FastFE: Accelerating ML-based Traffic Analysis with Programmable Switches

Jiasong Bai°, Menghao Zhang°, Guanyu Li°, Chang Liu°, Mingwei Xu°, Hongxin Hu†
°Tsinghua University †Clemson University

ABSTRACT

Modern traffic analysis applications are usually designed to identify malicious behaviors by inferring sensitive information with machine learning (ML) techniques from network traffic, and they are of great importance to security with the growing use of encryption and other evasion techniques that make classic content-based analysis infeasible. However, with the soaring throughput of networks reaching hundreds of Gbps, it becomes more and more challenging for traffic analysis applications to keep up with today's high-speed large-volume network traffic. In particular, existing feature extractor components in traffic analysis are suffering from undesirable communications, storage, and computation bottleneck. To this end, this paper presents FastFE, a high-speed feature extractor that leverages the capability of new-generation programmable switches to generate desired traffic features flexibly and efficiently. We provide a set of general, easy-to-use, and expressive interfaces for operators to express which traffic features they desire, and a policy enforcement engine that can effectively translate these policies into underlying primitives in programmable switches and commodity servers. Our case study on a state-of-the-art ML-based traffic analysis application, Kitsune, demonstrates the significant advancement of FastFE and its low overheads. As an ongoing work, we are working on a full prototype design and implementation, and hope FastFE can serve as a crucial build block for future ML-based traffic analysis applications.

CCS CONCEPTS

• **Networks** → *Programmable networks*; *Network security*.

KEYWORDS

Traffic Analysis; Programmable Switches; Scalability.

ACM Reference Format:
Jiasong Bai, Menghao Zhang, Guanyu Li, Chang Liu, Mingwei Xu, Hongxin Hu. 2020. FastFE: Accelerating ML-based Traffic Analysis with Programmable Switches. In *ACM SIGCOMM 2020 Workshop on Secure Programmable Network Infrastructure (SPIN 2020) (SPIN '20), August 14, 2020, Virtual Event, NY, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3405669.3405818

SPIN '20, August 14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-8041-6/20/08…$15.00
https://doi.org/10.1145/3405669.3405818

1 INTRODUCTION

Traffic analysis [23] refers to the class of applications that infer sensitive information to identify malicious behaviors from network communication patterns, and they are transitioning to use machine learning (ML), especially deep learning, to achieve a better accuracy [18, 22]. These applications are becoming more and more important to network security with the prevalence of encryption and other evasion techniques that make traditional payload-based analysis of network traffic infeasible. By using traffic analysis techniques, network operators can identify cybercriminals (e.g., botmasters) that proxy their attack traffic via compromised machines or public relays in order to conceal their identities [13, 28, 31, 34, 36], pinpoint individuals engaged in illegal activities within privacy technologies such as ToR [11, 27, 32], detect covert channel attacks that exfiltrate confidential information from compromised machines [4, 9], and prevent malicious activities that intrude the network [18].

In a common traffic analysis application [23], there are usually two typical components: a *feature extractor* component that extracts necessary traffic features (in the form of *feature vectors*) from raw network traffic, a *behavior detector* component that leverages machine learning algorithms to detect the desired network behaviors. For instance, in Kitsune [18], a neural network based network intrusion detection system, there is first a feature extraction framework, which extracts 115-dimension traffic feature vectors with incremental statistics over a damped window, followed by an online detection algorithm, an ensemble of autoencoders, which takes feature vectors as input to detect abnormal packets that have high root mean squared error (RMSE) values. As another example, a website fingerprinting approach on ToR [22], extracts a set of features from raw network traffic, and uses a k-NN classifier to identify which website an individual accesses.

One major challenge for traffic analysis applications is to scale to today's high-speed large-volume network traffic. With the dramatic increase of the network traffic and network bandwidth (e.g., from multi-10s of Gbps to multi-100s of Gbps), there is a growing performance gap for existing traffic analysis applications. While we have seen various solutions (e.g., via GPUs [6], TPUs [15]) to accelerate ML-based behavior detectors in these applications, the performance of feature extractors has not caught up. Existing feature extractors [13, 18, 32] usually use port mirroring to duplicate the collected network traffic, and leverage a large set of servers to store these large volumes of network traffic and extract the desired traffic features, which inevitably impose enormous communication, storage and computation overheads. For instance, a data center with 100,000 servers may require another 40,000~50,000 servers just to keep up with the flood of network traffic, let alone the extra bandwidth to steer these packets [25]. Latest proposal [23] alleviates this by compressing traffic features with compression algorithm, i.e.,

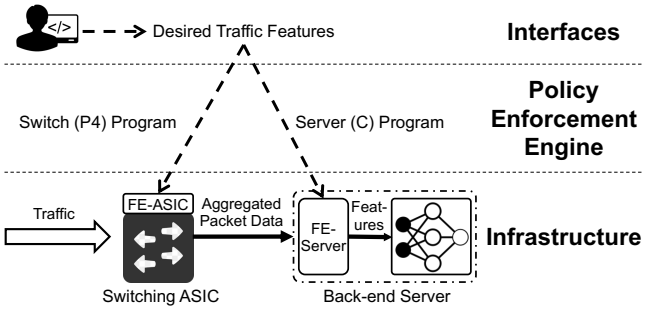

Figure 1: FastFE architecture.

linear projection algorithms, and demonstrates that it is possible to achieve the same accuracy by performing traffic analysis operations only on such compressed features, instead of on raw traffic features. Nevertheless, the performance gain still lags behind changes in the volume of network traffic.

To address these problems, in this paper, we present FastFE, a high-speed feature extractor that enables the generation of desired traffic features flexibly and efficiently, which allows ML-based traffic analysis applications to use feature extraction as a service. By leveraging the capability of new-generation programmable switches, FastFE is able to achieve substantial saving in communications, storage and computation. FastFE also enables unprecedented opportunities to conduct traffic analysis in an online and performant manner, which is important to mitigate malicious network activities timely. Besides, by taking advantage of low unit packet processing cost of programmable switches, FastFE provides network operators a strong incentive to deploy such a defense mechanism.

Figure 1 shows the design of FastFE: it provides a set of high-level interfaces that can help network operators express which traffic features they desire, without reasoning about where and how to execute the feature extractor. It also has a policy enforcement engine that can efficiently translate these policies into underlying primitives that run on programmable switches and commodity servers. We port our FastFE to a state-of-the-art ML-based traffic analysis application, Kitsune, and preliminary evaluations demonstrate FastFE can accelerate the procedure of feature extraction significantly, with only minor overheads. We hope FastFE can become a basic build block for future ML-based traffic analysis applications.

In summary, this paper makes the following contributions:

- We show the scalability issue of ML-based traffic analysis applications in dealing with today's high-speed large-volume network traffic, and identify the bottleneck existing in these applications (§2).
- We propose FastFE, a high-speed feature extractor that enables operators to generate traffic features as they desire. FastFE provides a set of policy interfaces for operators to express the traffic features flexibly, and a policy enforcement engine that enforces these policies in programmable switches and servers efficiently (§3).
- We apply our approach to a state-of-the-art ML-based traffic analysis application, Kitsune, and conduct preliminary evaluations to show the advantages of FastFE in addressing this scalability problem (§4).

Finally, we discuss some related works in §5, and conclude this paper with our ongoing explorations in §6.

2 BACKGROUND AND MOTIVATION

In this section, we first give more background on ML-based traffic analysis applications, then discuss existing systems and their limitations in detail.

2.1 Background on Traffic Analysis

Modern traffic analysis applications usually use machine learning techniques to infer sensitive information from network traffic characteristics, and they are becoming especially useful in the emerging scenarios where encryption and other content evasion techniques do not allow one to inspect packet payload. There are already explicit definitions and comprehensive surveys for traffic analysis [23], and we supplement a few more below.

Botnet detection. Botnets are a severe threat to organization networks, and they are becoming more difficult to be taken down with the emergence of decentralized P2P and stealthy communications. To mitigate this, researches have proposed to identify bots through analysis of packet sizes and packet time intervals [20]. These techniques can help network administrators prevent and mitigate stealthy botnet threats effectively.

Website fingerprint. Privacy enhancing technologies such as VPNs and ToR enable attackers to hide their source/destination IP addresses and the content of the visited websites via encryption, which brings authorities much more difficulty for accountability. Fortunately, website fingerprinting technologies make it possible to identify which websites attackers access by collecting necessary traffic features and feeding them to machine learning algorithms [11, 27, 32]. This may help authorities to pinpoint individuals engaged in illegal activities.

Covert channel detection. Attackers can exfiltrate confidential information from compromised machines in organizations through timing covert channels, without being detected by classic firewalls and intrusion detection systems. Nevertheless, operators can uncover stealthy communication patterns by capturing packet time intervals and analyzing them with machine learning techniques [4, 9]. This is invaluable to protect the property of assets in organizations.

Intrusion detection. Network intrusion detection systems are of great importance to monitor traffic for malicious activities. State-of-the-art intrusion detection systems extract contextual features from network traffic, and use machine learning algorithms to differentiate between normal and abnormal traffic patterns [18]. This is crucial to guarantee the security of the protected networks.

2.2 Problem Statement

As we can see above, traffic analysis applications are widespread in the security community and they are playing a more and more crucial role in identifying a wide variety of cybercriminals. However, existing traffic analysis applications suffer from a severe scalability issue, and cannot catch up with the dramatic increase of network bandwidth and network traffic, which incurs considerable communication, computation and storage overheads.

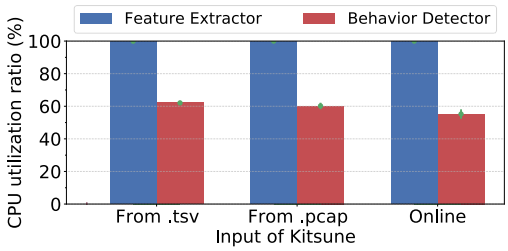

Figure 2: Resource consumption of different components in Kitsune.

To illustrate this, we conduct an experiment on Kitsune [18], a neural network based network intrusion detection system, and show the resource consumption of different components. We modify the original code of Kitsune and divide it into a feature extractor component and a behavior detector component accordingly, where each runs on a separate CPU core as a process. In our experiment, we gradually increase the packet reading rate of Kitsune, and measure the CPU utilization of the two components to pinpoint the system bottleneck. The experimental results are shown in Figure 2, where Kitsune fetches packets from three different ways. As we can see from this figure, no matter which input mode is selected, the CPU utilization of the behavior detector is only about 60% when that of the feature extractor reaches 100%. This indicates that the feature extractor is the bottleneck component of Kitsune. Furthermore, this phenomenon will become much worse when the machine learning algorithms in the behavior detector are accelerated with GPUs/TPUs. All these results illustrate that the current feature extractor is the bottleneck component in traffic analysis applications.

3 FASTFE DESIGN

In this section, we give the detailed design of FastFE: a *policy interface* that can help operators express which traffic features they desire, without reasoning about where and how to execute the feature extractor (§3.1); a *policy enforcement engine* that can efficiently translate these high-level policies into the underlying primitives that run in programmable switches and commodity servers (§3.2).

3.1 Expressing FastFE Policy

FastFE provides a series of interfaces to help operators write their own feature extraction programs, which explains how the required feature vectors are generated from raw packets. At first glance, different traffic analysis applications seem to adopt different ways to generate their feature vectors, as they require different protocol semantics and traffic characteristics, and also apply to different application scenarios. However, we observe that these different traffic analysis applications share a common feature extraction procedure: selecting interested traffic, grouping traffic into multiple sets, generating and updating intermediate variables, producing and composing the final feature vectors. This commonality indicates that an opportunity exists to design such a set of general, easy-to-use and expressive interfaces: filter, groupby, update, and produce. Such a sequence of interfaces are defined as a *process* for feature extraction. In some cases, one application may need several different ways to compose a complete feature vectors, thus operator can also specify multiple processes to achieve their intents. In a typical process, these interfaces are usually organized in this sequential order, and each interface takes streams as inputs and produces streams as outputs. The tuples in the stream only contain packet header fields and some required metadata (e.g., packet arriving timestamp) initially (represented as pktstream), and operators can append new fields into stream in the program subsequently. A brief description of these interface is summarized in Table 1.

Table 1: Summary of FastFE interfaces.

Construct	Description
pktstream	Raw packet stream
filter(R, pred)	Filter stream R with predicate pred
groupby(R, [fields])	Partition stream R into several groups by fields
update(R, func)	Call function func to update variables with tuples in R
produce(R, func)	Call function func to calculate several dimensions of feature vectors from variables and tuples

Variable declaration. Stateful information is required in most feature extractors, therefore we provide stateful variables in FastFE. There are two kinds of variables supported by FastFE, i.e., numerical variables and list variables, and the declarations should be placed at the beginning of the program. The former is used to record a one-dimension packet statistic, where the arriving packet metadata can be calculated by the current variable with a given expression. The latter is used to store multiple numerical variables, responsible for recording packet metadata which cannot be merged (e.g., a set of arrival timestamps). Besides, FastFE variables have a strict bit width, defined upon initialization. In particular, list variables can only contain numerical variables with the same width. For example, numeric<16> count = 0 defines a 16-bit width numeric variable count, and list<32> timestamps = [] describes a list variable where the width of each element is 32 bits.

FastFE sets an explicit scope restriction on variables that each variable can be only used in a single FastFE process. Inside a process, the packet stream is partitioned into multiple groups, and each group possesses a separate replica of variables. Take the program in Figure 3 as an example, the program only defines a single process which partitions packets according to their source IP addresses, so packets with different IP addresses are going to modify different replicas of three variables defined in the beginning. Whenever a new group is created in a process, all the corresponding variables are initialized simultaneously.

Traffic filtering. Since not all packets are required to construct the feature vectors, FastFE provides the filter interface to prevent unrelated packets from further processing. Its parameter pred is a predicate expressing the filtering conditions. This interface can be omitted if no packet should be filtered. For example, following statement only allows TCP traffic to pass.

```
R1 = filter(pktstream, ip.proto == TCP)
```

```
1  numeric<16> w = 0
2  numeric<32> ls = 0
3  numeric<32> ss = 0
4  R1 = groupby(packetstream, [packet.SrcIP])
5  R2 = update(R1, func_update)
6  R3 = produce(R2, func_produce)
7  def func_update([w, ls, ss], [packet.size]):
8      w = (w + 1) * LAMBDA
9      ls = (ls + size) * LAMBDA
10     ss = (ss + size*size) * LAMBDA
11 def func_produce([], [w, ls, ss]):
12     emit([w, ls/w, sqrt(|ss/w-(ls/w)^2|)])
```

Figure 3: Expressing a portion of Kitsune feature extractor in FastFE.

Traffic aggregation. Features are usually extracted upon grouped packets (e.g., 5-tuple), and the groupby interface is designed to define such a granularity to partition packet stream. The argument of this interface is a set $S \in P(T)$, where P(T) stands for power set of the packet tuple. When S is empty, no aggregation is conducted upon traffic. For example, the following statement aggregates packets in R1 stream by 5-tuple.

R2 = groupby(R1, [5tuple])

Variables update. The update interface defines how to update stateful variables using packet tuples, which usually works behind filter and groupby in one process. Parameter func of this interface is a function pointer, pointing to a function implementing the concrete updating logic. The function has two arguments, a list of modified state variables and a list of needed incoming tuple fields. Operators can flexibly express their variables update logic with various calculative operators. Take the following statement as an example, it updates variable count for each incoming tuples in R2 stream. Combining with filter and groupby statements listed above, the following statement records the length of each TCP flow.

```
R3 = update(R2, func_update)
def func_update([count], []) :
    count = count + 1
```

Specially, if operators want to use some variables as the original tuples in packet stream, they should use the append command to add variables to the packet tuples.

Feature production. The produce interface locates at the tail of one process, which is responsible for building the final features vectors from incoming tuple and state variables. Similar to update, produce also has a function pointer argument, which refers to the function implementing computational logic, and the function also takes two lists as arguments. Since a process only builds a sub-vector of features, produced values need to be transmitted to and recorded on the global vector. FastFE supplies an emit command to send the values. For processes which do not produce features, this interface can be omitted. We take the following statement as an example to illustrate the usage of produce. The statement transmits variable count every minute.

```
R4 = produce(R3, func_produce)
def func_produce([last_transmit], [count, packet.ts]) :
    if packet.ts − last_transmit ≥ 60 :
        last_transmit = packet.ts
        emit([count])
```

```
1  numeric<16> flow_length = 0
2  numeric<16> num = 0
3  numeric<32> last_time = 0
4  R1 = groupby(packetstream, [5-tuple])
5  R2 = update(R1, update_length)
6  def update_length([flow_length], []):
7      flow_length = flow_length + 1
8      append(flow_length)
9  R3 = filter(R2, proto == TCP)
10 R4 = groupby(R3, [srcIP])
11 R5 = update(R4, update_num)
12 R6 = produce(R5, func_produce)
13 def update_num([num], [flow_length]):
14     if flow_length == 1:
15         num = num + 1
16     if flow_length == 4:
17         num = num - 1
18 def func_produce([last_time], [num, ts]):
19     if ts - last_time >= 60:
20         emit(num)
21         last_time = ts
```

Figure 4: Expressing TCP flow number querying in FastFE.

All processes in the program together build the complete feature vectors. FastFE maintains a global vector to store features, sends the vector to backend system when it is complete, and clears the global vector to build next feature vectors.

Example. We take two examples to illustrate the FastFE program format. Figure 3 shows a FastFE program expressing a part of Kitsune feature extraction [18], which has only one process to extract mean value and standard deviation of packet size from SrcIP-aggregated network traffic. The program defines three numerical variables (lines 1-3), updates them with packet count and size (lines 7-10, LAMBDA is a constant), and builds features from these variables (lines 11-12). Since Kitsune does not filter any packet, filter is omitted in this program.

The program shown in Figure 4 counts the number of TCP flows whose packet number is less than 4 for each source IP address, and emits the count every minute as a dimension of feature vector [8]. The program defines three numeric variables (lines 1-3) at first to store the packet number of each flow (flow_length), the number of flows satisfying the condition (num) and the last sending time of the feature (last_time). Since the flow length cannot be obtained directly, the program uses a process to compute the length of each flow (lines 4-5), and adds the length to packet tuples in the function for the following process (line 8). In the second process, the program updates the number of required flows according to flow lengths (lines 9-11), and sends the number every minute (line 12).

3.2 Policy Enforcement Engine

In this part, we describe how to translate a FastFE program into concrete codes deployed in programmable switches and commodity servers. To achieve higher scalability, our principle here is to let high-speed programmable switches serve as first-class citizens to deploy the program. In other words, if primitives in a FastFE program can be expressed within the capability of programmable switches, these primitives are offloaded to the switches. Sometimes if the resources of switches are insufficient for all the deployment, we choose to conduct traffic aggregation in switches prior to the other computations. This is because hash table computation is extremely computation-intensive in software [30]. Since software on servers has enough flexibility to execute the computational tasks of FastFE program, here we mainly focus on how to orchestrate

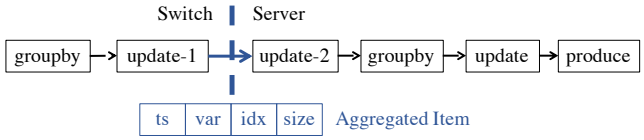

Figure 5: FastFE policy enforcement example.

the FastFE program into underlying programmable switches, how to achieve multi-switch scalability, and how to guarantee order-preserving within switches.

Orchestration. Similar to previous works [10, 21], we translate groupby into a match-action table, by taking argument fields as matching fields of the table, and computing a group index for every packet with the corresponding hash action. filter can be translated in a similar way, in which the predicate pred is converted to matching fields and values, which are installed immediately when translation is finished.

Interface update and produce are more complicated since they involve functions to express logic, which may contain operations and statements unsupported by programmable switches, e.g., division operation and loop statement. Therefore, we need to take their computational complexities into account when deploying these two interfaces on switches. When the function contains computation unsupported by switches, we put the previous operations on the switch and put the remaining operations on the server. If the operation at the beginning is unsupported, we only collect raw data on the switch, and leave the whole computation to the server.

Taking the program in Figure 3 as an example, to update variables, the program needs to conduct multiplication between raw data and float number, which cannot be precisely expressed by current programmable switches. Therefore, the programmable switch only records packet sizes to ensure the detection accuracy, and the recorded data will be sent to servers periodically. Besides, the switch also aggregates packet sizes by source IP addresses to save servers from hash table computation. In contrast, all the operations of the program in Figure 4 can be fitted to programmable switches, so the whole program can be placed at the switch.

When an interface gets deployed on the server, the following interfaces which depend on this interface should also be put on the server. We use the example in Figure 5 to illustrate this scenario, the first update interface is partly deployed on the server and the following FastFE process requires state variables updated by this interface, therefore, the following interfaces are put on the server. In order to accelerate the execution of these interfaces, some preprocessing can be conducted on the programmable switch, and the results are transmitted to the server along with aggregation data. As shown in Figure 5, besides packet arriving time (ts) and some variables (ts) acquired by the server part of the first update interface, we execute hash table computation of second groupby on the switch and append the result to the aggregation item (idx), as well as packet sizes needed by second update interface (size). In this way, we explore the potential of programmable switches to improve the system performance as much as possible.

The primitives on servers can be easily implemented in DPDK-based C program, which takes messages sent by the switch as input, and conducts operations including possible aggregation, calculation and feature production. The complete feature vector will be sent to the back-end behavior detection component.

Multi-switch Scalability. Codes generated by the policy enforcement engine are executed serially. If operators want to support a higher bandwidth, they can employ multiple switches or servers in a parallel way and distribute the incoming traffic evenly to them. Similarly, when a single switch cannot provide sufficient storage/-computation resources and operators want to have more switch stages to conduct the feature extraction, they can employ multiple switches in a pipeline to achieve such a goal.

Order Preserving. Although FastFE can significantly improve the system performance, it also causes potential packet disorder, which further brings challenges to the soundness of target systems. During packet aggregation, packets will be divided into different groups. As a result, later-arrived packets on switches could be sent to the servers earlier, resulting in inconsistent feature vectors. One way to address this issue is to set a timestamp for each packet group on switches. Then the servers restore the packet order according to these timestamps. To avoid some packets staying on switches too long, the servers can track the last arrival time of each memory slot on switch. If a slot has not sent a packet for a long period, the servers can invoke the switch control plane to send a forged packet to evict the content at this slot. In this way, we can preserve packet order in a given time window.

4 CASE STUDY

We use Kitsune as a concrete case study to evaluate how efficient FastFE is for feature extracting. We employ the same modified Kitsune mentioned in §2.2 as the native Kitsune. In order to accelerate Kitsune, the intention of Kitsune's feature extraction can be expressed through the policy interface of FastFE, as shown in §3.1. The policy is inputed into FastFE and then translated into the P4 program running on Tofino ASIC (FE-ASIC) and the Python program running on the server (FE-Server). FE-Server is responsible for receiving the processed and aggregated traffic data from FE-ASIC, and computing the final feature vectors, which are transferred to the behavior detector of the application.

Implementation. Since the full implementation of FastFE is still under development, we conduct the initial validation using a preliminary prototype. Without the policy enforcement engine fully implemented, we manually generate FE-ASIC and FE-Server. FE-ASIC is implemented with ~3K lines of P4 [3] code for the Barefoot Tofino ASIC [24], while the corresponding controller program is written in Python using ~1K lines of code. FE-Server is implemented with ~200 lines of Python code to be compatible with the native Kitsune.

Experimental setup. Our testbed consists of one 3.3Tb/s Barefoot Tofino switch and two Dell R730 servers. Both servers are equipped with Intel(R) Xeon(R) E5-2698 v4 CPUs, 15360K L3 cache, 64GB RAM, and the 40Gbps Intel XL710 NICs to be connected to the switch. Particularly, one server acts as the traffic generator, running `pktgen` or `tcpreplay` to send traffic to the switch, and the other one is used as the back-end server of FastFE to receive aggregated packet data from the switch. In our case study of Kitsune, we choose the SYN-DoS trace provided in its paper as the target of traffic

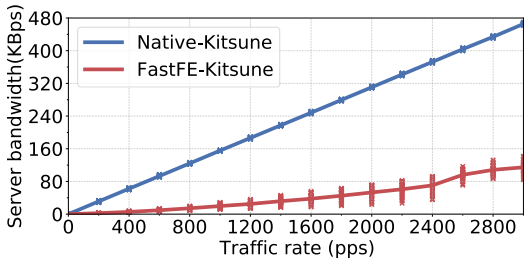

Figure 6: Communication reduction by FastFE.

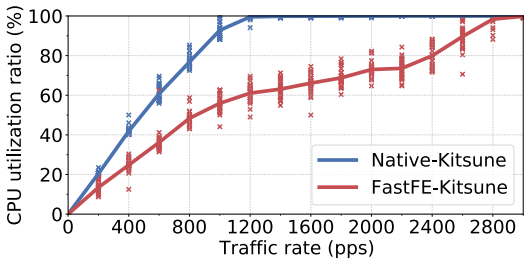

Figure 7: CPU alleviation by FastFE.

Table 2: Resource utilization of Kitsune-FastFE.

Computing			
Tables	**sALUs**	**VLIWs**	**Stages**
27.08%	40.63%	13.54%	50.0%
Memory			
SRAM		**TCAM**	
18.33%		14.58%	

analysis. This trace contains both normal background traffic and abnormal traffic of SYN-DoS.

Soundness. In order not to affect the output of traffic analysis, e.g., the RMSE values that the neural network produces in Kitsune, FastFE needs guarantee to generate the consistent feature vector for each packet with the native one. To validate this property of FastFE, we compare the 115-dimensional feature vectors generated by Kitsune with FastFE and the native Kitsune, and compute their cosine similarity as the metric. Our experiments show that the average cosine similarity for all packets in the trace is as high as 0.989, which indicates the soundness of FastFE. The little difference is due to the rare out-of-order packets in the network.

Scalability. To demonstrate the scalability benefit of FastFE, we measure the bandwidth and CPU utilization of the back-end server as the test stream increases. As shown in Figure 6, with FastFE, the traffic received by the back-end server is only the quarter of the native Kitsune. And Figure 7 indicates employing FastFE reduces the CPU utilization of feature extractor on the server to nearly half of the native Kitsune. These all benefit from the preliminary processing and aggregation for original packets on the programmable switch in FastFE. Both experiments prove the potential of FastFE to scale traffic analysis to large-volume network traffic.

Overhead. Table 2 displays the resource usage of Kitsune with FastFE in our Tofino switch. As we can see, FastFE occupies less than half of computational resources, and uses about 20% of the SRAM and 15% of the TCAM. This result manifests FastFE still leaves enough space for traditional network processing, and it can even be further optimized with more tuning.

5 RELATED WORK

Besides the most relevant traffic analysis applications in §2.1, our work is also inspired by the following topics.

High-level interface. There are many different domain-specific languages and interfaces for different tasks, including packet processing in SDN [1, 7, 19], network monitor [10, 21], network intrusion detection systems [2], DDoS defense [35] and etc. The goal of FastFE is to enable operators to use feature extraction as a service, as a result, FastFE interfaces are specially customized for the feature extracting procedures in traffic analysis applications.

Programmable switch. FastFE builds on the recent trend of leveraging programmable switches to accelerate various application in networking [10, 12, 17, 21, 26, 29, 30, 33], security [5, 16, 35] and distributed systems [14], but focuses on a very different problem: the procedure of traffic feature extraction. We design customized high-level interfaces to express feature extraction procedures for traffic features applications and resolves unique challenges in translating these interfaces into underlying primitives.

6 CONCLUSION AND FUTURE WORK

In this paper, we identify the bottleneck component of today's traffic analysis applications in dealing with high-speed large-volume network traffic, and sketch the vision of FastFE, a high-speed feature extractor that is designed to alleviate the scalability issue in these applications. The core of FastFE is a set of general and expressive interfaces that allow operators to express their desired traffic features flexibly, and a policy enforcement engine that can enforce these policies in programmable switches and servers efficiently. Our initial prototype and case study demonstrate that FastFE can accelerate the procedure of feature extracting significantly.

Nevertheless, our current design, prototype and evaluations are still very preliminary, which leaves a lot of future works to continue. In our ongoing explorations, we are planning to customize our interfaces to make it fitter for traffic analysis tasks, build a full prototype of the policy enforcement engine, apply our approach to more traffic analysis applications, conduct larger-scale real-world experiments, and consider more complex scenarios.

ACKNOWLEDGEMENT

We sincerely thank anonymous reviewers for their valuable comments, and also thank Feng Wei from Clemson university for joining some discussions of this paper. This work is supported in part by the National Key R&D Program of China (2017YFB0801701) and the National Natural Science Foundation of China (No.61625203 and No. 61832013). Menghao Zhang and Mingwei Xu are the corresponding authors.

REFERENCES

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. *Acm sigplan notices* 49, 1, 113–126.

[2] Kevin Borders, Jonathan Springer, and Matthew Burnside. 2012. Chimera: A declarative language for streaming network traffic analysis. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 365–379.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[4] Serdar Cabuk, Carla E Brodley, and Clay Shields. 2004. IP covert timing channels: design and detection. In *Proceedings of the 11th ACM conference on Computer and communications security*. 178–187.

[5] Jiarong Xing Qiao Kang Ang Chen. 2020. NetWarden: Mitigating Network Covert Channels while Preserving Performance. In *USENIX Security Symposium*.

[6] Jack Choquette, Olivier Giroux, and Denis Foley. 2018. Volta: performance and programmability. *IEEE Micro* 38, 2 (2018), 42–52.

[7] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A network programming language. *ACM Sigplan Notices* 46, 9 (2011), 279–291.

[8] Carrie Gates, Joshua J McNutt, Joseph B Kadane, and Marc I Kellner. 2006. Scan detection on very large networks using logistic regression modeling. In *11th IEEE Symposium on Computers and Communications (ISCC'06)*. IEEE, 402–408.

[9] Steven Gianvecchio and Haining Wang. 2007. Detecting covert timing channels: an entropy-based approach. In *Proceedings of the 14th ACM conference on Computer and communications security*. 307–316.

[10] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 357–371.

[11] Jamie Hayes and George Danezis. 2016. k-fingerprinting: A robust scalable website fingerprinting technique. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 1187–1203.

[12] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 161–176.

[13] Steve TK Jan, Qingying Hao, Tianrui Hu, Jiameng Pu, Sonal Oswal, Gang Wang, and Bimal Viswanath. 2020. Throwing Darts in the Dark? Detecting Bots with Limited Data using Neural Data Augmentation. (2020).

[14] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 121–136.

[15] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1–12.

[16] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. 2020. Programmable In-Network Security for Context-aware BYOD Policies. In *USENIX Security Symposium*.

[17] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 15–28.

[18] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. 2018. Kitsune: an ensemble of autoencoders for online network intrusion detection. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*.

[19] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing software defined networks. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 1–13.

[20] Pratik Narang, Subhajit Ray, Chittaranjan Hota, and Venkat Venkatakrishnan. 2014. Peershark: detecting peer-to-peer botnets by tracking conversations. In *2014 IEEE Security and Privacy Workshops*. IEEE, 108–115.

[21] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 85–98.

[22] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. 2018. Deepcorr: Strong flow correlation attacks on tor using deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1962–1976.

[23] Milad Nasr, Amir Houmansadr, and Arya Mazumdar. 2017. Compressive traffic analysis: A new paradigm for scalable traffic analysis. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2053–2069.

[24] Barefoot Networks. 2017. Tofino: World's fastest P4-programmable Ethernet switch ASICs. https://barefootnetworks.com/products/brief-tofino/. (2017). [Online; accessed Jun. 13, 2019].

[25] MIT News. 2020. Monitoring network traffic more efficiently. http://news.mit.edu/2017/monitoring-network-traffic-more-efficiently-0823. (2020). [Online; accessed Apr. 22, 2020].

[26] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Vol. 18. 125–139.

[27] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. 2016. Website Fingerprinting at Internet Scale.. In *NDSS*.

[28] Daniel Ramsbrock, Xinyuan Wang, and Xuxian Jiang. 2008. A first step towards live botmaster traceback. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 59–77.

[29] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Turboflow: information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 11.

[30] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With* Flow. In *2018 USENIX Annual Technical Conference USENIX ATC 18)*. USENIX Association.

[31] Stuart Staniford-Chen and L Todd Heberlein. 1995. Holding intruders accountable on the internet. In *Proceedings 1995 IEEE Symposium on Security and Privacy*. IEEE, 39–49.

[32] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. 2014. Effective attacks and provable defenses for website fingerprinting. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 143–157.

[33] Nofel Yaseen, John Sonchack, and Vincent Liu. 2018. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 402–416.

[34] Kunikazu Yoda and Hiroaki Etoh. 2000. Finding a connection chain for tracing intruders. In *European Symposium on Research in Computer Security*. Springer, 191–205.

[35] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qi Li, Mingwei Xu, and Jianping Wu. 2020. Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches. In *NDSS*.

[36] Yin Zhang and Vern Paxson. 2000. Detecting stepping stones.. In *USENIX Security Symposium*, Vol. 171. 184.