The Nuts and Bolts of Building FlowLens

Diogo Barradas, Nuno Santos, Luís Rodrigues,
Salvatore Signorello, Fernando M. V. Ramos, André Madeira

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

{diogo.barradas, nuno.m.santos, ler, salvatore.signorello, fvramos, andre.madeira}@tecnico.ulisboa.pt

***Abstract*—In this paper, we augment our previous NDSS publication and discuss our experience evaluating FlowLens [7], a system that leverages programmable switches to support machine learning-based network security applications. Specifically, we discuss our experience when seeking to understand whether FlowLens was compatible with applications that classify network flows using frequency distributions of packet lengths and timing. To perform this assessment, we started by determining the eligibility of multiple ML-based network flow analyzers using five criteria and selected three analyzers. We then adapted each analyzer's classifier to FlowLens and evaluated the accuracy of flow markers (compressed representations of raw packet length and timing distributions) in comparison with the use of the raw distributions used to evaluate each selected analyzer in their original papers. To this end, we compared the use of FlowLens-produced flow markers with the use of raw packet distributions in the profiling process used to train and test the analyzer's model. Our evaluation revealed that FlowLens achieves a comparable accuracy with the flow analyzers that use raw packet distributions. Its core benefit is that its use of flow markers allows for the monitoring of up to 150x simultaneous flows crossing a switch when compared to the use of raw packet distributions. Our exposition provides a number of observations, lessons learned, and recommendations based on our experience evaluating FlowLens. These focus on challenges considering mismatches between the programming model of software and hardware P4 targets, the harmonization efforts to ensure the compatibility of heterogeneous traffic analysis tasks with FlowLens, and the difficulty in reproducing networking-based experiments. The lack of open experimental artifacts of competing solutions also limited our comparative analysis.**

I. Introduction

Many security-driven network monitoring scenarios require the ability to identify specific flows in real-time. For this purpose, traditional deep-packet inspection has become increasingly ineffective as a result of a global trend towards encrypting all Internet traffic [13], being complemented by sophisticated flow identification techniques based on machine learning (ML) [30]. ML-based techniques can be employed in the analysis of encrypted traffic based on various packet features [2] and classify flows with high accuracy for a range of application scenarios, such as covert channel detection [8], website fingerprinting [19], botnet traffic identification [27], malware tracking [2], IoT device behavioral analysis [31], or detection of DRM-protected streaming [35].

Learning from Authoritative Security Experiment Results (LASER) 2021
25 February 2021
ISBN 1-891562-71-1
https://dx.doi.org/10.14722/laser.2021.23067
www.ndss-symposium.org

The recent adoption of programmable switching devices has enabled the deployment of efficient packet processing primitives in large-scale, high-speed networks. These capabilities have sparked a consistent effort from the research community to perform network security tasks in such switches with the goal of decreasing reaction times to threats and reducing costs associated with equivalent centralized server-based infrastructures. Unfortunately, existing solutions [20, 23, 41] target specific security-driven tasks and cannot accommodate ML tasks that perform targeted flow classification based on packet size or inter-packet frequency distributions.

To tackle these limitations, we introduced FlowLens [7], a system that leverages programmable switches to efficiently support multi-purpose security network applications based on machine learning algorithms. Our system enables network operators to program their switches in order to automatically scan and classify flows with high accuracy for a wide range of scenarios, such as multimedia covert channel detection, website fingerprinting, or botnet traffic identification. To this end, FlowLens introduces a new system design (detailed in Section II) that solves a fundamental tension between the need for meaningful flow information required by machine learning algorithms and the scarcity of hardware resources available in modern programmable switches. Briefly, FlowLens computes a memory-efficient representation of each flow's relevant features, named *flow marker*. Flow markers are specifically tailored for a given ML task, ensuring a reasonable balance between the size of the flow marker and the accuracy of the flow classification task. FlowLens also implements a data structure, named flow marker accumulator, which is responsible for collecting flow markers as packets cross a programmable switch. Periodically, flow markers are used to classify flows locally on the switching device without requiring any additional infrastructure.

In this paper, we discuss how we have addressed three major challenges faced during the implementation and evaluation phases of FlowLens. These challenges have raised the bar in proving the feasibility and effectiveness of our design. First, we identified a substantial gap between the initial programming environment (based on the P4 programming language) targeting a software-emulated switch, and a production-level hardware switch (i.e., the Barefoot Tofino). This gap forced us to deeply refactor our code and revisit the assumptions underpinning our original flow compression technique (Section III). Second, we realized that different machine learning security tasks proposed in the literature had been fine-tuned for their specific application domains. This means that not only do they employ different classification algorithms but even the datasets used and the training processes are different from one another. As

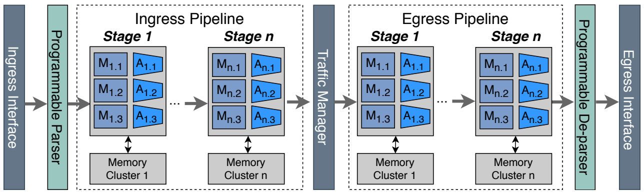


Figure 1. Protocol Independent Switch Architecture (PISA).

such, we had to repurpose the classification machinery of previously existing applications to ensure their compatibility with FlowLens (Section IV). Lastly, the comparison between our compression technique and other related techniques was hampered by the lack of access to the implementations of the latter's. This forced us to re-implement several such approaches and in some cases to resort to analytical evaluations of their compute, storage, and communication costs (Section V).

During our exposition, we seek to provide the reader with a summary of the lessons we have learned when overcoming the above challenges. We expect that other researchers and practitioners may take heed on such lessons to learn from our own experimental methodologies, execution, and results. Likewise, we make available a set of recommendations aimed at fostering the reporting of know-how, in the form of experimental artifacts, towards reproducible security research (Section VI). We conclude our paper by presenting directions for future work (Section VII) and our conclusions (Section VIII).

Availability: Our experimentation artifacts include a P4 prototype of FlowLens, and the necessary code for adapting three ML-based security applications' workflow to FlowLens. We made these artifacts publicly available [9] and encourage the community to build on and extend our results.

II. Background

A. Programmable Switches

Software-defined networking aims to improve the scalability and control of computer networks by simplifying the operation of network switches through the disaggregation of the forwarding process of network packets (data plane) from the routing process (control plane). Existing programmable switches implement this concept, featuring two different processing units that act on each of the different planes of the network, respectively. On the data plane, programmable switches feature specialized forwarding ASICs to perform simple computations on and/or forward packets at line-rate. On the control plane, programmable switches feature a general-purpose CPU that is responsible for performing routing decisions, handling traffic policies, and performing other general-computing tasks. Examples of such switches include the Barefoot Tofino [6] and Broadcom Tomahawk [11].

P4-programmable switches: Programmable switches allow users to define, by means of a program, the exact way packets should be processed in a switch data plane. The programs can be written in a hardware-independent programming language, such as P4 [10], which allows for expressing the packet parsing and packet processing operations. Figure 1 depicts the internals of a modern programmable switch, the Protocol Independent Switch Architecture (PISA) [12]. When a packet arrives at the switch's ingress interface, it is processed by a pipeline of *match+action units* (MAUs) organized into *stages*. Packet headers along with packet metadata may then *match* (M) a given table in a stage, triggering further processing by an *action* (A) associated with the matching table's entry. These actions may modify packet header fields, packet header metadata, or some persistent state in the switch (e.g. writing to a register in stateful memory). The tables are initialized by the P4 program and are later populated by the control plane.

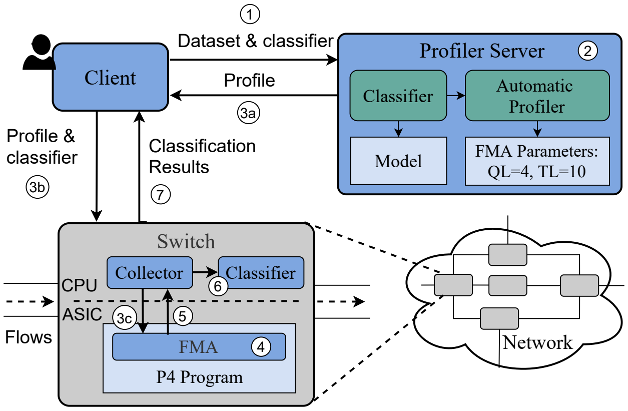


Figure 2. FlowLens architecture (adapted from our original paper [7]).

Constraints: To achieve line-rate at Tbps, PISA-like switches impose important constraints on the operations that MAU actions can do, requiring careful engineering when programming such devices effectively [16]. First, switching ASICs feature a small amount of stateful memory (≈100MB SRAM [24]), and only a fraction of the available SRAM can be used to allocate register arrays. Second, the registers in one stage cannot be accessed by different stages since the SRAM is uniformly distributed amongst the different stages of the pipeline (see Figure 1). Third, packets must spend a fixed amount of time in each pipeline stage (a few ns [37]) to guarantee line-rate processing. This restricts the number and type of operations allowed within each stage. Multiplications, divisions or floating-point operations, and variable-length loops, are not supported. Moreover, each table's action can only perform a restricted set of simpler operations, like additions, bit shifts, and memory accesses so as not to stall the whole pipeline.

B. FlowLens

We proposed FlowLens [7], a system that leverages programmable switches to collect packet distributions at line speed to support the efficient multi-purpose classification of flows on the switches, e.g., for botnet detection. Figure 2 depicts the architecture and the different modules of FlowLens.

Components: FlowLens' architecture is composed of five main components. First, FlowLens features a *profiler server* which generates application-specific profiles that identify compression parameters and the best available ML model to use in the classification task. Second, FlowLens features a P4 program running on the switch data plane that implements a custom data structure named *Flow Marker Accumulator* (FMA). The FMA is responsible for collecting a compressed representation of the packet length or inter-packet timing distributions of flows, named *flow markers*, during a given sampling period. Third, FlowLens features a *collector* on the switch control plane that loads the P4 program in the forwarding pipeline, initiates the flow collection process, and collects the resulting flow markers. Fourth, also in the control plane, a *classifier* executes the supervised ML algorithms that classify flows based on the collected markers. Lastly, FlowLens includes a software *client* interface that allows system operators to use FlowLens.

Operation workflow: Suppose that a network operator is interested in using FlowLens to perform the task of botnet traffic detection. In such a case, the operation of FlowLens proceeds with the following seven steps, as labeled in Figure 2:

① FlowLens' operator uploads a classifier and a training dataset composed of legitimate and botnet packet traces to the profiler.

② The profiler processes the traces in the dataset and generates flow markers according to different compression parameters. The profiler will automatically select a parameterization (which we call *profile*) that achieves a reasonable trade-off between the memory occupied by flow markers and the accuracy achieved when carrying out the detection of botnet traffic.

③ The selected profile and classifier are loaded into the P4 program that will run in the data plane of the switch.

④ The collector instructs the P4 program to begin the flow collection process. As flows cross the switch, their respective flow markers will be computed and stored inside the FMA.

⑤ After a sampling period defined by the operator, the collector retrieves flow markers from the FMA data structure.

⑥ The collected flow markers are fed to the classifier co-located in the control plane, which classifies individual flows as legitimate or botnet-generated traffic.

⑦ The results of the classification are finally propagated back to the operator, who can then issue targeted actions like dropping or trigger further logging operations over flagged flows.

Flow marker generation: To generate flow markers, e.g. based on flows' packet length distributions, the FMA features the efficient implementation of two operators: (i) *quantization*, which consists in counting packet lengths in coarse bins that represent ranges of contiguous packet lengths, and (ii) *truncation*, which further trims the number of bins that need to be reserved for a given application. The rationale for the use of such operators is twofold. First, the use of quantization allows us to obtain a coarser (yet smaller) representation of the overall shape of a flow's packet distribution, yielding important space savings on the size of a flow marker. Further, quantization may also allow for decreasing nefarious effects caused by the "curse of dimensionality" when training ML models, which may actually improve the accuracy of classification tasks to some extent. Second, the literature shows that, for obtaining accurate predictions on many traffic analysis tasks, it is only necessary to collect a limited number of bin values that correspond to the most relevant features employed by the ML model [8, 42]. In the next sections, we discuss the three main challenges we faced when implementing and evaluating FlowLens.

III. Mismatches between implementation targets

Due to the scarce memory and intrinsic programming restrictions of current programmable switches, our FMA implementation is driven by the goal of achieving high efficiency. In other words, we aim at utilizing the memory available in each stage of the switch's pipeline to its fullest to store flow markers, and to implement the quantization and truncation operators resorting to simple sets of instructions.

During the development of FlowLens, we adopted the common practice of implementing a prototype in the reference P4 software switch before porting it to a hardware target. Yet, the process of porting our initial implementation to hardware has raised significant challenges in the implementation of FlowLens' quantization and truncation operators. After delivering a brief overview of our implementation targets, we will detail these challenges and lessons learned.

A. P4 targets used in FlowLens development

The newest version of the P4 language ($P4_{16}$) supports *architectures* to enable P4 on a multitude of hardware devices, such as switches, routers, or NICs, but also on software devices like Open vSwitch. Architectures provide a logical view of the packet parsing and processing pipeline, allowing programmers to abstract away from gritty hardware details that can slowdown prototyping. Despite being an important step towards the portability of P4 code, the intrinsic characteristics of different targets may force programmers to adapt their P4 implementations to adhere to target-specific constraints. Specifically, we dealt with two different programmable switch targets that implement the PISA architecture:

Reference P4 software switch: In 2016, the P4 language consortium has released `bmv2` [32], the second version of the reference P4 software switch. `bmv2` is a tool intended for the development, testing, and debugging of P4 data/control plane software, and is not affected by the constraints listed in Section II-A. However, due to its development-oriented design and to the fact that it processes packets in software, `bmv2`'s performance is orders of magnitude slower when compared to production-grade software switches. On the other hand, `bmv2` is open-source and provides a flexible target architecture, making it ideal for prototyping. In addition, the P4 language consortium also makes available to practitioners a VirtualBox image [3] that includes all the switch software and P4 code examples.

Barefoot Tofino ASIC: Barefoot Tofino [6] is the first programmable Tbps Ethernet switching ASIC designed for production environments. To assess whether a P4 program can run at line rate without stalling the switch's pipeline, Tofino ships with a dedicated P4 compiler as part of the Intel P4 Studio SDE. The program is ensured to run at line rate as long as it compiles successfully. As noted in Section II-A, in contrast to `bmv2`, the Tofino enforces several constraints that must be taken into account to program the device effectively.

Since the initial implementation of FlowLens was developed for the reference P4 software switch, it did not fully consider the intrinsic restrictions of the Tofino ASIC. While this first prototype was seemingly effective, its adaptation to the Tofino required a significant redesign in order to implement FlowLens' quantization and truncation operators. Next, we detail the design process leading to the implementation of these operators and highlight the major obstacles overcome during this process.

B. Target mismatches when implementing quantization

In this section, we start by describing our initial implementation of the quantization operator in the P4-reference switch.

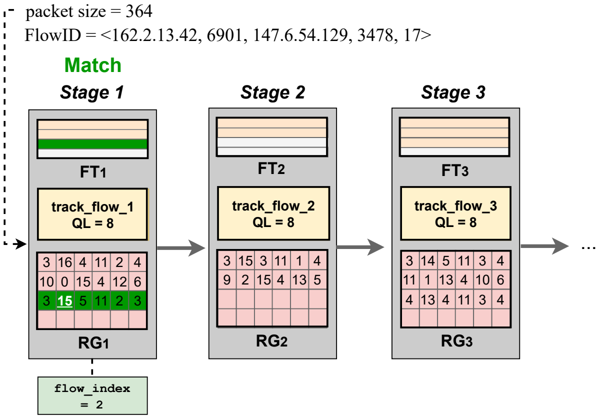


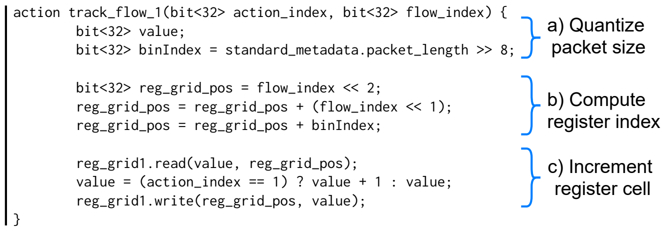
Figure 3. Implementation of FMA's quantization operator on the reference P4 software switch. A quantization level (QL) equal to 8 will discretize packet lengths in 2^8 bytes-sized bins up to a total of 6 bins, assuming the maximum packet size to be equal to an Ethernet's MTU (1500 bytes). The numbers in the register grids (RGx) represent counters of packets pertaining to each bin.

Then, we detail a number of changes and optimizations required to make our implementation compatible with the Tofino ASIC.

Initial FMA implementation for bmv2: Figure 3 depicts a schematic of our initial working implementation of FMA's quantization operator on the P4-reference switch. Each pipeline stage n is composed of three major units: i) a *flow table* (FT_n) that indexes new flows arriving at the switch, ii) an *action* (`track_flow_n`) that is triggered when a packet is matched against one of the flow table rules, and iii) a *register grid*, held in stateful memory, where each row stores the flow marker for a given flow indexed in the flow table.

The example shown in Figure 3 works as follows. First, the FMA's flow table matches the incoming packet with the corresponding flow ID, which is a 5-tuple of header fields $\langle IP_{src}, Port_{src}, IP_{dst}, Port_{dst}, Proto\rangle$ that is used as lookup key to return its associated flow index. For instance, in the running example, the input packet is matched against the rule $\langle 162.2.13.42, 6901, 147.6.54.129, 3478, 17\rangle \rightarrow 2$. Upon matching this rule, the corresponding action `track_flow_1` is invoked. The P4 implementation of this action (reported in Figure 4) features three main sections responsible for a) quantizing the packet size by applying a pre-determined amount of bit-shifts over the packet size, b) computing the register grid index where the packet should be accounted for, and c) incrementing the register cell at the computed index. Put simply, the action works as follows in this particular example. First, we apply a quantization level QL=8 to an incoming packet size of 364B. This means that we perform the integer division of 364 by 2^8, which yields a bin index value of 1. Second, we compute which row of the register grid corresponds to the flow marker of the current flow by multiplying its flow index by the size of a register grid row. Finally, we find and increment the position of the bin index that was previously computed resorting to quantization, within the flow marker (bin index 1, highlighted in white).

Already aware of some of the major hardware switches' programming constraints, our implementation of the `track_flow_1` action for `bmv2` already included a number of optimizations. First, quantization generates bins in ranges aggregated as powers of two (2^{QL}), allowing for an efficient implementation that resorts to bit-shifts in order to avoid divisions. Second, we leverage bit-shift and addition operations to express the necessary multiplication for computing the register grid index. Lastly, we make a single read-modify-update sequence to take into account the occurrence of a new packet in the flow marker.



```
action track_flow_1(bit<32> action_index, bit<32> flow_index) {
        bit<32> value;
        bit<32> binIndex = standard_metadata.packet_length >> 8;

        bit<32> reg_grid_pos = flow_index << 2;
        reg_grid_pos = reg_grid_pos + (flow_index << 1);
        reg_grid_pos = reg_grid_pos + binIndex;

        reg_grid1.read(value, reg_grid_pos);
        value = (action_index == 1) ? value + 1 : value;
        reg_grid1.write(reg_grid_pos, value);
}
```

a) Quantize packet size
b) Compute register index
c) Increment register cell

Figure 4. P4 code of the `track_flow_1` action for bmv2.

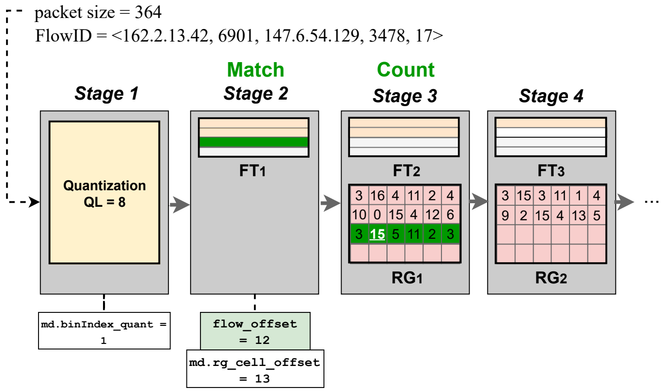
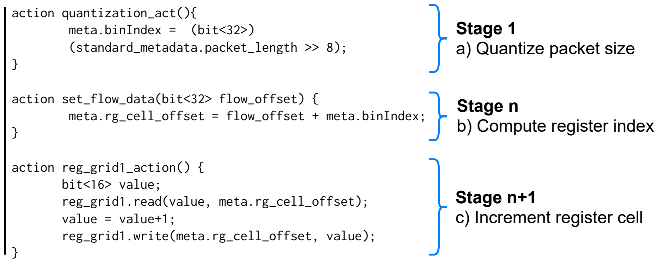


Figure 5. Implementation of FMA's quantization operator on hardware.

Porting the bmv2 code to hardware: Despite the optimizations introduced in our initial implementation for the reference P4 software switch, we were unable to successfully compile this code on the Tofino ASIC. In fact, the operations specified in the `track_flow_n` action cannot all be executed within a single stage of the switch pipeline.

To get around this constraint, we implemented an alternative FMA design that splits the quantization operator across multiple stages of the pipeline, as shown in Figure 5. In this updated design, the quantization operator is now split into three different stages, where each stage is responsible for a single section of the old `track_flow_n` action (see the corresponding P4 snippet in Figure 6). In Figure 5's running example, the first stage quantizes the packet size through the `quantization_act` action (Figure 6 a)). For this particular example, where the incoming packet is matched in stage 2, the second stage computes the register index (Figure 6 b)). Correspondingly, the third stage increments the matching cell of the flow marker. Note however that, as it can also be observed in Figure 5, reworking our FMA implementation is made at the expense of SRAM memory. Due to the feed-forward nature of the switch pipeline, the SRAM memory clusters in stage 1 and stage 2 cannot be used to store flow markers with this FMA design.

Observation 1: Per-stage computation must be kept simple in current programmable switches. The logic for multi-step operations may need to be divided across different stages, possibly leading to an additional waste of resources, e.g., SRAM local to a certain stage.



```
action quantization_act(){
        meta.binIndex =  (bit<32>)
        (standard_metadata.packet_length >> 8);
}

action set_flow_data(bit<32> flow_offset) {
        meta.rg_cell_offset = flow_offset + meta.binIndex;
}

action reg_grid1_action() {
       bit<16> value;
       reg_grid1.read(value, meta.rg_cell_offset);
       value = value+1;
       reg_grid1.write(meta.rg_cell_offset, value);
}
```

Stage 1
a) Quantize packet size
Stage n
b) Compute register index
Stage n+1
c) Increment register cell

Figure 6. P4 code of the quantization operator for Tofino.

C. Target mismatches when implementing truncation

As previously described in Section II-B, FlowLens' truncation operator aims at filtering out quantized bins that bring no significant advantage for a given network traffic classification task. In practice, this means that only a pre-selected subset of bins is actually accounted for in the flow marker, allowing us to reduce the overall flow marker memory footprint. A naïve approach to implement this behavior would be to build an `if` chain to test whether the bin outputted by the quantization stage should be accounted for in the flow marker. However, for small values of the quantization level (e.g., QL=[0,1,2,3]), the number of such comparisons would be in the order of hundred operations, requiring multiple stages of the pipeline to be dedicated to this computation alone.

A key observation, however, is that the choice of bins to truncate can be computed offline during FlowLens' profiling stage. This makes it possible for the control plane to initialize a match table that triggers a specific action when a given quantized bin matches some table rule. Ultimately, this allows us to avoid the waste of computation and memory resources in the switch pipeline. Figure 7 depicts the implementation of this truncation design. As in the previous example, stage 1 is responsible for quantizing the incoming packet length. In this setting, stage 2 is used for implementing the truncation operator. It works by matching the quantized bin index (`qt_idx`) computed in stage 1 against a match+action table. As there is a rule for `qt_idx` = 1 in the table, the action triggered in stage 2 sets the truncated `bin_offset` to 1 and sets the truncation flag `trunc_flag` to signal further stages that this particular packet should be accounted in the flow marker. The remaining pipeline is similar to our initial implementation. In stage 3, upon matching the packet in the flow table and checking that the truncation flag is set, the register cell offset is computed, and further incremented on stage 4.

> **Observation 2:** Complex computations should be offloaded to the control plane whenever possible in order to save the limited resources on the switch data plane.

IV. Harmonization of ML-based security tasks

FlowLens was conceived as a traffic analysis system able to support any generic ML-based network security application that focuses on the analysis of packet lengths and inter-arrival timing distributions. To show that FlowLens can provide support to multiple heterogeneous applications, we were required to find a set of representative network security applications whose classification workflow can be adapted to use flow markers instead of raw packet distributions. However, the selected third-party applications had to adhere to a number of non-trivial requirements so as to be compatible with our testbed.

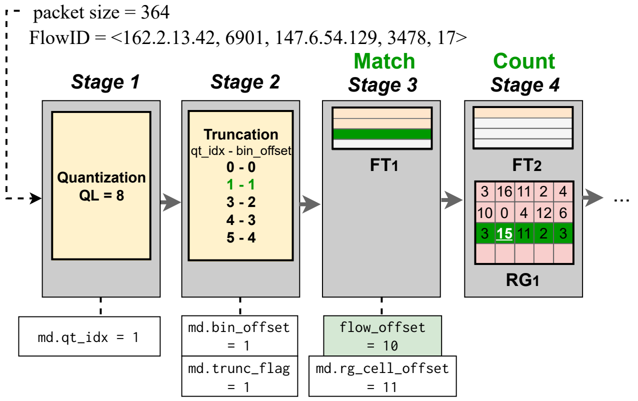


Figure 7. Implementation of FMA's truncation operator for the Tofino target.

In the next sections, we aim to answer what are the necessary conditions for an application to be compatible with FlowLens, to detail the process we have followed to select potential candidate applications, and to describe how we adapted the classification workflows of the selected applications to use flow markers.

A. Eligibility criteria for experimentation with FlowLens

In our earlier experiments with FlowLens, we were interested in adapting ML-based network security applications whose methodology was sound, whose code and datasets were available for experimentation, and whose results we would be able to validate prior to adaptation. In essence, to be eligible for experimentation in the context of FlowLens, candidate third-party applications had to adhere to the below criteria.

Peer-reviewed publications: The first criterion demands that a candidate application is supported by a peer-reviewed publication. This requirement ensures that the application's methodology is sound, and that its integration with FlowLens may raise the interest of the research community.

Datasets of packet traces: The second criterion for the selection of candidate applications is that these should perform some network traffic classification task resorting to features extracted from flows' packet length or inter-arrival time distributions. In the particular context of our system, we require the access to a raw representation of such distributions so as to be able to apply FlowLens' quantization and truncation operators.

Compatible classifiers: The third criterion demands that the selected applications perform flow classification tasks through the use of supervised classification techniques. This is imperative due to FlowLens' profiling step described in Section II-B. Moreover, such classifiers are required to be executed efficiently resorting to the resources available in the control plane of existing programmable switches, which include a general-purpose CPU and several tens of GB RAM.

Experimental artifacts: The fourth criterion for the eligibility of an application is concerned with the availability of experimental artifacts that could speed up our development efforts. Ideally, the necessary code for pre-processing and classifying network flows should be publicly available.

Application category	Publication	Dataset of packet traces	Compatible classifier	Experimental artifacts	Reproducible
Botnet Traffic Detection	Narang et al. [27]	✓	✓	✓	✓
	Meidan et al. [22]	-	✓	✓	✓
Covert Channel Detection	Barradas et al. [8]	✓	✓	✓	✓
	Geddes et al. [17]	-	✓	-	-
Website Fingerprinting	Herrmann et al. [19]	✓	✓	✓	✓
	Nasr et al. [28]	-	✓	-	-
Application Fingerprinting	Taylor et al. [40]	-	✓	✓	✓
	Aceto et al. [1]	-	✓	-	-
Video Fingerprinting	Schuster et al. [36]	-	-	-	-
	Li et al. [21]	-	-	-	-

Table I. ELIGIBILITY ASSESSMENT OVER TEN DIFFERENT ML-BASED NETWORK SECURITY APPLICATIONS.

Reproducible results: The last eligibility criterion dictates that an application's methodology and experimental artifacts must suffice to enable the faithful reproduction of the results reported in the supporting peer-reviewed publication. This would enable us to understand to what extent an application's accuracy would be impacted by the adaptations performed by FlowLens.

Next, we detail our reviewing process over multiple network security applications in light of the aforementioned criteria towards the selection of our candidate applications.

B. Selection of ML-based network security applications

To evaluate FlowLens, we assessed the eligibility of multiple ML-based applications, in the realm of network security, that rely on the analysis of the distributions of packet lengths and inter-arrival timing for performing the classification of network flows. Specifically, we sought to understand whether such applications were compatible with FlowLens and if their repurposing was practical. Table I illustrates a summary of the eligibility assessment we conducted over these applications, regarding the five criteria considered in Section IV-A. Below, we describe each of the considered applications and present the results of our assessment with respect to each criterion.

Pool of ML-based network security applications: We assessed the eligibility of ten emerging ML-based network security applications scattered over five different categories. Each application is supported by a peer-reviewed publication. Below, we describe each category:

- *Botnet traffic detection:* The applications in this category propose detection frameworks for identifying the presence of botnet communication flows amongst legitimate P2P traffic [27] and IoT device traffic [22].
- *Detection of covert channels:* These applications envision the deployment of traffic analysis frameworks aimed at the detection of covert channels established over multimedia streams such as Skype [8, 17].
- *Website fingerprinting:* These applications leverage statistical traffic analysis techniques to identify webpages browsed over encrypted tunnels such as OpenSSH [19] or over anonymity networks like Tor [28].
- *Application fingerprinting:* The applications in this category [40, 1] aim to recognize mobile apps through the traffic patterns that these generate.
- *Video fingerprinting:* These applications resort to traffic analysis to identify encrypted video streams [21, 36].

Datasets of packet traces: FlowLens demands the use of raw packet traces for applying the quantization and truncation operators during the generation of flow markers. As it can be observed in Table I, the availability of data comes across as the most effective elimination criterion to single-out the eligibility of applications, since only three out of the ten applications we considered make raw packet traces available.

While Meidan et al. [22] and Taylor et al. [40] have publicly released their datasets, these only expose pre-processed features such as statistical indicators derived from raw packet traces. This prevents us from re-generating feature sets through the application of FlowLens quantization and truncation operators, so making us unable to test such applications due to the lack of access to raw packet traces. Unfortunately, Geddes et al [17], Nasr et al. [28], Aceto et al. [1], Schuster et al. [36], and Li et al. [21] do not make their datasets available.

Compatible classifiers: To generate and classify flow markers, FlowLens requires the use of efficient supervised ML algorithms. From our analysis, all the classifiers used in the considered applications work in a supervised fashion. Further, the results in Table I show that, with exception to the work of Schuster et al. [36], and Li et al. [21], the classifiers for all the remaining applications can be run efficiently in the control plane of the programmable switch.

Classifiers that we deem compatible with FlowLens include, for instance, the tree-based classifiers like random forests and XGBoost, used by Narang et al. [27] and Barradas et al. [8], respectively. Such classifiers can be efficiently parallelized on the commodity CPU that is typically available in programmable switches. Instead, the classifiers incompatible with FlowLens include those that resort to deep learning approaches and that typically require the use of GPUs to accelerate model inference for near real-time traffic inspection. Since programmable switches are not typically equipped with GPUs, we did not consider the corresponding deep learning applications.

Experimental artifacts: In Table I, we can see that experimental artifacts are only available for half of the considered applications. For such applications, the authors provide pointers to code hosted on GitHub, alongside proper documentation, on the peer-reviewed publications that introduce their own traffic analysis frameworks. While some experimental artifacts were less comprehensive than others, this was often not a huge barrier when adapting such applications to our experimental testbed. For instance, in the case of Herrmann et al. [19], the authors

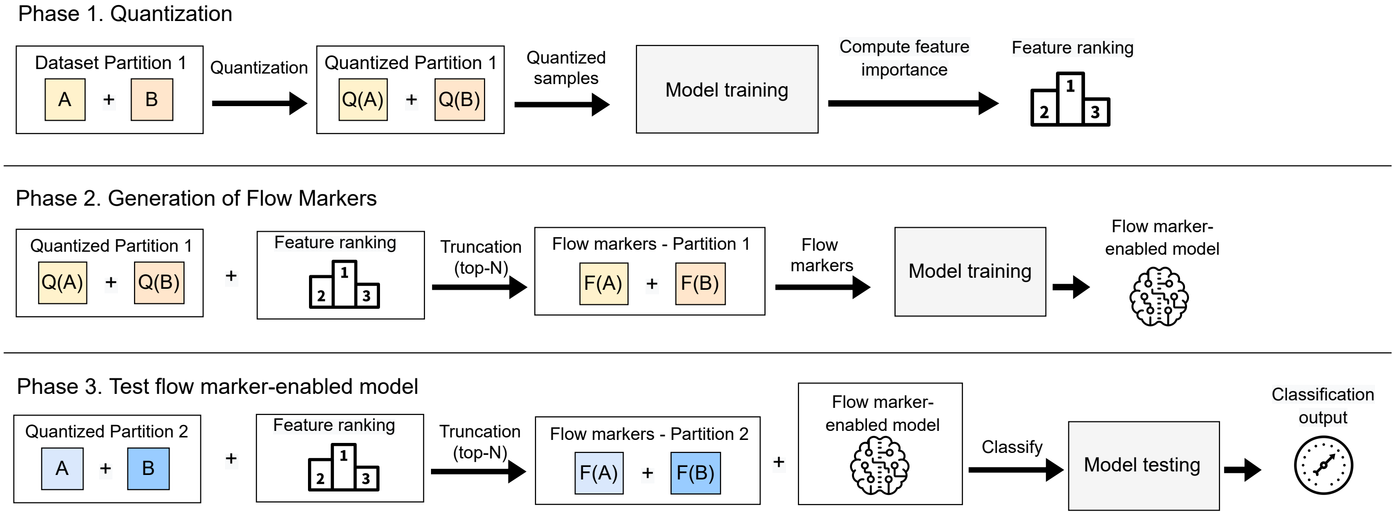


Figure 8. The three phases of the FlowLens profiling procedure.

omitted the code for partitioning their dataset but provided a detailed description of their methodology and released a plugin for the Weka ML library which contained the implementation of their classifier.

Reproducible results: Given the availability of data and the correctness of available experimental artifacts, the last column of Table I depicts the results of our efforts with reproducing the original flow classification results reported for each of the considered applications. Briefly, the table shows that we were able to reproduce the experimental results for all the applications for which experimental artifacts were made available. For all such applications, the reproduction of the original results required only the installation of the `sklearn` Python package, along with other classifier-specific packages such as `xgboost` and the Weka library for machine learning.

Selected applications: Despite the fact that we were able to correctly reproduce the results for five of the ten applications considered in Table I, only three of the applications fulfilled the five criteria outlined in Section IV-A. Thus, for evaluating FlowLens, we focused our attention on three different applications: botnet traffic detection (Narang et al. [27]), covert channel detection (Barradas et al. [8]), and website fingerprinting (Herrmann et al. [19]).

Observation 3: Sharing datasets that feature complete traffic traces can foster experimentation with frameworks that analyze raw flow characteristics.

C. Adaptation of the classification workflow

As mentioned in Section II-B, FlowLens requires a profiling procedure for picking a flow marker configuration that achieves a reasonable balance between classification accuracy and memory footprint. In short, this profiling procedure demands for the generation of flow markers and the subsequent training and testing of supervised ML models using said flow markers. For enabling this procedure, FlowLens requires important adaptations of the classification workflow of the heterogeneous ML-based network security applications considered in the context of our work. Next, we outline such necessary adaptations, detail the profiling procedure used in FlowLens, and explain how we evaluate the accuracy of the produced flow markers.

Adaptation of the classification workflow: FlowLens' profiling procedure requires the partitioning of the original application dataset in two balanced halves, where the first half is used for training models based on different flow marker configurations, and the second half is used for testing said models. In essence, this corresponds to the notion of *holdout evaluation* [34]. However, this exercise is different for most ML-based network security applications found in the literature. In fact, a similarity between all the classifiers used in the applications we have selected is that these were evaluated following a 10-fold *cross-validation* [33] process. Briefly, cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample that enables the assessment of how the classifier results' will generalize to an independent data set. As detailed next, FlowLens replaces cross-validation with holdout evaluation during profiling.

FlowLens profiling procedure: Figure 8 depicts the three phases involved in FlowLens' profiling procedure. In *Phase 1*, we quantize the samples in the first dataset partition using different quantization levels and build several classification models accordingly. After validating these models through the use of a holdout set, we compute feature importance for each model. In *Phase 2*, we generate multiple models trained with different flow marker configurations. Flow markers are generated by selecting a quantization level and by selecting the top-N most important features obtained from the feature ranking performed over the model created with the corresponding quantization level. Lastly, *Phase 3* is responsible for evaluating the models trained with the use of flow markers. To this end, we generate flow markers for the samples in the second partition of the dataset, and use these to test the different models obtained in Phase 2. Upon obtaining the classification results, the FlowLens operator can select the model that achieves a pre-defined trade-off between classification accuracy and flow marker size.

Evaluating the accuracy of flow markers: Since FlowLens modifies the classification workflow of existing applications, this causes the obtained results not to be directly comparable in an apples-to-apples fashion with the original application's classification accuracy. As described above, adapting an application to FlowLens involves a) the slicing of the original dataset, and b) exchanging the application's model evaluation process from cross-validation to holdout validation. However, we note that the ultimate goal of FlowLens is that of generating a flow marker that can achieve an accuracy akin to the use of raw packet traces while occupying a much smaller memory footprint. Thus, to evaluate the quality of flow markers, it is enough to follow the same profiling process tied to model training and testing. In this case, however, we conduct the profiling process using different flow marker configurations instead of the original raw packet data, and assess the achieved classification accuracy. Nevertheless, for all the applications we adapted in our work (`WF`, `BTD`, `CCD`), the accuracy obtained when following FlowLens profiling process while using raw packet data is comparable to the results obtained in the original papers (e.g., `WF` achieves 97% accuracy both in the original classification setting and in the modified FlowLens classification workflow when using raw packet length distributions).

Observation 4: It is possible to adapt heterogeneous ML scenarios into a uniform classification framework like FlowLens, despite the fact that such applications rely on different classification processes.

V. Hard-to-reproduce networking experiments

The main goal of FlowLens is that of scaling the amount of flows that can be inspected simultaneously in a single switch, while enabling ML-based network security applications to retain their accuracy. Needless to say, however, is that FlowLens is not the only system that attempts to scale traffic inspection capabilities while reducing the costs of network telemetry infrastructure. In particular, we were interested in comparing the scalability gains of FlowLens with two different classes of approaches proposed in the literature based on i) disparate flow compression techniques like *online sketching* [14] and *compressive traffic analysis* [28], and ii) packet aggregation solutions such as like *Flow [39]. Next, we describe the two major deterrents we faced when accomplishing this task.

A. Unavailability of experimental artifacts

An important step in our evaluation comprised the comparison of FlowLens against online sketching and compressive traffic analysis, two techniques that rely on linear transformations to generate compressed representations of packet length and inter-arrival timing distributions. Alas, upon a thorough search, we were unable to find a public release of the code of neither technique. This may suggest that prototypes for these techniques were not ready for being released and that such preparation could impose additional hours of work that authors would be unwilling to spend.

In light of these difficulties, we decided to re-implement these flow compression schemes using the definitions and methodologies expressed in the papers. Oftentimes, however, and due to space limitations, the authors omit implementation details and overlook the description of slight coding nuances, providing only a simplified implementation description. As a result, other researchers and practitioners who are forced to re-implement such methodologies may inadvertently fail to respect original implementation decisions. As a result, it is possible for the results emerging from an *aposteriori* comparison to be skewed due to implementation discrepancies.

Nevertheless, in our paper [7], we attempted to follow the steps described in the original manuscripts and produce faithful re-implementations of the online sketching and compressive traffic analysis solutions. We are confident of our re-implementation since these techniques were well documented and relied on rather straightforward mathematical concepts like matrix multiplication. Ultimately, the results of our experiments revealed that FlowLens was able to outperform both techniques and we have publicly released our re-implementations [9].

Observation 5: The reproducibility of experiments with traffic analysis frameworks may be hampered by code unavailability. This means that re-implementation is required and that practitioners may run the risk of failing to respect original implementation decisions.

B. Cumbersome deployment of existing testbeds

The second major step of our comparative evaluation encompassed the juxtaposition of FlowLens against *Flow, a network monitoring solution that takes advantage of programmable switches to export telemetry data. In a nutshell, *Flow relies on the data plane-level ASIC to extract and group multiple per-packet features in *grouped packet vectors*, ensuing their periodic offloading to a monitoring server. To compare the computation, storage, and bandwidth costs of both approaches, we intended to run the available *Flow code hosted on GitHub [38].

However, upon analysing the *Flow code repository, it became apparent that the available implementation corresponded to a redacted version of Tofino-specific code. In particular, the code for a number of target-specific functions was replaced by pseudocode and placeholders, since the original code cannot be disclosed due to NDAs. Despite this fact, however, the authors did not produce a `bmv2`-compliant version. Unfortunately, this meant that *Flow experimental setup could not be reproduced without further implementation efforts. Ultimately, we resorted to an analytical estimation of the resource expenditure of *Flow, showing that it incurs in a significant bandwidth overhead when compared to FlowLens.

Observation 6: At the time of writing FlowLens, it was difficult to experiment with existing traffic analysis tooling for programmable switching hardware due to the fact that target-specific P4 implementations were redacted prior to release to adhere to NDAs. As of today, this difficulty has been partially addressed by Barefoot and Intel through the Open Tofino [29] project.

VI. Recommendations

The lessons learned when tackling the several challenges tied to the implementation and evaluation of FlowLens allow us to draft three recommendations, below, related to experiment design and the sharing of experimental data and artifacts.

A. Target-aware system design

Our endeavors on adapting the implementation of FlowLens' FMA between P4-capable targets with different characteristics (Section III) put in evidence an important trade-off between fast prototyping and efficiency in real hardware. On the one hand, writing P4 programs is simpler if the target allows for greater levels of abstraction and flexibility, as is the case of software targets. On the other hand, developing P4 code while failing to adhere to existing hardware restrictions means that the program may require deep restructuring to be run successfully.

Recommendation 1: If you intend to deploy your system onto a specific target platform, design it in such a way you can respect the intrinsic target-specific restrictions. You should however try to abstract your design beyond these restrictions since the system may be adaptable to multiple target platforms.

B. Quality and availability of experimental data

Our quest towards finding network security applications that depended on the analysis of packet distributions, and whose classification workflow could be adapted to FlowLens (Section IV), revealed that several authors only make available datasets with post-processed traffic features. This is a stumbling block to the experimentation with traffic analysis frameworks that perform (and potentially improve) flow classification tasks based on raw packet characteristics, like FlowLens.

Recommendation 2: When releasing a dataset for ML-based network security research, consider to make available packet traces comprised of a timeseries of packet lengths and a timeseries of packet inter-arrival times. This may be helpful for practitioners interested in manipulating your data in unforeseen ways. Alternatives when there are business and confidentiality impediments that preclude full sharing of the raw packet traces exist, both in the form of privacy-enhanced techniques [26] or by exploring other models of collaboration with industry [25].

C. Sharing of experimental artifacts

Over the last few years, numerous researchers in computer security have striven to include pointers to experimental artifacts in their publications. As a whole, the dissemination of such artifacts (comprising tools, benchmarks, and datasets) provides evidence about the soundness of the experimental methodologies introduced in their papers, encourages other researchers to tinker with these artifacts, and serves as a jumping off point for future work. Still, the hindrances encountered during our efforts to compare FlowLens with other works in the literature (Section V) have shown that artifacts are oftentimes incomplete or otherwise cumbersome to experiment with.

Recommendation 3: Strive to make a working version of your code available, alongside thorough documentation. A prominent formal process that encourages the submission and evaluation of artifacts, and which has been progressively adopted in multiple conferences and journals, is the ACM Artifact Review and Badging initiative [4].

VII. Future Work

In this section, we outline some avenues for future work targeting the development of a comprehensive traffic analysis testbed for FlowLens and the experimentation with traffic analysis testbeds based on programmable switches.

Fully-fledged FlowLens testing suite: The experimental artifacts released alongside our FlowLens paper [9] include a P4 prototype of FlowLens' flow marker collection process and the necessary code for adapting three ML-based network security application to FlowLens. However, for our experiments, these software components are decoupled in such a way that the generation of flow markers and the ensuing application-specific classification tasks are simulated in a Python program. As future work, we intend to build a *fully fledged* software testbed that would implement the full FlowLens' workflow described in Section II-B. In short, this testbed would involve providing security practitioners with the ability to replay traffic traces (e.g., in the form of `.pcap` files) through the FlowLens P4 code running in the reference P4 software switch (`bmv2`) and execute the classification procedure in the switch control plane. This testbed can be built for the official P4 Tutorial VM [3], which already incorporates all the required simulation environment.

Distributed network testbed: The steep cost of programmable switching hardware makes it difficult for the average practitioner to develop and test software for such hardware. To make it worse, these costs are further exacerbated if a central point of the system design comprehends a distributed setting (e.g., requiring the cooperation of multiple switches to withstand a range of attacks to the network infrastructure [7]). Thus, an interesting direction for future work would be the development of a distributed testbed, much like PlanetLab, but enabled with programmable switches. In such a way, it would be possible to significantly drop the barrier to entry for network security practitioners. While a few research groups have already paved the way towards the creation of such a testbed [18, 5], existing proposals are still in their infancy and only recently began paving the way to provide global-scale connectivity [15].

VIII. Conclusions

This paper shed light on the major challenges involved in building and evaluating FlowLens, a traffic analysis system for generic ML-based network security applications. We detailed how we overcame the implementation hurdles caused by mismatches between the programming model of software and hardware P4 targets, the standardization efforts to ensure the compatibility of heterogeneous traffic analysis tasks with FlowLens, and the shortage of traffic analysis testbeds. We drafted a number of lessons and recommendations to researchers working at the intersection of the network security and machine learning fields, and presented directions for future work which aim at boosting the cooperation between security practitioners.

Acknowledgments

We thank the LASER program committee and workshop attendees for their insightful comments. This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) via the SFRH/BD/136967/2018 grant, and the PTDC/EEI-COM/29271/2017, UIDB/50021/2020, and PTDC/CCI-INF/30340/2017 (uPVN) projects.

References

[1] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, and Antonio Pescapé. Multi-classification approaches for classifying mobile app traffic. *Journal of Network and Computer Applications*, 103:131–145, 2018.

[2] Blake Anderson and David McGrew. Identifying encrypted malware traffic with contextual flow data. In *Proc. of the 9th ACM Workshop on Artificial Intelligence and Security*, pages 35–46, 2016.

[3] Antonin Bas. P4 Tutorial, https://github.com/p4lang/tutorials. Accessed: 2021-12-28.

[4] Association for Computing Machinery. Artifact Review and Badging – Version 2.0. https://www.acm.org/publications/policies/artifact-review-badging, 2021. Accessed: 2021-12-28.

[5] Ilya Baldin, Anita Nikolich, James Griffioen, Indermohan Inder S. Monga, Kuang-Ching Wang, Tom Lehman, and Paul Ruth. Fabric: A national-scale programmable experimental network infrastructure. *IEEE Internet Computing*, 23(6):38–47, 2019.

[6] Barefoot Networks. Tofino Switch https://www.barefootnetworks.com/products/brief-tofino/. Accessed: 2021-12-28.

[7] Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello, Fernando Ramos, and André Madeira. FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications. In *Proc. of the 28th Network and Distributed System Security Symposium*, 2021.

[8] Diogo Barradas, Nuno Santos, and Luís Rodrigues. Effective detection of multimedia protocol tunneling using machine learning. In *Proc. of the 27th USENIX Security Symposium*, pages 169–185, 2018.

[9] Diogo Barradas and Salvatore Signorello. FlowLens code repository. https://github.com/dmbb/FlowLens, 2020. Accessed: 2021-12-28.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[11] Broadcom. Tomahawk / BCM56960 Series, https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56960-series. Accessed: 2021-12-28.

[12] Calin Cascaval and Dan Daly. P4 Architectures, https://p4.org/assets/p4-ws-2017-p4-architectures.pdf. Accessed: 2021-12-28.

[13] Cisco. Cisco Encrypted Traffic Analytics Whitepaper, https://www.cisco.com/c/dam/en/us/solutions/collateral/enterprise-networks/enterprise-network-security/nb-09-encrytd-traf-anlytcs-wp-cte-en.pdf. Accessed: 2021-12-28.

[14] Baris Coskun and Nasir Memon. Online sketching of network flows for real-time stepping-stone detection. In *Proc. of the 25th Annual Computer Security Applications Conference*, pages 473–483, 2009.

[15] FAB Core Team. Fabric Across Borders. https://fabric-testbed.net/about/fab/, 2021. Accessed: 2021-12-28.

[16] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. Challenging the stateless quo of programmable switches. In *Proc. of the 19th ACM Workshop on Hot Topics in Networks*, page 153–159, 2020.

[17] John Geddes, Max Schuchard, and Nicholas Hopper. Cover your acks: Pitfalls of covert channel censorship circumvention. In *Proc. of the 20th ACM Conference on Computer and Communications Security*, pages 361–372, 2013.

[18] Paola Grosso, Cristian Hesselman, Luuk Hendriks, Joseph Hill, Stavros Konstantaras, Ronald van der Pol, Victor Reijs, Joeri de Ruiter, and Caspar Schutijser. A national programmable infrastructure to experiment with next-generation networks. In *Proc. of the IFIP/IEEE International Symposium on Integrated Network Management*, 2021.

[19] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *Proc. of the ACM Workshop on Cloud Computing Security*, pages 31–42, 2009.

[20] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. Programmable in-network security for context-aware BYOD policies. In *Proc. of the 29th USENIX Security Symposium*, pages 595–612, 2020.

[21] Y. Li, Y. Huang, R. Xu, S. Seneviratne, K. Thilakarathna, A. Cheng, D. Webb, and G. Jourjon. Deep content: Unveiling video streaming content from encrypted WiFi traffic. In *Proc. of the 17th IEEE International Symposium on Network Computing and Applications*, pages 1–8, 2018.

[22] Yair Meidan, Michael Bohadana, Yael Mathov, Yisroel Mirsky, Asaf Shabtai, Dominik Breitenbacher, and Yuval Elovici. N-BaIoT – network-based detection of IoT botnet attacks using deep autoencoders. *IEEE Pervasive Computing*, 17(3):12–22, 2018.

[23] Roland Meier, Petar Tsankov, Vincent Lenders, Laurent Vanbever, and Martin Vechev. NetHide: Secure and practical network topology obfuscation. In *Proc. of the 27th USENIX Security Symposium*, pages 693–709, 2018.

[24] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[25] Jeffrey C. Mogul, Priya Mahadevan, Christophe Diot, John Wilkes, Phillipa Gill, and Amin Vahdat. Data-driven networking research: Models for academic collaboration with industry (a google point of view). *SIGCOMM Computer Communication Review*, 51(4):47–49, 2021.

[26] Meisam Mohammady, Lingyu Wang, Yuan Hong, Habib Louafi, Makan Pourzandi, and Mourad Debbabi. Preserving both privacy and utility in network trace anonymization. In *Proc. of the 25th ACM Conference on Computer and Communications Security*, pages 459–474, 2018.

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

[28] Milad Nasr, Amir Houmansadr, and Arya Mazumdar. Compressive traffic analysis: A new paradigm for scalable traffic analysis. In *Proc. of the 24th ACM Conference on Computer and Communications Security*, pages 2053–2069, 2017.

[29] Barefoot Networks. Open Tofino. https://github.com/barefootnetworks/Open-Tofino, 2020. Accessed: 2021-12-28.

[30] Thuy Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys and Tutorials*, 10(1-4):56–76, 2008.

[31] TJ O'Connor, Reham Mohamed, Markus Miettinen, William Enck, Bradley Reaves, and Ahmad-Reza Sadeghi. Homesnitch: Behavior transparency and control for smart home iot devices. In *Proc. of the 12th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 128–138, 2019.

[32] p4language Consortium. P4 Behavioral Model, https://github.com/p4lang/behavioral-model. Accessed: 2021-12-28.

[33] Claude Sammut and Geoffrey I. Webb, editors. *Cross-Validation*, pages 249–249. Springer US, 2010.

[34] Claude Sammut and Geoffrey I. Webb, editors. *Holdout Evaluation*, pages 506–507. Springer US, 2010.

[35] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *Proc. of the 26th USENIX Security Symposium*, pages 1357–1374, 2017.

[36] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *Proc. of the 26th USENIX Security Symposium*, pages 1357–1374, 2017.

[37] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proc. of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2016.

[38] John Sonchack. StarFlow. https://github.com/jsonch/StarFlow, 2018. Accessed: 2021-12-28.

[39] John Sonchack, Oliver Michel, Adam Aviv, Eric Keller, and Jonathan Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. of the USENIX Annual Technical Conference*, pages 823–835, 2018.

[40] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic. In *Proc. of the 1st IEEE European Symposium on Security and Privacy*, pages 439–454, 2016.

[41] Jiarong Xing, Qiao Kang, and Ang Chen. NetWarden: Mitigating network covert channels while preserving performance. In *Proc. of the 29th USENIX Security Symposium*, pages 2039–2056, 2020.

[42] Junhua Yan and Jasleen Kaur. Feature selection for website fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2018(4):200–219, 2018.