also discarded.

We therefore investigated a number of thresholds θ_s and θ_{cs} for score and combined score, respectively. Figure 3 presents results for data set 1 and $\theta_s = \theta_{cs} = 0.2, 0.3, 0.4$ as well as a variant of the algorithm where the threshold for discard was the sum of both scores. As the figure shows, all variants of the algorithm produce exceedingly good classification accuracy. As the thresholds get stricter, the number of correct assignments decreases, but even the strictest presented thresholds recover

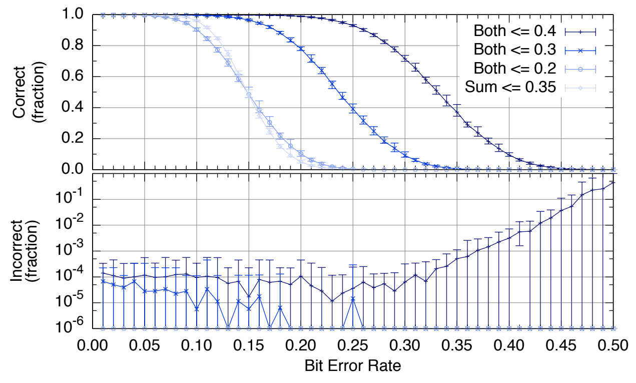


Fig. 3. Classification accuracy for all streams in evaluation data set 1. More lenient thresholds provide high accuracy up to extremely high error rates, but at the cost of misassignments. Stricter thresholds prevent misassignments and still work well at error rates up to 10%. Note linear scale on the top and logarithmic scale on the bottom graph.

almost all packets at BERs above 7–10%. As shown above, this results in extremely high packet error rates (and consequently packet loss in standard systems), and even robust voice codecs show strong quality degradation at such BERs [2]. Regarding misclassification, all variants produce at least reasonably good performance. Even lenient thresholds produce misclassification rates of 10^{-4} to 10^{-3} up to 35% BER. At the stricter thresholds, we witnessed no misclassifications at all, over the whole spectrum of 0%–50% BER. We omit the results for data set 2 for reasons of space and because the results are virtually identical to those of data set 1.

After these very encouraging results, we decided to stress-test the classification algorithm even further. In the next step, we took nine YouTube video streams and let them run concurrently. While this does not constitute realistic user behavior, it gives us a possibility to further investigate the capabilities and limitations of the classification algorithm. Each of the nine streams comprised two RTP, two RTCP, and one RTSP connection. These connections were very similar to the others of the same protocol, in protocols used and protocol header contents (e.g., source and destination IP addresses). The results are presented in Figure 4. The results are roughly similar to the ones presented in Figure 3. Regarding correct classification, the accuracy is somewhat lower, but still allows almost perfect recovery up to BERs of 5–8%, and much higher at more lenient thresholds. Misclassification at lower BERs interestingly is somewhat lower than in the other scenario. We attribute this to the fact that in this setup, the total number of connections is lower than in data set 1, which reduces the number of wrong choices the classifier can make.

Concluding from these results, we suggest the stricter of the investigated thresholds (both ≤ 0.2 or sum ≤ 0.35) for use. At these thresholds, our classification algorithm produces no noticeable misclassification, while producing almost perfect classification in BER ranges that allow practical use of the received data by typical error-tolerant applications.

C. Classification Speed

One of our requirements for the classification algorithm was that it should be fast, to not slow down the receiving system. To investigate this, we measured the time learner and predictor took at different BERs and numbers of streams. The

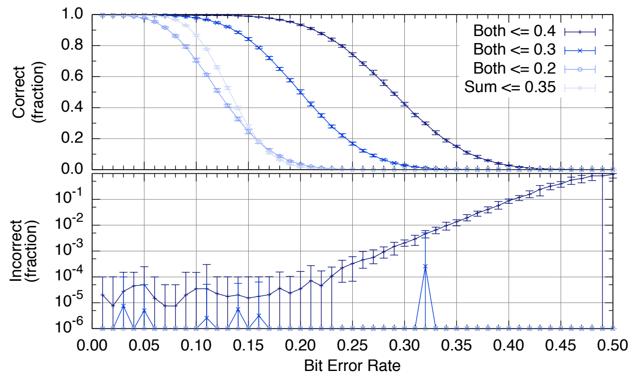


Fig. 4. Classification accuracy for nine concurrent YouTube streams. Even in this extreme scenario with a large number of very similar streams and with strict thresholds, correct assignments show little degradation up to 5–8% BER.

following results used $\theta_s = \theta_{cs} = 0.2$ as threshold and were measured on a PC with an Intel Core 2 Duo CPU at 2.66 GHz. The measurements only used a single core at a time.

Figure 5 shows the processing time the learner takes per packet. BER does not change these results. The time increases linearly with the number of open connections. The chief contributor is the data format in which the classifier saves information about streams. The current implementation uses a linked list, so it takes linear time to look up the right stream to save the learned data to. The time could be further reduced by a different lookup mechanism, for example, by extending the kernel's socket structure to keep a pointer to the right data element, leading to constant lookup time. However, the processing time is so low even now (on the order of nanoseconds) that performance is not impacted.

The predictor has to perform significantly more computations than the learner. While the latter only has to update mask and value for a connection, the predictor has to compare the incoming data to all open connections. This means that lookup time in the linked list is overshadowed by these additional operations. As Figure 6 shows, the prediction time is on the order of microseconds. However, this still does not constitute a bottleneck for typical connections. For example, the interframe space time alone (to acquire medium access) in IEEE 802.11 is a minimum of 28 µs. Two results are of note: First, processing time decreases with increasing BER. This is because at higher BERs, more connections are ruled out as potential receivers during step 1 of the algorithm. (Note that the results for BER=0 are artificial, because if no errors occurred, no prediction would be done.) Second, even though the algorithm has a potential complexity of $\mathcal{O}(n^2)$, in practice, the processing time increases linearly. This is because θ_s typically removes most connections from consideration for step 2 of the algorithm.

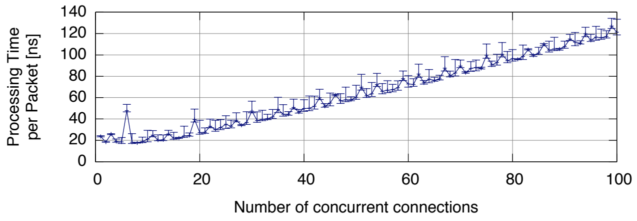


Fig. 5. Processing time in the learner is on the order of nanoseconds. It increases linearly with number of open connections.

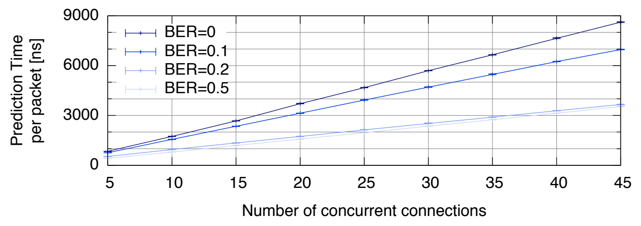


Fig. 6. Processing time in the predictor is on the order of microseconds. It increases linearly with number of open connections. Processing time decreases as BER increases because at high BERs, more streams are ruled out as candidates during step 1 of the classification algorithm.

Overall, the results show that our classification algorithm is fast enough to be of practical use in packet classification.

V. Excursus: Exploitation of Further Packet-Specific Information

During the development of the classifier, we investigated further input that is not part of the packet content, but specific to every packet. Specifically, each packet has a size and an inter-arrival time (IAT), that is, the time that has passed since the reception of the previous packet for that connection. Because these are not part of the packet content, we termed these properties *extrinsic information*. The motivation behind using this information comes (apart from the goal of using as much information as possible to potentially improve classification accuracy) from the observation that in media streaming these properties sometimes show very specific patterns. Especially in live audio streaming or telephony, data needs to be sent out fast enough to not introduce any noticeable delay, so it is common to see packets of the same size arriving with very regular inter-arrival times (a typical value is every 20 ms [10]).

We therefore investigated our data sets for exploitable patterns in packet size and inter-arrival time that identify each connection. Regarding size, we expected to see very static packet sizes for streaming applications. However, we soon noticed that this assumption is too simplistic. Even in audio streams, many codecs do not employ completely static bit rates. Switching between different bit rates creates different packet sizes. Furthermore, during periods of silence, no packets or only very small ones indicating the silence are transmitted. For video streams, the situation is even more complicated: even individual frames are often too large to fit into single packets. Therefore, the data is fragmented into several packets, some with the maximum permissible size, and the last one being of unpredictable size. Therefore, these connections will not show any preference to specific packet sizes – except for a large number of maximum-size packets, which all those connections share, which is not conducive to classification.

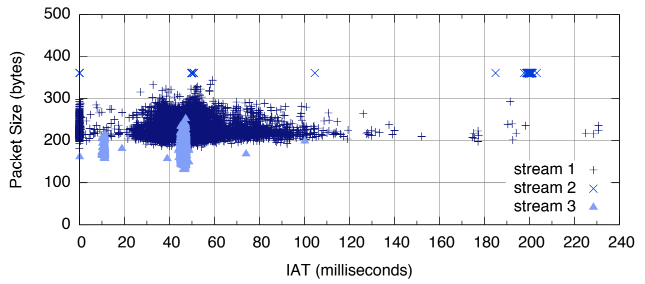
Similar problems arise with the exploitation of packet inter-arrival times (IATs). If video data does not fit into a single packet, the data is transmitted in a burst of many packets within a short time frame, which leads to extremely low IATs. Only the first packet of such a burst will show the "real" IAT to the last packet of the last burst. This leads to at least two data clusters: one with the connection's characteristic IAT, and another close to 0, which is common to all such connections. But even audio streaming does not necessarily produce regular IATs. This is due to several effects.

First, if the audio stream is not a live stream, then pre-buffering leads to packet bursts similar to the ones in the video streaming case. Furthermore, in non-live streaming cases, the application can refill the buffers with intermittent bursts of data packets, instead of relying on regular transmissions.

Second, even in the live streaming case, network effects can significantly alter IATs. Packets can be lost on the way between sender and receiver, for example, due to congestion or interference. In this case, the IAT as observed by the receiver is twice the normal IAT. In case of multiple consecutive losses, this increases further. Furthermore, intermittent congestion can lead to delayed packets that further skew IATs. This well-known effect of jitter is one of the reasons that streaming applications employ buffers.

These problems are highlighted by the graphic representation given in Figure 7. We took three audio and video streams each out of our data set 1 and show for each packet the size and time since the last packet (IAT) in a two-dimensional space. Except for one audio stream, the regions of all streams significantly overlap each other. Note the IAT jitter, even though the data was collected over a switched Ethernet link. In a shared medium, such as in wireless communications, the jitter can be expected to be even stronger. We therefore see no way how this information could be used to classify packets on its own. At best, we can envision using it as a sort of pre-filter for our content classifier described in Section II. Returning all possible connections for a given packet's size and IAT could replace the content classifier's θ-thresholds. This could result in faster processing without losing accuracy.

However, there is an important tradeoff in that employing the size/IAT classifier would take processing time itself. Furthermore, we have shown in Section IV that the content classifier is already both fast and accurate without this additional help. Finally, preliminary experiments showed that



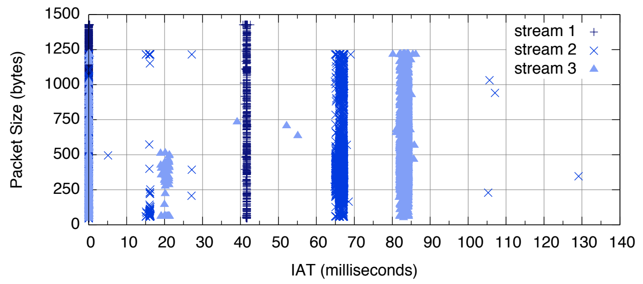


Fig. 7. An example of the problem of exploiting packet size and inter-arrival time (IAT): Two-dimensional representation of packet size and IAT for three audio (left) and video (right) streams each (note the different scales). Except for one audio stream, all streams overlap each other, suggesting problematic separability.

training of the size/IAT classifier to return accurate results takes significantly more packets (and hence time) than the about 5 packets it takes the content classifier. During this time, the filter will produce unreliable results.

All of these factors combined mean that for the time being, we decided against using packet size and IAT as input for our classifier. There are several significant problems in using this information to create or support a highly accurate packet classifier. Nevertheless, if these problems could be solved, we envision the use of such information to be helpful in further improving packet classification.

VI. Related Work

Related work can be roughly split into two areas: work that improves retransmission concepts, but enforces total correctness, and work that tolerates errors in packet payloads and sometimes headers.

Maranello [11] uses partial checksums to facilitate partial retransmissions of corrupted packet areas. ZipTx [12] sends additional Reed–Solomon code data in case a packet contained errors. Other solutions require additional support from the PHY layer (collaborative decoding on several nodes in SOFT [13]), retransmissions to reconstruct packets from (ZigZag [14]), or both (MRD [15]). All these either require PHY support or change protocol behaviors fundamentally. TVA [16] requires neither, but focuses on only correcting specific predefined error patterns.

When it comes to tolerating errors, the best-known solution is UDP-Lite [3]. However, as a new protocol, it requires support from both communication partners. UDP-Liter [17] stays compatible to UDP while tolerating payload errors, but not header errors. Error tolerance for protocol headers has been implemented for specific protocols, such as RTP [18], UDP and IP [8], and suggested for IEEE 802.11 MAC [19] and TCP [20]. These works leverage specific knowledge about header fields in the investigated protocols. In contrast, PICCETT does not require any such information to tolerate and recover from header errors; it merely needs a way to disable packet discarding due to errors in the protocol handlers.

VII. Conclusion

We presented PICCETT, a novel scheme that identifies the connection a network packet belongs to, even in the presence of payload *and* header errors. If an application has signaled error tolerance, such packets will then be assigned to the connection. Thus, it allows error-tolerant application to benefit from receiving partially corrupted packets, while not negatively influencing error-sensitive applications. PICCETT needs no prior training or information about the protocols whose headers it classifies. Without any off-line training, PICCETT both trains (learns) and classifies (predicts) with a speed per incoming packet that is well below packet inter-arrival times in IEEE 802.11, facilitating real-time performance. The classifier can correctly assign virtually all packets up to bit error rates in excess of 7%, while preventing misassignments.

We identify the following aspects as future work. (1) While our results with a set of static θ_s and θ_{cs} have shown very satisfactory performance, dynamically adapting the threshold could optimize both classification accuracy and speed. (2) Investigation into creating a fast and accurate classifier for packet sizes and inter-arrival times could replace the static thresholds and likewise improve classification performance.

Overall, we consider the work presented in this paper a feasible, fast, and very generally applicable approach to introducing support for error tolerance into the network stack.

Acknowledgment

We thank Matthias Lederhofer for fruitful discussions and prototypical implementation of the classification algorithm. This research was funded in part by the DFG Cluster of Excellence on Ultra High-Speed Mobile Information and Communication (UMIC).

References

[1] F. Hammer, P. Reichl, T. Nordström, and G. Kubin, "Corrupted Speech Data Considered Useful: Improving Perceived Speech Quality of VoIP over Error-Prone Channels," *Acta acustica*, vol. 90, no. 6, Dec. 2004.

[2] S. Nguyen, C. Okino, L. Clare, and W. Walsh, "Space-Based Voice over IP Networks," in *Proc. Aeroconf*, Mar. 2007.

[3] L.-Å. Larzon *et al.*, "The lightweight user datagram protocol (UDP-Lite)," RFC 3828, Jul. 2004.

[4] L. Torvalds. (2004, May) LKML archive: How to use floating point in a module? [Online]. Available: https://lkml.org/lkml/2004/5/31/5

[5] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1995.

[6] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[7] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proc. USENIX Winter Technical Conference*, 1993.

[8] F. Schmidt, M. H. Alizai, I. Aktaş, and K. Wehrle, "Refector: heuristic header error recovery for error-tolerant transmissions," in *Proc. ACM CoNEXT*, Dec. 2011.

[9] B. Han *et al.*, "All Bits Are Not Equal – A Study of IEEE 802.11 Communication Bit Errors," in *Proc. IEEE INFOCOM*, Apr. 2009.

[10] ETSI EN 301 704 V7.2.1 (2000-04), *Adaptive Multi-Rate (AMR) Speech Transcoding (GSM 06.90 Version 7.2.1 Release 1998)*, Apr. 2000.

[11] B. Han *et al.*, "Maranello: practical partial packet recovery for 802.11," in *Proc. NSDI*, Apr. 2010.

[12] K. C.-J. Lin, N. Kushman, and D. Katabi, "ZipTx: Harnessing partial packets in 802.11 networks," in *Proc. ACM MobiCom*, Sep. 2008.

[13] G. R. Woo, P. Kheradpour, D. Shen, and D. Katabi, "Beyond the bits: cooperative packet recovery using physical layer information," in *Proc. ACM MobiCom*, Sep. 2007.

[14] S. Gollakota and D. Katabi, "ZigZag decoding: Combating Hidden Terminals in Wireless Networks," in *Proc. ACM SIGCOMM*, Aug. 2008.

[15] A. Miu, H. Balakrishnan, and C. E. Koksal, "Improving loss resilience with multi-radio diversity in wireless networks," in *Proc. ACM MobiCom*, Aug. 2005.

[16] T. Mandel and J. Mache, "Practical error correction for resource-constrained wireless networks: Unlocking the full power of the CRC," in *Proc. ACM SenSys*, Nov. 2013.

[17] P. P.-k. Lam and S. C. Liew, "UDP-Liter: an improved UDP protocol for real-time multimedia applications over wireless links," in *Proc. ISWCS*, Sep. 2004.

[18] F. Schmidt, D. Orlea, and K. Wehrle, "A Heuristic Header Error Recovery Scheme for RTP," in *Proc. IEEE WONS*, Mar. 2013.

[19] W. Jiang, "Bit Error Correction without Redundant Data: a MAC Layer Technique for 802.11 Networks," in *Proc. IEEE WiOpt*, Apr. 2006.

[20] S. Alfredsson and A. Brunstrom, "TCP-L: Allowing bit errors in wireless TCP," in *Proc. IST Mobile and Wireless Summit*, 2003.