IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 11 July 2022, accepted 22 August 2022, date of publication 29 August 2022, date of current version 2 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3202644

RESEARCH ARTICLE

A Control Plane Enabling Automated and Fully Adaptive Network Traffic Monitoring With eBPF

SIMONE MAGNANI[1,2], (Member, IEEE), FULVIO RISSO[3], (Member, IEEE), AND DOMENICO SIRACUSA[1]

[1]Cybersecurity, Fondazione Bruno Kessler, 38121 Trento, Italy
[2]DIBRIS Department, Università di Genova, 16145 Genova, Italy
[3]DAUIN Department, Politecnico di Torino, 10129 Turin, Italy

Corresponding author: Simone Magnani (smagnani@fbk.eu)

ABSTRACT The *extended Berkeley Packet Filter (eBPF)* enables the dynamic injection of user-defined processing logic at run-time in the Linux networking stack without disrupting any active monitoring process. This enables the selective extraction of only the traffic features that are needed in a given instant of time, which is what we define *fully adaptive* network traffic monitoring. However, eBPF programs require ad-hoc control plane routines for each specific scenario in order to orchestrate the underlying data plane and export the required metrics, resulting in potentially duplicated source codes to maintain, and creating the risk of deploying, at runtime, unverified user-defined code that controls the devices running the monitoring process. This paper presents a control plane that automatically adapts both its management tasks and data extraction methodologies based on the underlying data plane provided by the user, who can merely focus on the monitoring logic definition. The paper evaluates the performance of the control plane's modules and demonstrates the advantages, in terms of processing speed and memory consumption, of a fully-adaptive monitoring approach with respect to nProbe (a state-of-the-art solution), an *adaptive* and a *non-adaptive* methodology in eBPF. Experiments prove that the control plane monitoring options do not significantly affect the underlying data plane (0.15% degraded throughput) and leverage the most efficient extraction primitives (20x faster execution time). Moreover, the fully-adaptive monitoring leads to a higher number of processed packets (10x) and significantly lower memory occupancy (10x) when extracting the smallest set of features.

INDEX TERMS Adaptive monitoring, control plane, data plane, eBPF, network traffic monitoring.

I. INTRODUCTION

The increasing need for computation capabilities at the edge of the infrastructure (e.g., outside data centres or enterprise networks) has reshaped the concept of network perimeter and spurred the emergence of zero-trust architectures [1], posing more severe challenges for network monitoring tools, which need to adapt to dynamic and heterogeneous environments including devices with different capacities (e.g., powerful servers, end-hosts, small IoT gateways, mini PCs), capabilities (e.g., machines with GPUs or specific architectures), locations (on-premise, remote), and more. To scale up with the enormous amount of potential scenarios, infrastructure managers should have the possibility to define the logic of monitoring functions in an infrastructure-agnostic way and should be able to deploy them with unprecedented levels of adaptability, effectiveness, and automation.

Conventional monitoring tools present a control logic pre-tailored to the extraction primitives they employ, resulting in ad-hoc monolithic systems that accomplish scenario-specific purposes. In addition, many solutions build upon proprietary non-native drivers that not only need to be constantly maintained according to the underlying operating system updates, but they also re-implement part of the networking logic present in the kernel of the system to preserve the default network packets routes. Moreover, such *non-adaptive* solutions statically define the nature of gathered data in terms of amount, type, and granularity of information

The associate editor coordinating the review of this manuscript and approving it for publication was Jon Atli Benediktsson.

extracted from packets throughout the entire monitoring session, resulting in a constant and fixed logic. This methodology not only has a noticeable impact on high-speed networks due to its unnecessary additional memory consumption and processing overhead, but it might also be too heavy to be supported by a resource-constrained device with old/low requirements, and, therefore, unusable in many solutions.

A partial solution to the above problem consists in what we call *adaptive* network monitoring, a methodology for enabling the dynamic activation of features through additional support data structures used for selecting, among a long and immutable predefined list, which features the monitoring program extracts from network traffic. Despite enabling a finer tuning of the computational power at runtime, this methodology still implies a non-negligible cost in terms of processing overhead, as the monitoring system needs to keep consulting the support structures to decide whether to extract or not each feature, and even conditional statements and memory loads are expensive operations in high-speed packet processing environments. More importantly, this solution prevents extending the extraction pipeline, at least not without a cold restart of the system, thus limiting the flexibility of the monitoring logic.

On the other hand, what we call *fully-adaptive* methodology introduces the possibility to precisely define the set of information to be gathered, characterising custom features (even more complex than a simple counter) and the associated extraction logic, and applying such configurations at runtime without disruptions to monitoring operations. The *extended Berkeley Packet Filter* (*eBPF*) [2] enables such monitoring by providing a virtualized in-kernel environment that, thanks to its companion compiler and verifier, defines a secure and safe boundary where to extend the monitoring pipeline by adding and removing custom user-defined programs. Differently than the traditional Linux Traffic Control (TC) [3] networking layer hook, combining eBPF with the *eXpress Data Path (XDP)* [4] allows moving the entire execution of the monitoring program to the lowest layer of the operating system, even directly within the network card interface itself (if supported), resulting in early (and efficient) processing of network packets.

As a matter of fact, each eBPF data plane program needs a sibling control plane for management tasks, including accessing and exporting the content of the monitoring values extracted from network traffic and contained in specific data structures. However, eBPF only provides a safe sandbox for the compilation and execution of the data plane program, which means that any remote device under monitoring should accept user-defined and unverified control plane routines that might potentially harm the system. Moreover, control plane programs referring to different scenarios still share a high portion of control routines for accessing monitoring data. This does not only worsen the software maintainability, as infrastructure managers have to maintain and update different source codes, but it also potentially introduces flaws and latency due to an ineffective (or incorrect) extraction methodology.

Taking all these considerations into account, our key idea is to supply a control plane to limit the tasks of infrastructure managers to the mere creation of the data plane logic, by providing verified and optimised management and extraction routines, which can be safely pushed into the remote device to monitor alongside the user-defined data plane to supervise and export its gathered data. More specifically, this work introduces two major contributions:

- First, it proposes a control plane to enable and support the *fully-adaptive* approach in remote monitoring devices through exposing unified methods to interact with the system, which allow to (*i*) dynamically receive the data plane configuration (i.e., eBPF monitoring program), (*ii*) safely compile and inject it into the monitoring pipeline of the system, and (*iii*) automatically access and export the defined metrics from the monitoring program to the requesting entity (e.g., user or automated system). We discuss the adoption of such controller and evaluate its monitoring options with respect to a baseline version, in terms of performance degradation.
- Second, it evaluates the *fully-adaptive* network monitoring methodology and compares it against nProbe [5], a NetFlow-compatible [6] implementation, and the *adaptive* and a *non-adaptive* approach in eBPF, reporting significant advantages of adopting such methodology, in terms of number of network packets processed and memory consumption.

The rest of the paper is structured as follows: Section II reviews and discusses related works. Section III enriches the introduction by presenting the motivations behind our research. Then, Section IV provides information concerning the proposed methodology, the adopted architecture and monitoring options. The experimental validation and the comparison with state-of-the-art methodologies is described in Section V. Finally, Section VI outlines future research directions and extensions to this work, while Section VII concludes the paper.

II. RELATED WORK

In this section, we present some application of both eBPF and non-eBPF based state-of-the-art monitoring solutions, pointing out the difference with our approach in terms of monitoring logic definition and injection. Works that analyse gathered data for any further purposes (e.g., detect cyberattacks or gather insights on network performance) are therefore willingly omitted.

A. eBPF-BASED APPLICATIONS AND APPROACHES

The creation of programmable data plane monitoring programs in-kernel led many researchers to investigate the effects, in terms of both performance improvement/degradation [7] and programmability/feasibility [8], of adopting eBPF-based solution to replace state-of-the-art

approaches and monitoring functions. In fact, not only communities [9], [10] and open-source frameworks for observability (e.g., Cilium[1]) constantly grow, but also many companies including Google, Microsoft, Facebook, Cloudfare,[2] and Sysdig[3] re-designed an optimised version of many applications in eBPF, such as firewalls, load-balancers, and more. For instance, in [11] authors use eBPF to efficiently replace *iptables*,[4] a well-known firewall and traffic management tool for Linux systems. Additional applications include safeguarding users privacy while using common domain resolution protocols [12], network congestion [13], [14] and fault [15] detection, network traffic mirroring [16], the deployment of mobile gateway for 5G networks [17], identification and performance tuning of a Redis database [18], *Intrusion Detection and Prevention Systems* [19], [20], and the detection of specific cyberattacks, such as for *Distributed Denial of Service* [21] or *Water Torture* [22].

The authors of [23] and [24] propose elastic in-kernel transparent monitoring systems for microservices observability using fine-grained indicators, which allow inspecting internal states of microservice instances. A similar purpose is achieved in [25] for locating insects of troubles in a network. Moreover, in [4] authors partially address few limitations we discussed in Section I, such as the importance of having a non-intrusive monitoring logic, by combining eBPF with XDP and extracting ad-hoc features to reduce the resource footprint of software network analytics.

However, all the mentioned applications share at least one of the following constraints. First, they couple the data plane monitoring logic with a static control plane specific for the application they are designed for, resulting in having a multitude of similar code replicated for each scenario. Second, the majority of these solutions builds upon precompiled programs that are not meant to change throughout the entire execution of the software, resulting in a bounded monitoring logic. The remaining ones support the dynamic tuning of the extraction logic, but only between the previously defined ones as they do not accept external user-defined code, resulting in a limited adaptive monitoring.

B. NON-eBPF BASED APPROACHES

Concerning non-eBPF technologies, the de-facto monitoring protocols are NetFlow and IPFIX [26], its open counterpart. A famous compliant implementation used also as an evaluation metric in this paper is nProbe [5], which relies on different underlying packet processing technologies (PF_RING[5] and libpcap[6]). However, it does not support changing the monitoring pipeline at runtime, hence extracting a constant set of features from the traffic.

Other approaches focus on different adaptive properties. In fact, they propose to adapt the monitoring frequency [27], [28], the granularity of traffic aggregate [29], [30], [31] (e.g., the session identifier used to group packets), and the features collected for monitoring [32], [33], [34] depending on decisions that rely on the undergoing networking situation. In [35], the authors propose a similar approach that enables the dynamic data plane program reconfiguration, but it builds upon a different technology (P4[7]) that restricts data structures and extraction operations to traditional flow-table based aggregates, limiting the nature of the network monitoring. Finally, a solution for data centres network event monitoring at full line rate is proposed in [36], but it supports only a limited set of features and event definition.

The most important limitation that all these methodologies suffer consists in a constrained choice of features among only the ones defined before starting the monitoring session, as they do not support dynamic program recompilation and injection. This results in having a bounded piece of software that can support only the foreseen and designed logic at most. Finally, the purpose of such methodologies is limited to a single application-specific scenario, resulting in monolithic solutions.

On the other hand, we propose an architecture that overcomes all the limitations of conventional network monitoring technologies, enabling a fully-adaptive network monitoring system that requires only the desired processing logic to reconfigure both the monitoring pipeline and all management tasks, including the automatic export of the monitoring metrics to the user.

III. MOTIVATIONS

We envision the deployment scenario depicted in Fig. 1. Differently than a traditional approach whose feature-gathering process remains constant among the execution of the monitoring session, in this scenario we foresee the deployment of smart algorithms that analyse the collected data and determine whether it is rich enough for the undergoing analysis. In this case, many network nodes may participate in the features-gathering process, which are orchestrated by an apposite controller, likely running on a different node, which is then required to inject the updated remote monitoring probes on the gathering devices. When more data is required (e.g., more features referring to the traffic coming from a specific set of network sources), the monitoring probe (hence data and control plane) is updated and then injected in the remote network nodes of interest. Such probes, distributed across the infrastructure, gather data and send it to a (centralized) data collector that provides the way to perform further data analysis (monitoring, security, etc).

The variability of the source monitoring code is a pillar in such a dynamic environment, where the element taking decisions can potentially adjust the granularity of the analysis

[1] https://cilium.io/
[2] https://www.cloudflare.com/
[3] https://sysdig.com/
[4] https://linux.die.net/man/8/iptables
[5] https://www.ntop.org/guides/pf_ring/
[6] https://github.com/the-tcpdump-group/libpcap
[7] https://opennetworking.org/p4/

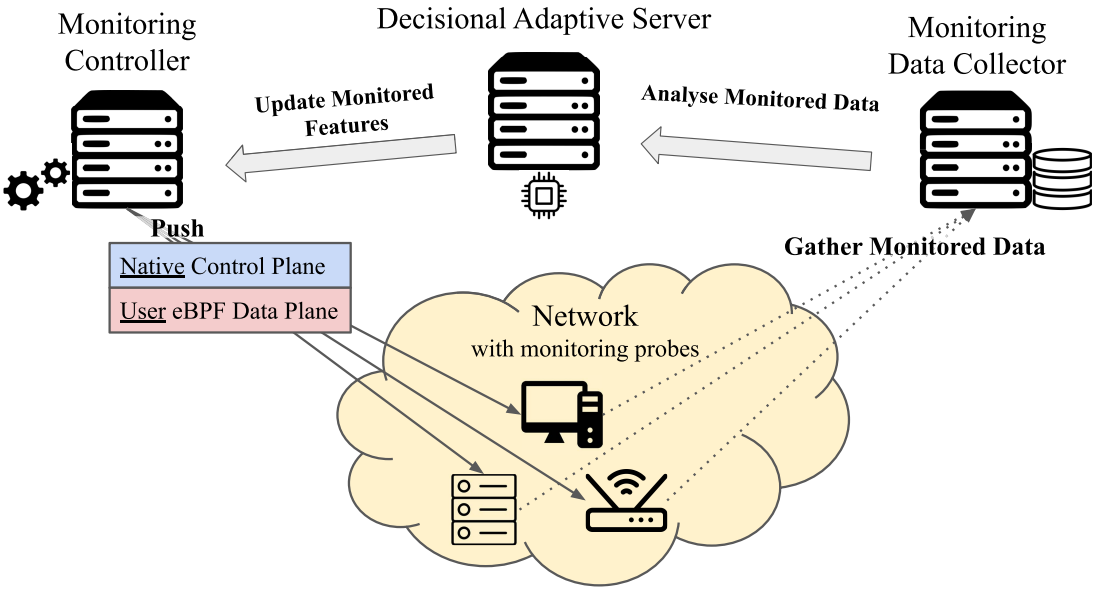


FIGURE 1. Possible deployment scenario.

very frequently. Within this scenario, our research aims at addressing the following needs:

- *Simplification*: every application that leverages eBPF for network monitoring requires a data plane program and a control plane to supervise it and extract its gathered data from specific data structures. However, those two primitives result extremely similar among different monitoring applications, differing by only the reference to the actual data structure. As a result, infrastructure managers need to maintain duplicated source control plane codes for every deployment scenario, which is clearly inefficient and expensive in terms of time and resources. Our control plane provides a generalised version of such functions that can automatically adapt to the underlying data plane program, inferring the correct (and most optimised) extraction methodology for the specific metric, leaving users focusing on the mere data plane logic definition.
- *Runtime Customisation*: features engineering is a well-known state-of-the-art problem that cannot be defined in a unique global solution, as every application aims at monitoring different parameters to achieve the best results. We provide a generalised network monitoring methodology, so that it can be deployed in different monitoring scenarios, just by providing the data plane configuration, and letting the rest of the architecture automatically adjust the management tasks accordingly. Our solution does not only allow defining completely customized features within the monitoring pipeline, but it also supports dynamic data plane updates so that the set of instructions and features can be enlarged/restricted as needed, without disruptions to the monitoring process.
- *Heterogeneity*: monitoring probes can be installed on different devices within the network, such as servers, data centres, end-user laptops, switches, routers, both physical or even virtualised resources. As a consequence, to be fully-compatible with a cloud-native and distributed scenario, probes must be versatile and ready to be used, independently by the underlying architecture, which must only support eBPF. This requires a control plane that supervises their execution throughout their lifecycles, and transparently addresses issues arisen by the different systems specifications and support (e.g., different system calls, missing operations on data structures, etc.).
- *Safety*: while eBPF guarantees that the code injected at run-time is safe and cannot harm the system, no such a sandbox exists for the control plane running in user-space, which has to be tailored to the data produced by the underlying monitoring code, hence dynamically updated as well. As a consequence, in a deployment scenario with strong safety requirements in which user-defined ad-hoc routines cannot be accepted, the control plane needs to properly interact with any eBPF data plane program without requiring any source code update, dynamically re-configuring its extraction primitives at run-time, and it must be verified and guaranteed to be safe ahead-of-time.

IV. METHODOLOGY

This section presents the envisioned control plane architecture, discussing all the design choices dictated by the needs introduced in Section III, and a typical interaction workflow between a user and the system. Finally, we also discuss additional monitoring options needed to offer a more complete

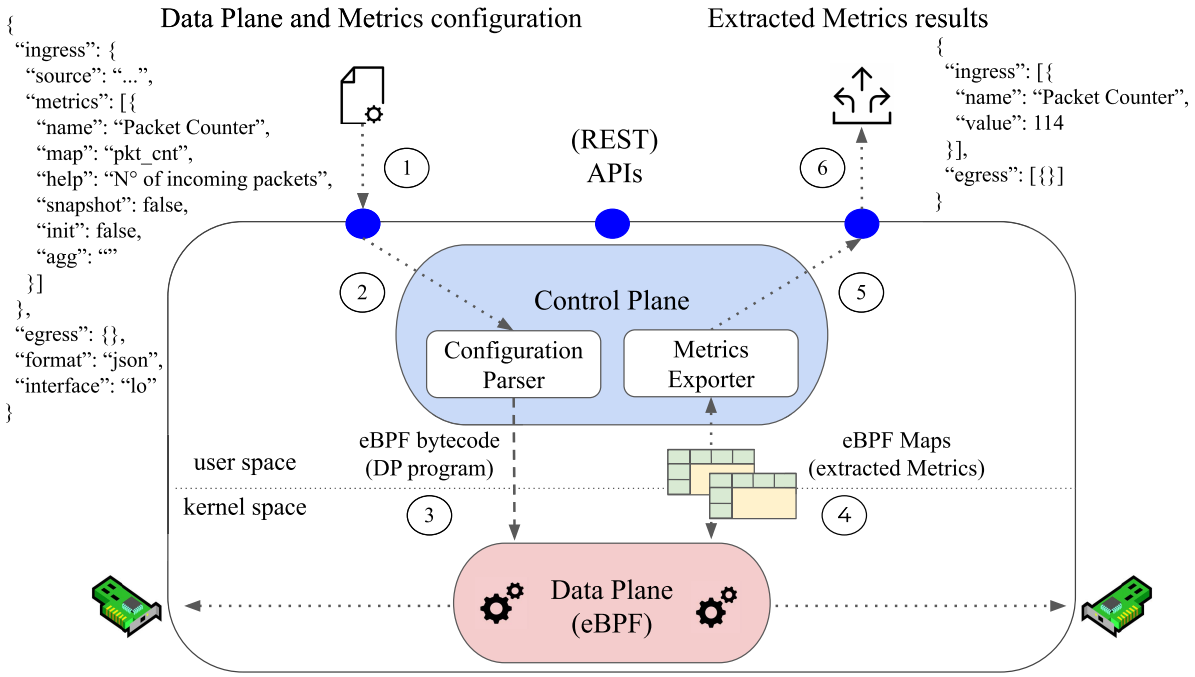


FIGURE 2. Envisioned architecture and interaction workflow.

support for different types of monitoring, presented later in this section.

More details about possible implementations of the proposed architecture are available in some open-source projects[8] [9] we contributed to, which helped us to verify the feasibility and completeness of our architecture, and served as main environment for easily deploying our monitoring probes for the evaluation. We also publish the code for replicability of tests in the following open source repository.[10] Please note that results are strictly related to the used testbed, hence slight differences on the setup may lead to completely different results.

A. ARCHITECTURE

We propose the architecture depicted in Fig. 2. Differently from many state-of-the-art approaches where both the monitoring logic and the set of control functions are performed by a unique monolithic component, we prefer the data and control planes to be decoupled, as it prevents that additional (and potentially slow) user-space operations (e.g., system calls) interfere with the monitoring process and affect its performance.

The data plane includes the kernel space functions and processes that analyse packets coming from network interfaces according to the user-defined monitoring logic. While conventional technologies support only dynamic feature selection, consisting in selecting monitored features among a predefined and immutable list, the eBPF re-programmability allows performing entire dynamic user-defined programs replacement, meaning both features (either simple counter or more complex personalized ones), operations to be applied on network traffic, and entire data structures definitions. Each program defines the required tables to store the data gathered from the analysis (*eBPF maps*), which are very efficient in-kernel data structures that can be accessed also from user space, allowing the control plane to read and export their values afterwards. Each map is characterised by (*i*) the type (e.g., FIFO, LIFO, Hash); (*ii*) the precise structure for their entries (e.g., key as a simple integer or as a more complex structure); (*iii*) the number of entries; and (*iv*) the visibility level that either enables to share maps among different eBPF programs or to keep them local to the current one.

The data plane component is provided within a monitoring configuration (step 1 in Fig. 2) containing, for each direction of the traffic to analyse (`ingress` and `egress`, using TC in case of outgoing traffic as XDP is not yet supported[11]) from a specific network interface (`interface`), the source code to be compiled and injected (`source`), and a list of metrics to be exported as a result of the analysis (`metrics`) using the required data interchange format (`format`, such as JSON or XML). Each metric is characterised by (*a*) a name (`name`), (*b*) the name of the corresponding underlying eBPF map (`map`), (*c*) a brief description (`help`), (*d*) two variables (later discussed in Section IV-B) that influence the nature of the access to the map (`snapshot`) and whether

[8] https://github.com/dechainers/dechainy
[9] https://github.com/polycube-network/polycube
[10] https://github.com/s41m0n/opportunistic_monitoring
[11] https://lwn.net/Articles/813406/

to re-initialise its content or not (`init`), and (*e*) a variable used only in case of *PERCPU* eBPF maps,[12] whose values are scattered across multiple arrays, in case only the aggregate result needs to be exported, computed using the algorithm selected from those supported by the control plane (`agg`, such as averaging or summing values). Furthermore, all the eBPF maps declared in the source code that do not correspond to a metric specified by the user cannot be accessed from outside the system for security reasons, to prevent exporting sensitive data (e.g., internal data required for connection tracking).

The duties of a control plane that must interact with any user-defined monitoring program include the capability to dynamically compile and inject such programs in the system, but most importantly it has to correctly (and efficiently) interact with any eBPF program without requiring a source code update, extract the kernel-gathered data and return it back to the user using the desired format. As a result, users need to supply only the data plane configuration (i.e., monitoring code and the list of metrics to be exported), which can be even replaced at runtime by providing a new one, and the controller automatically re-adjusts the monitoring-and-extraction pipeline. While the north-bound interface of the control plane (blue circles referred as APIs in Fig. 2) enables such interactions between users and the system by exposing (REST[13]) APIs, the two underlying macro-modules designed to handle those requests are:

- *Configuration Parser*: it parses the received data plane configuration and prepares the desired monitoring pipeline accordingly. In particular, it extracts and compiles the eBPF source code representing the monitoring probe, and injects the resulting executable in the system. During such phase, it uses the standard eBPF compiler and verifier to check the correctness of the program and compile it according to the platform, while it manually handles the remaining parameters in the configuration.
- *Metrics Exporter*: it provides a unified and standardised method to retrieve the defined monitoring metrics according to the received data plane configuration. Extracting those values from the underlying data plane program requires taking into account all the (*i*)-(*iv*) characteristics of each map presented above, plus the possible monitoring options for each map (i.e., snapshot retrieval and content re-initialisation). This component retrieves at runtime this information from the received monitoring configuration, and infers the apposite and most optimised extraction method to use, presented in Algorithm 3. Finally, it converts the values extracted from the eBPF maps into the specified data interchange format, requiring additional conversion depending on the original data type (e.g., integer value to string, a user-defined structure to a key-value formatted string).

[12]https://lwn.net/Articles/674443/
[13]https://restfulapi.net/

B. MONITORING OPTIONS

Among all the possible use cases and monitoring needs (e.g., flow aggregation or per-packet analysis), we identify the following two properties that characterise the nature of the network monitoring, hence requiring additional support in the control plane:

- Timing
  - Incremental analysis: statistics and features are incrementally updated throughout the monitoring session, operating on values that are never reset and computed since the beginning of the session;
  - Time-window based analysis: the system computes statistics and features within a specified time-window, which forces the system resetting such values when a new one begins.
- Consistency:
  - Snapshot access: when accessing one of the data structures defined within the monitoring session, the system ensures that none of the remaining ones are updated in the meantime, preventing inconsistent states and offering a fair snapshot of monitoring values (e.g., two structures are strictly related, such as an internal session tracker and its public accessible counterpart);
  - Independent access: the access to one of the monitoring data structures does not require locking all the other ones defined within the program, offering direct access to data (e.g., access to data structure *X* containing information from the network packets does not require locking the entire structure *Y*, whose values represent firewalling rules).

To enable such different monitoring natures, a control plane must support at least the following options that users can specify when providing the monitoring configuration.

1) SNAPSHOT METRIC RETRIEVAL

Since data and control plane are fully decoupled, it may happen that while the control plane retrieves certain values from a specific eBPF map, the data plane updates other entries from the same map or from a different one that is strictly related to the one handled by the control plane. For instance, a program might use a private session tracker map and its public counterpart, containing only a subset of values allowed to exit the monitoring system. While this behaviour can be acceptable for some monitoring applications, in other cases all the gathered metrics should be consistent, hence raising the need for a snapshot data retrieval.

A swappable dual-map approach allows the control plane to retrieve data from a first eBPF map that represents a snapshot of the traffic at a given time, while the data plane program keeps analysing packets and computing the traffic information within a second (equivalent) map. Whenever the control plane needs to retrieve data, it swaps the two maps. However, as pointed out in [8], the dual-map approach turns out to affect the performance of the monitoring program, due

to a significantly high swapping latency in eBPF. As a result, we promote a dual-program approach.

The approach consists in cloning and compiling a second eBPF program equal to the original one, each program using its own data structures. Once the controller successfully compiles and injects both the two programs in the system, it activates only one of them, which will perform the analysis on the network traffic and store metrics on its own maps. When the metrics need to be exported, the system swaps the currently active program with the unloaded one by changing the index of the program to be invoked from a specific map, which is a simple and atomic operation, hence with supposed small impact on the system in terms of packets loss. The semantic of the original data plane program injected is preserved thanks to an intense precompile stage performed by the Configuration Parser, which ensures that references to the eBPF maps and the sequence of operations are correctly preserved throughout the entire monitoring.

On the other hand, keeping aligned both sets of data maps belonging to the two programs with the same values would imply an excessive swapping latency, as values from the old maps should be copied into new ones, while at the same time stopping the network monitoring process, which is clearly undesirable. As a result, such attribute likely implies that metrics are periodically reinitialised, resulting in a time-window based network monitoring. Moreover, while it addresses issues typical of transactional systems, the issues of multi-access to the resources still needs to be handled by the implementation of the controller, which must ensure that no concurrent external entities access the same data via apposite thread-lock and mutual exclusion mechanisms.

2) METRIC REINITIALISATION

In a time-window based network monitoring, data structures need to be reinitialised at the beginning of each monitoring interval, after the system retrieves and exports their values. This requires the Metrics Exporter module to erase the content of the eBPF maps, so that the data plane program can collect monitoring metrics referring to the new time-window starting with clean counters. Even though this operation is intuitively easy to perform, its actual implementation is less straightforward because (*i*) the control plane has to invoke the correct reset mechanism for each type of map (which, for performance reasons, should be selected at compile time), and (*ii*) this operation could potentially raise concurrency issues between data and control plane (e.g., data plane updating some counters while the control plane is still clearing the remaining part of the table). While the concurrency issues can be partially solved by enabling the snapshot metric retrieval, invoking the correct reset method requires taking into account all the properties of the eBPF maps under analysis. Unfortunately, standard eBPF does not provide a unique reset method usable with all data structures, hence the control plane carefully selects the correct (and optimised) one for each metric, as later presented in Algorithm 3.

On the other hand, supporting all the reset methods that eBPF offers to empty its data structures allows the Controller (i.e., its Metric Exported module) to properly use the most efficient implementation, such as for the batch operations, which enable faster and atomic accesses to the maps by acquiring the lock only once, instead of iterative and slow routines. As a result, especially when dealing with huge maps, those methods bring significant advantages to the controller, which can then process faster all the incoming requests. Again, to preserve the monitoring performance especially in high-speed environments, this attribute should be combined with the snapshot metric retrieval. In fact, despite all the optimisations, every operation on the same data structure used simultaneously with the data plane program could degrade its performance when handling traffic at line rate, while the snapshot retrieval allows operating on the data structure belonging to the unloaded program, leaving the monitoring process handling packets undisturbed.

C. WORKFLOW AND ALGORITHMS

A typical interaction workflow between an external entity and the fully-adaptive monitoring system consists of the 1-6 steps depicted in Fig. 2:

1) The user, or an external automated system, interacts with the system by providing the monitoring configuration to the controller, using the dedicated API.
2) Once received the configuration, the control plane leverages its Configuration Parser module to parse it, extract the source code of the monitoring program to be compiled, and executed apposite routines related to the monitoring options specified for each metric.
3) The controller compiles the extracted source code into bytecode and checks its correctness using the standard eBPF compiler and verifier, and it finally injects the executable safe probe in the kernel, if no errors arise. From this point on, the monitoring code will be executed on each incoming/outgoing packet, depending on the configuration.
4) The data plane program stores monitoring data in the proper set of declared eBPF maps, which represent the only communication channel with the control plane.
5) When the user issues a request to read the collected metrics using the proper API, the control plane invokes its Metrics Exporter module, which is in charge of retrieving data from the underlying maps and converting it. During this phase, the controller checks whether monitoring metrics need to be re-initialised and if the data plane program needs to be dynamically substituted with its cloned version (i.e., snapshot data retrieval attribute enabled).
6) Finally, the controller returns output data to the user using the desired format.

We therefore propose the main procedures that the controller recalls while performing the mentioned steps in the workflow. To start with, Algorithm 1 describes the *parseConfiguration(C)* procedure that represents the controls and

Algorithm 1 parseConfiguration(C), Controller Procedure to Parse the Received Data Plane Configuration

Input: C, the data plane configuration
Output: P, the resulting monitoring probe

```
1:  P ← Probe()
2:  for H in [“ingress”, “egress”] do
3:      C[H].cloned ←””
4:      P[H].snapshot ← FALSE
5:      for M in C[H].metrics do
6:          if M.snapshot then
7:              P[H].snapshot ← TRUE
8:              C[H].cloned ← C[H].source
9:              break
10:         end if
11:     end for
12:     if P[H].snapshot is TRUE then
13:         for L in findDeclarations(C[H].source) do
14:             if L in C[H].metrics then
15:                 mapFictitious(C[H].cloned, L, randName())
16:             else
17:                 mapShared(C[H].source, C[H].cloned, L)
18:             end if
19:         end for
20:         P[H].inactive_prog ← compile(C[H].cloned)
21:     end if
22:     P[H].active_prog ← compile(C[H].source)
23: end for
24: return P
```

Algorithm 2 exportMetrics(N), Controller Procedure to Extract and Export Monitoring Metrics

Input: N, the name of the monitoring probe
Output: V, the content of all monitoring metrics

```
1:  V ← {“ingress” : {}, “egress” : {}}
2:  for H in [“ingress”, “egress”] do
3:      T ← Ctr.probes[N][H]
4:      if T.snapshot is TRUE then
5:          T.changeActiveEbpfProgram()
6:      end if
7:      E ← T.inactive_prog
8:      for M in T.metrics do
9:          if M.exportable then
10:             V[H][M] ← extractMetric(E, M)
11:             if E[M].type is PERCPU and M.agg is not ”” then
12:                 aggregateValues(V[H][M], M.agg)
13:             end if
14:         else if M.reinitialise then
15:             eraseMetric(E, M)
16:         end if
17:     end for
18: end for
19: return convertToFormat(Ctr.probes[N].format, V)
```

operations performed in steps 1-3 once received the data plane configuration C, to create the final probe P. Such algorithm ensures support for both in/out traffic, while we omit other possible low-level details, such as mutual exclusion, invalid references and other minor controls. Note that the algorithm controls whether at least one metric requires the snapshot retrieval attribute, which will force the generation of a new copy of the program with fictitious data structures, while preserving all those without such attribute between the two compiled programs. If this attribute is active, then the resulting probe will contain two executables, which alternatively swaps at every metrics retrieval.

In addition, Algorithm 2 describes the metrics-extraction procedure more in general, while Algorithm 3 describes how the controller extracts the actual monitoring values from the underlying eBPF maps, using the standard system calls associated to the map type, and applying controls belonging to the additional monitoring options enabled. In Algorithm 2, the controller first checks whether the requested probe exists, and if the underlying monitoring program needs to be swapped, meaning that at least one metric requested the snapshot retrieval. Then, for each metric to be exported, it executes the Algorithm 3, a more in-depth procedure, consisting in selecting the proper set of functions to call according to the eBPF map of interest. Thanks to an analysis of the eBPF support in the Linux kernel, including the more enhanced *batch operations* that allow accessing the entire content of the map at once preventing iterative and slower accesses, we support all the possible cases included in the algorithm. Moreover, we optimise this procedure by referring to the right operations to be invoked also considering whether the map needs to be reinitialised or not, to avoid accessing it twice. We omit the description of the *eraseMetric(E, M)* procedure in Algorithm 2 used for metrics that do not need to be exported, as it performs the same exact operations listed in Algorithm 3 without retrieving their values, but directly calling the apposite delete method (i.e., *deleteBatch()* or *delete()*) on the eBPF map.

Finally, the controller checks whether the value of each retrieved metric needs to be aggregated, as it belongs to the special PERCPU eBPF map category. In that case, the value would actually be an array of values, each one representing a single core of the CPU. The controller supports simple aggregation algorithms such as *averaging* or *summing* all the values, which needs to be specified inside the monitoring configuration alongside the specific metric to aggregate. Once terminated, it returns the monitoring metrics converted according to the specified format in the configuration.

V. EXPERIMENTAL VALIDATION

This section presents (*a*) the validation of the monitoring options introduced in Section IV-B and (*b*) comparisons of the fully-adaptive network monitoring approach with an *adaptive* and a *non-adaptive* methodology. The advantages of a

Algorithm 3 extractMetric(E, M), Controller Procedure to Extract the Value of a Monitoring Metric From an eBPF Program

Input: E, the inactive eBPF program in a probe
Input: M, the metric to be extracted
Output: V, the content of the monitoring metric

$V \leftarrow NULL$
if E[M].type is FIFO or LIFO **then**
  $V \leftarrow []$
  for i = 0 **to** E[M].size **do**
    $V.append(lookupAndDelete(E[M]))$
  end for
else if E[M].type is ARRAY **then**
  $V \leftarrow []$
  if OS_SUPPORT_BATCH() is TRUE **then**
    $K \leftarrow [0, 1, \ldots E[M].size - 1]$
    $V \leftarrow lookupBatch(E[M], K)$
    if M.reinitialise **then**
      $B \leftarrow Array(E[M].zero, size = E[M].size)$
      $updateBatch(E[M], K, B)$
    end if
  else
    for $i = 0$ **to** E[M].size **do**
      $V.append(lookup(E[M], i))$
      if M.reinitialise **then**
        $update(E[M], i, E[M].zero))$
      end if
    end for
  end if
else if E[M].type is HASH **then**
  $V \leftarrow \{\}$
  if OS_SUPPORT_BATCH() is TRUE **then**
    if M.reinitialise **then**
      $V \leftarrow lookupAndDeleteBatch(E[M])$
    else
      $V \leftarrow lookupBatch(E[M])$
    end if
  else
    $K \leftarrow NULL$
    while $(K \leftarrow E[M].nextKey(K))$ is not NULL **do**
      if M.reinitialise **then**
        $V.append(lookupAndDelete(E[M], K))$
      else
        $V.append(lookup(E[M], K))$
      end if
    end while
  end if
end if
return V

non-invasive monitoring are evaluated from (*i*) the CPU and (*ii*) memory consumption points of view. We willingly omit testing all the properties unrelated to network monitoring, such as the performance of the framework (web server) chosen for providing REST APIs.

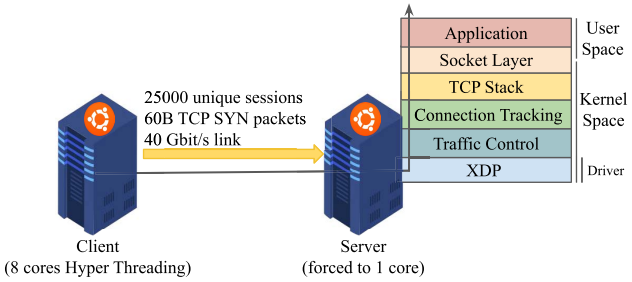


FIGURE 3. Test bed setup.

The testbed depicted in Fig. 3 includes two physical client-server machines running Ubuntu Server 18.04.3 LTS (x86_64), kernel 5.8.0-43-generic, provided with an Intel(R) Xeon(R) E3-1245 v5 3.50 GHz CPU, 64 GB DDR4 RAM, and dual Intel® Ethernet Converged Network Adapter XL710 10/40 GbE with the i40e driver for the XDP programs injection (XDP_DRV).

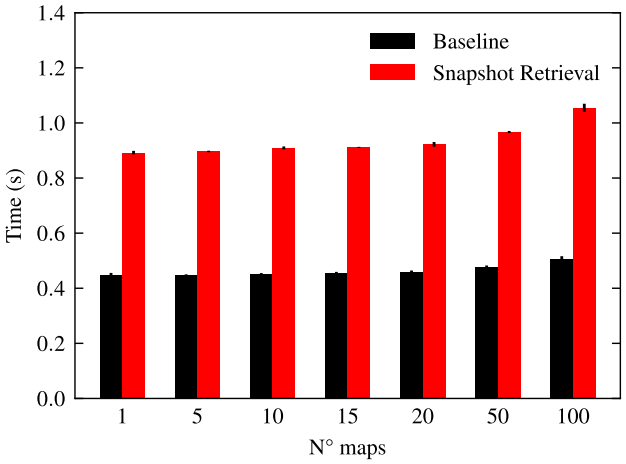
The client uses the MoonGen[14] high-performance packet generator to continuously generate traffic with all the available CPU cores at line rate, so that we can evaluate the monitoring configuration performance in terms of number of network packets handled by the server. The network traffic is composed by simple TCP SYN packets of total size equal to 60 B belonging to 25000 different sessions (the maximum number allowed by the trial version of nProbe), so that we could extract the same transport-layer features of nProbe. Finally, we configured all the incoming interrupt requests coming from the network card of the server to be dispatched to a single CPU core, in order to avoid the influence of possible multicore synchronization issues in our results that are orthogonal to the proposed monitoring technique.

A. SNAPSHOT METRIC RETRIEVAL

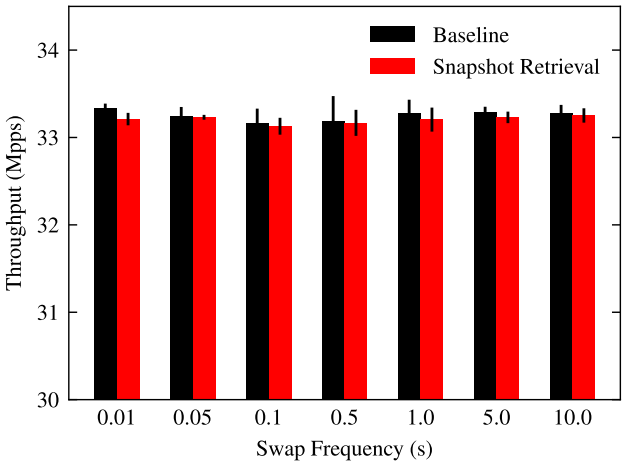
This test measures the impact of the snapshot metric retrieval described in Section IV-B in terms of compilation overhead and throughput with respect to a baseline version with this feature disabled. In particular, we measured the overhead introduced by the creation and compilation of the additional monitoring program while defining a variable number of eBPF maps that requires further controls (differently than the number of entries in each map that do not affect such process). Moreover, we simulate a metric export request at different frequencies to measure whether the swapping process involving eBPF maps affects the monitoring.

Fig. 4a depicts results concerning the compilation time of the eBPF program. Despite the number of maps used, the snapshot retrieval attribute requires slightly more than 2x the time with respect to the baseline version, as the system compiles two eBPF programs instead of one. However, the

[14]https://github.com/emmericp/MoonGen



(a) Program(s) compilation time with different eBPF maps.



(b) Throughput with different metric retrieval frequencies.

FIGURE 4. **Throughput and compilation time overhead of the snapshot metric retrieval vs baseline.**

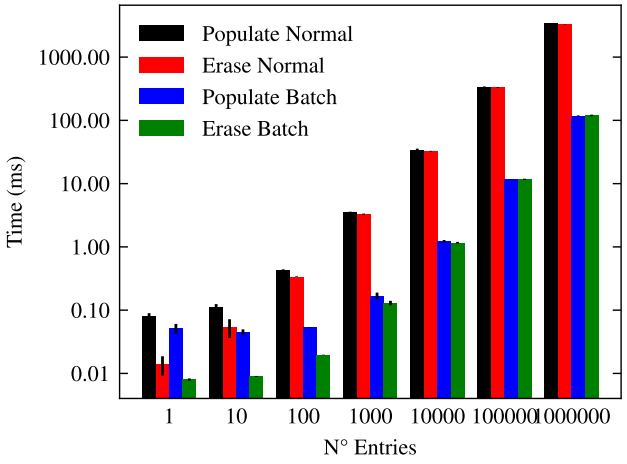


FIGURE 5. **Execution time of populate-and-erase operations on a eBPF array map with different entries (log scale on y-axis).**

overhead of all the additional checks and controls performed during the compilation phase is negligible with respect to the total compilation time, but it slowly increases with the number of maps declared. In fact, the ratio between the compilation time of the two versions increases from 2 when declaring only one map, to 2.1 when defining 100 maps.

Despite swapping the two programs is a simple operation to understand and explain, in a high-performance environment, such as in our testbed, the additional overhead could potentially worsen the performance of the entire monitoring pipeline. Under the hood, the swap operation consists in acquiring a lock to access a specific eBPF map and change the index of the program to be executed.

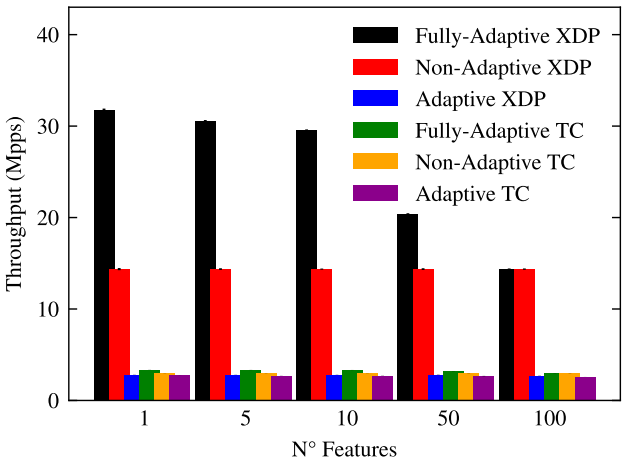
Fig. 4b shows the measured throughput while exporting the defined metric at a different frequency (x-axis). The baseline version reports a stable value throughout the various extraction frequencies (on average equal to 33.25 Mpps). Interestingly, activating this functionality means having slight fluctuations on the maximum number of packets handled by the device, with a peak at 33.25 Mpps and a minimum at 33.13 Mpps. On average, the measured throughput is 33.20 Mpps, resulting in 0.15% degraded monitoring performance with respect to the baseline scenario.

B. METRIC RE-INITIALISATION

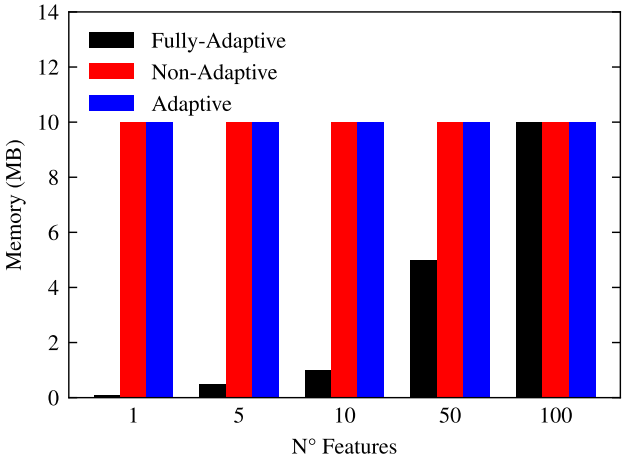
This test compares the execution time of most significant eBPF operations carried out by the control plane on maps. In particular, we focused on the metric re-initialisation, a monitoring option supported by the controller, while performing both iterative and batch accesses to the map. For this purpose, we defined an eBPF array map with an increasing number of entries, and we measured the total time required for both populating the map with fictitious values, and for emptying its content (zeroing all the entries) using the two different methods. We achieved similar results while repeating the test using eBPF hash maps, which support also batch operations, and eBPF FIFO/LIFO maps, while iteratively erasing their content (multiple consecutive push/pop operations), as they do not offer support for batch operations.

Fig. 5 reports the results of the test, using a logarithmic scale on the y-axis. Batch operations are always more than 1x order of magnitude faster than their iterative versions, even when using small eBPF maps with a few entries, such as 1 or 10 in our case. However, this advantage becomes even more relevant when exponentially increasing the number of entries, reaching a peak of more than 3 seconds of difference (∼27x times faster) when using 1000000 entries.

Unfortunately, despite batch operations are widely supported by almost all eBPF maps, some edge cases such as the FIFO/LIFO maps cannot benefit from those optimisations, hence leading to a significantly slower emptying phase. On the other hand, the controller can efficiently handle such complex tasks for all the other types of map, wasting less time performing system calls to access their content and leaving the data plane gathering information effectively. When



(a) Maximum throughput reached.



(b) Memory occupation.

FIGURE 6. Throughput and memory occupation comparison while increasing the set of extracted features using Non-Adaptive vs Adaptive vs Fully-Adaptive methodologies, both in XDP and TC.

combining the metric re-initialisation attribute with the snapshot retrieval, the controller spends such time performing the zeroing operation on the unloaded program, without affecting the performance of the active monitoring process. However, even in this case the advantage of batch operations is notable, as they prevent blocking the controller for a long period operating on the unloaded program, while it should be ready to execute important and potentially frequent tasks on the active one, such the program swap.

C. FULLY-ADAPTIVE MONITORING EVALUATION

This section covers the comparison with other two state-of-the-art monitoring methodologies we developed within eBPF, to have a fair comparison of these approaches within the same underlying technology. We defined the implementation that does not allow modifying at all the monitoring logic as *Non-Adaptive*, the fallback solution described in Section I based on a eBPF support map to keep track of active features as *Adaptive*, and our approach as *Fully-Adaptive*.

We decided a potential sequence of maximum 100 fictitious features represented by counters (for instance, OpenFlow v1.3[15] tracks 40 parameters), corresponding to exactly 100 eBPF instructions, which need to be continuously updated when analysing new packets, and we measured the throughput while increasing and reducing this subset.

Fig. 6a reports the measured throughput using the three techniques, injected either in XDP or TC. As a matter of fact, executing these programs in TC drops the performance, flattening the measured throughput around 3 Mpps for almost all the programs. Moreover, the additional controls (which almost double the number of instructions) to check whether a feature is enabled or not lead the adaptive solution to a significant low throughput even when using XDP, which makes this solution barely usable in production high-speed environments. The advantage of enlarging/restricting the set of features within this approach is minimal, dropping the throughput from 2.82 Mpps and 2.73 Mpps, to 2.71 Mpps and 2.61 Mpps in XDP and TC when passing from 1 to 100 features respectively. Furthermore, the non-adaptive program presents a constant behaviour both in XDP (14.3 Mpps) and TC (3 Mpps), since it always requires extracting all the features.

On the other hand, Fig. 6a clearly confirms the benefits of a fully-adaptive approach, thanks to its capability to extract only the requested features. In fact, while its performance in TC slightly drops from 3.35 Mpps to 2.99 Mpps when extracting all the features, such program injected in XDP outperforms all the others, reaching a peak of 31.8 Mpps and a low of 14.3 Mpps when passing from 1 to 100 features.

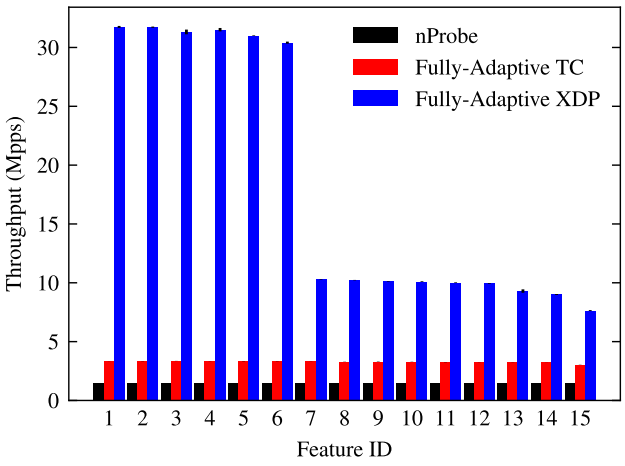
Other than reaching a higher throughput, the fully-adaptive version brings advantages in terms of memory consumption, as depicted in Fig. 6b. Both the non-adaptive and adaptive programs require a constant and maximum (10 MB) amount of memory, independently by the number of features used. In fact, even though a feature is deactivated, the adaptive approach consumes the same amount of bytes to store an empty value. On the other hand, the fully-adaptive approach turns out to be more flexible, adjusting such requirement from 0.1 MB up to 10 MB when gathering all the features.

As a result, the fully-adaptive approach makes the best use of resources, allowing saving both space in memory to store the extracted values, and computational power to handle more network packets when analysing a smaller set of features.

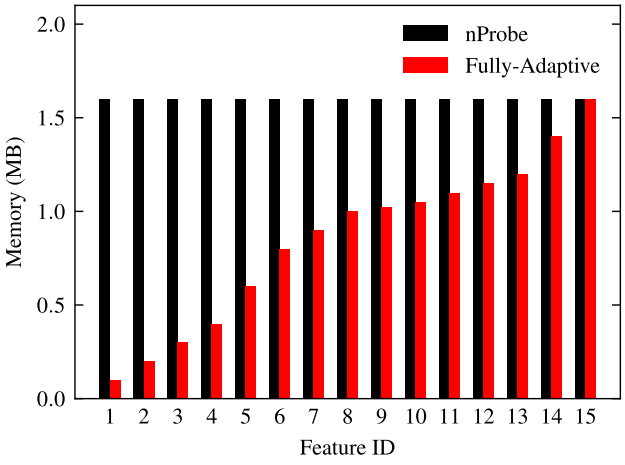
D. COMPARISON WITH nProbe

The second comparison with state-of-the-art approaches involves nProbe, a NetFlow compatible implementation, and our fully-adaptive version developed accordingly to extract the exact same features, with the difference that in our approach it is possible to incrementally widen the set of

[15]https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf



(a) Maximum throughput reached.



(b) Memory occupation.

FIGURE 7. Throughput and memory consumption comparison while increasing the set of extracted features using nProbe vs Fully-Adaptive, both in XDP and TC.

monitored features. As a reference for the set of features to be extracted from the traffic for each flow, we took and numbered with IDs the following ones from the NetFlow v.9 specification[16]:

1) INPUT_SNMP [17]: input SNMP interface number;
2) OUTPUT_SNMP[17]: output SNMP interface number;
3) SRC_AS[17]: source autonomous system number.
4) DST_AS[17]: destination autonomous system number.
5) IN_PKTS: number of incoming packets;
6) IN_BYTES: number of incoming bytes;
7) IPV4_SRC_ADDR: source IPv4 address;
8) IPV4_DST_ADDR: destination IPv4 address;
9) PROTOCOL: IPv4 carried protocol;
10) SRC_TOS: type of service;
11) L4_SRC_PORT: L4 source port;
12) L4_DST_PORT: L4 destination port;
13) TCP_FLAGS: logical OR of TCP flags from packets;
14) FIRST_SWITCHED: timestamp of the first packet;
15) LAST_SWITCHED: timestamp of the last packet.

Since nProbe uses different technologies to analyse packets, we decided to try two different setups while injecting programs. First, we attached an additional eBPF program called *Dropper* in TC mode while testing nProbe to drop packets once reached the networking stack, to ensure that the system does not waste additional computational power. Then, we tested our fully-adaptive program in TC chained with the Dropper, to have a mere and fair comparison with nProbe while dropping packets at the same level. Finally, to leverage the benefits of early packet processing enabled by XDP, we tested our program coupled with the *Dropper* directly in XDP, to speed up the monitoring lookup-and-drop pipeline. Fig. 7a and Fig. 7b report the achieved results using as x-axis the feature ID that corresponds to extracting all those features with ID lower or equal to the current one under analysis. For instance, the column referring to the feature ID 7 reports the result obtained while extracting all the features previously listed with IDs between 1 and 7.

As depicted in Fig. 7a, our fully-adaptive approach always performs better than nProbe, independently of the operational mode. In fact, while our program injected in TC achieves 3.34 Mpps and 3.03 Mpps when extracting one and all features respectively, nProbe presents a constant and lower throughput (1.44 Mpps). Moreover, the adaptive program running in XDP significantly outperforms the other ones, leading to a maximum of 31.7 Mpps when extracting only one feature, and 7.63 Mpps when monitoring all the features. Clearly, the IPV4_SRC_ADDR feature requires the system to inspect network packets more in depth by parsing both the Ethernet and IPv4 layers, significantly affecting the performance of the monitoring pipeline, which has to continuously perform controls and conversions on every packet. The next heavy features are L4_SRC_PORT, which require inspecting also the transport layer (TCP in our case), and LAST_SWITCHED, which forces the program to compute the timestamp of every packet (potentially the last one of a TCP connection).

Another advantage of our fully-adaptive methodology, as depicted in Fig. 7b, consists in the reduced memory occupancy with respect to nProbe that, on the other hand, always reserves the maximum amount of space, as it extracts all the features. In fact, in a hypothetical situation where only a subset of such features are needed, the fully-adaptive methodology allows saving up to 16x times the space (when extracting only 1 feature in our experiment), which becomes significant in a real scenario with more monitored sessions and more features extracted.

VI. FUTURE DIRECTIONS

We forced the entire experimental validation of our approach on a single CPU-core to avoid problems due to concurrent

[16]https://www.cisco.com/en/US/technologies/tk648/tk362/technologies_white_paper09186a00800a3db9.html

[17]one-time feature computed the first time the monitor records a packet, and in our case always set to zero.

accesses to shared resources, to have a concrete measurement of the mere methodology, limiting side effects. We leave for further developments the validation on a multicore environment, leveraging specific data structures called PERCPU eBPF maps. Those maps allocate an independent memory area for each CPU core, enabling faster and (kernel-level) concurrency-free access. We expect that, combined with Receive Side Scaling (RSS),[18] a feature to ensure that the system handles packets belonging to a specific flow using the same CPU core, these maps further empower adaptive network monitoring reaching higher throughput. On the other hand, the entire metric extraction phase would require more time due to the aggregation of each array of values (one for each CPU core), in order to return the final value to the user.

Finally, focusing on specific use cases and adopting the fully-adaptive network monitoring approach, it would be interesting to analyse the design and trade-offs of making monitoring decisions locally to a specific probe (i.e., network device) or globally to the set of active probe running within an infrastructure, leveraging either distributed individual monitoring entities or a centralised model.

VII. CONCLUSION

This paper proposes the design of a control plane to support and enable a fully-adaptive network monitoring approach. The underlying user-defined monitoring pipeline builds upon *extended Berkeley Packet Filter (eBPF)*, a notable programmable data plane language and in-kernel virtual machine that allows to safely modify at runtime the monitoring logic, while the proposed controller automatically couples the defined monitoring program, providing access and control methodologies to interact with it and exporting the defined metrics.

At first, we discussed the proposed control plane, resulting in a highly programmable solution, where not only the underlying monitoring program can be dynamically changed, but also many higher-level properties can be activated at runtime, offering an efficient control of the entire monitoring pipeline and supporting many types of monitoring granularities (e.g., flow-level, packet-level, time-window, and more). Moreover, the fully-adaptive approach allows introducing and extracting new custom (and more complex) features that were not previously defined, resulting in a continuous integration of both older and newer requirements, without causing disruptions to the monitoring process.

We measured, in terms of performance degradation, the adoption of the monitoring options introduced in the controller, proving that they do not significantly affect the underlying monitoring pipeline (0.15% degraded throughput when frequently retrieving monitoring metrics) and leverage the most efficient extraction primitives (20x faster execution time using batch operations). Finally, we compared the *fully-adaptive* methodology with nProbe, a NetFlow-compatible implementation, and with an *adaptive* and a *non-adaptive* implementation in eBPF. The *fully-adaptive* approach has less impact on the system, allowing saving both more than 10x the computational power to process more incoming packets in XDP (∼2x when deployed in TC) and ∼10x memory space, with respect to both nProbe and a full-features analysis scenario of the other approaches in eBPF.

[18]https://developers.redhat.com/blog/2021/05/13/receive-side-scaling-rss-with-ebpf-and-cpumap

REFERENCES

[1] C. DeCusatis, P. Liengtiraphan, A. Sager, and M. Pinelli, "Implementing zero trust cloud networks with transport access control and first packet authentication," in *Proc. IEEE Int. Conf. Smart Cloud (SmartCloud)*, Nov. 2016, pp. 5–10, doi: 10.1109/SmartCloud.2016.22.

[2] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of eBPF for non-intrusive performance monitoring," in *Proc. IEEE/IFIP Netw. Operations Manage. Symp. (NOMS)*, Apr. 2020, pp. 1–7, doi: 10.1109/NOMS47738.2020.9110434.

[3] J. Vila-Carbo, J. Tur-Masanet, and E. Hernandez-Orallo, "An evaluation of switched Ethernet and Linux traffic control for real-time transmission," in *Proc. IEEE Int. Conf. Emerg. Technol. Factory Autom.*, Sep. 2008, pp. 400–407, doi: 10.1109/ETFA.2008.4638424.

[4] M. Abranches, O. Michel, E. Keller, and S. Schmid, "Efficient network monitoring applications in the kernel with eBPF and XDP," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2021, pp. 28–34, doi: 10.1109/NFV-SDN53031.2021.9665095.

[5] L. Deri, "nProbe: An open source NetFlow probe for gigabit networks," in *Proc. TERENA Netw. Conf. (TNC)*, 2003, pp. 1–4.

[6] Y. Liu, J. Sun, R. Sun, and Y. Wen, "Next generation internet traffic monitoring system based on NetFlow," in *Proc. Int. Conf. Intell. Syst. Design Eng. Appl.*, Oct. 2010, pp. 1006–1009, doi: 10.1109/ISDEA.2010.337.

[7] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance implications of packet filtering with Linux eBPF," in *Proc. 30th Int. Teletraffic Congr. (ITC)*, Sep. 2018, pp. 209–217, doi: 10.1109/ITC30.2018.00039.

[8] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with eBPF: Experience and lessons learned," in *Proc. IEEE 19th Int. Conf. High Perform. Switching Routing (HPSR)*, Jun. 2018, pp. 1–8, doi: 10.1109/HPSR.2018.8850758.

[9] T. Nam and J. Kim, "Open-source IO visor eBPF-based packet tracing on multiple network interfaces of Linux boxes," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2017, pp. 324–326, doi: 10.1109/ICTC.2017.8190996.

[10] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A framework for eBPF-based network functions in an era of microservices," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 1, pp. 133–151, Mar. 2021, doi: 10.1109/TNSM.2021.3055676.

[11] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu, and J. Pi, "Securing Linux with a faster and scalable iptables," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 3, pp. 2–17, Nov. 2019, doi: 10.1145/3371927.3371929.

[12] S. Rivera, V. K. Gurbani, S. Lagraa, A. K. Iannillo, and R. State, "Leveraging eBPF to preserve user privacy for DNS, DoT, and DoH queries," in *Proc. 15th Int. Conf. Availability, Rel. Secur.*, Aug. 2020, pp. 1–10, doi: 10.1145/3407023.3407041.

[13] X. Dong and Z. Liu, "Multi-dimensional detection of Linux network congestion based on eBPF," in *Proc. 14th Int. Conf. Measuring Technol. Mechatronics Autom. (ICMTMA)*, Jan. 2022, pp. 925–930, doi: 10.1109/ICMTMA54903.2022.00188.

[14] M. Jadin, Q. De Coninck, L. Navarre, M. Schapira, and O. Bonaventure, "Leveraging eBPF to make TCP path-aware," *IEEE Trans. Netw. Service Manage.*, early access, May 10, 2022, doi: 10.1109/TNSM.2022.3174138.

[15] M. Xhonneux and O. Bonaventure, "Flexible failure detection and fast reroute using eBPF and SRv6," in *Proc. 14th Int. Conf. Netw. Service Manage. (CNSM)*, 2018, pp. 408–413.

[16] J. Hong, S. Jeong, J. H. Yoo, and J. W. K. Hong, "Design and implementation of eBPF-based virtual TAP for inter-VM traffic monitoring," in *Proc. 14th Int. Conf. Netw. Service Manage. (CNSM)*, 2018, pp. 402–407.

[17] F. Parola, F. Risso, and S. Miano, "Providing telco-oriented network services with eBPF: The case for a 5G mobile gateway," in *Proc. IEEE 7th Int. Conf. Netw. Softwarization (NetSoft)*, Jun. 2021, pp. 221–225, doi: 10.1109/NetSoft51509.2021.9492571.

[18] J. Yang, L. Chen, and J. Bai, "Redis automatic performance tuning based on eBPF," in *Proc. 14th Int. Conf. Measuring Technol. Mechatronics Autom. (ICMTMA)*, Jan. 2022, pp. 671–676, doi: 10.1109/ICMTMA54903.2022.00139.

[19] S.-Y. Wang and J.-C. Chang, "Design and implementation of an intrusion detection system by using extended BPF in the Linux kernel," *J. Netw. Comput. Appl.*, vol. 198, Feb. 2022, Art. no. 103283, doi: 10.1016/j.jnca.2021.103283.

[20] I. Ben-Yair, P. Rogovoy, and N. Zaidenberg, "AI & eBPF based performance anomaly detection system," in *Proc. 12th ACM Int. Conf. Syst. Storage*, May 2019, p. 180, doi: 10.1145/3319647.3325842.

[21] S. Miano, R. Doriguzzi-Corin, F. Risso, D. Siracusa, and R. Sommese, "Introducing SmartNICs in server-based data plane processing: The DDoS mitigation use case," *IEEE Access*, vol. 7, pp. 107161–107170, 2019, doi: 10.1109/ACCESS.2019.2933491.

[22] N. Kostopoulos, D. Kalogeras, and V. Maglaris, "Leveraging on the XDP framework for the efficient mitigation of water torture attacks within authoritative DNS servers," in *Proc. 6th IEEE Conf. Netw. Softwarization (NetSoft)*, Jun. 2020, pp. 287–291, doi: 10.1109/NetSoft48620.2020.9165454.

[23] T. Weng, W. Yang, G. Yu, P. Chen, J. Cui, and C. Zhang, "Kmon: An in-kernel transparent monitoring system for microservice systems with eBPF," in *Proc. IEEE/ACM Int. Workshop Cloud Intell. (CloudIntelligence)*, May 2021, pp. 25–30, doi: 10.1109/CloudIntelligence52565.2021.00014.

[24] J. Levin and T. A. Benson, "ViperProbe: Rethinking microservice observability with eBPF," in *Proc. IEEE 9th Int. Conf. Cloud Netw. (CloudNet)*, Nov. 2020, pp. 1–8, doi: 10.1109/CloudNet51028.2020.9335808.

[25] C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, "A protocol-independent container network observability analysis system based on eBPF," in *Proc. IEEE 26th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2020, pp. 697–702, doi: 10.1109/ICPADS51040.2020.00099.

[26] R. Hofstede, P. Celeda, B. Trammell, I. Idilio, R. Sadre, A. Sperotto, and A. Pras, "Flow monitoring explained: From packet capture to data analysis with NetFlow and IPFIX," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 4, pp. 2037–2064, 4th Quart., 2014, doi: 10.1109/COMST.2014.2321898.

[27] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A low cost network monitoring framework for software defined networks," in *Proc. IEEE Netw. Oper. Manage. Symp. (NOMS)*, May 2014, pp. 1–9, doi: 10.1109/NOMS.2014.6838227.

[28] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," in *Proc. IEEE Netw. Oper. Manage. Symp. (NOMS)*, May 2014, pp. 1–8, doi: 10.1109/NOMS.2014.6838228.

[29] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: Dynamic resource allocation for software-defined measurement," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 419–430, doi: 10.1145/2619239.2626291.

[30] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "SCREAM: Sketch resource allocation for software-defined measurement," in *Proc. 11th ACM Conf. Emerg. Netw. Experiments Technol.*, Dec. 2015, pp. 1–13, doi: 10.1145/2716281.2836099.

[31] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 357–371, doi: 10.1145/3230543.3230555.

[32] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "PINT: Probabilistic in-band network telemetry," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 662–680, doi: 10.1145/3387514.3405894.

[33] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 561–575, doi: 10.1145/3230543.3230544.

[34] S. R. Chowdhury, R. Boutaba, and J. François, "LINT: Accuracy-adaptive and lightweight in-band network telemetry," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage. (IM)*, May 2021, pp. 349–357.

[35] Q. Zheng, S. Tang, B. Chen, and Z. Zhu, "Highly-efficient and adaptive network monitoring: When INT meets segment routing," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 3, pp. 2587–2597, Sep. 2021, doi: 10.1109/TNSM.2021.3069000.

[36] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 129–143, doi: 10.1145/2934872.2934879.

SIMONE MAGNANI (Member, IEEE) was born in Cesena, Italy, in 1996. He received the B.S. degree in computer science and engineering from the Università di Bologna, Italy, in 2018, and the M.S. degree in computer engineering from the Politecnico di Torino, Italy, in 2020. He is currently pursuing the Ph.D. degree in cybersecurity and reliable artificial intelligence jointly with the Università di Genova, Italy, and Fondazione Bruno Kessler, Italy. While working on his M.S. final thesis, in 2020, he was a Research Assistant of Prof. Fulvio Risso, while he carried on Toshi, a European project funded by EIT-Digital, and contributed to open source projects such as Polycube, CrownLabs, and BCC. His research interests include adaptive network traffic monitoring, cybersecurity, and flexible artificial intelligence methodologies for cyberattack detection and mitigation.

FULVIO RISSO (Member, IEEE) was born in Saluzzo, Italy, in 1971. He received the B.S., M.S., and Ph.D. degrees in computer engineering from the Politecnico di Torino, Italy, in 2000. He is currently an Associate Professor with the Politecnico di Torino, and responsible for the Network and Multimedia Laboratory, Department of Control and Computer Engineering. He started many open-source projects, including WinPcap (the de-facto standard library for network analysis tools under the Windows platform), NetBee, Liqo, CrownLabs, and Polycube, which represent a breakthrough in their respective fields. He has coauthored more than 100 scientific publications. His research interests include high-speed and flexible network processing, edge/fog computing, software-defined networks, and network functions virtualization.

DOMENICO SIRACUSA received the B.S., M.S., and Ph.D. degrees in telecommunication engineering from the Politecnico di Milano, Italy, in 2012. He is currently the Head of the Robust and Secure Distributed Computing (RiSING) Research Unit, Fondazione Bruno Kessler, Italy. He acted as a Co-ordinator of the H2020 EU-Korea DECENTER Project, the H2020 EU ACINO Project, and the EIT Digital Digiflow Project. He coauthored more than 100 scientific publications appeared in international peer-reviewed journals and in major conferences on networking and cloud technologies. His current research interests include orchestration of next generation internet infrastructures, cloud and fog computing, SDN/NFV and virtualization, security, and robustness.