Efficient Dynamic Flow Tracking for Packet Analyzers

Paul Emmerich, Maximilian Pudelko, Quirin Scheitle, Georg Carle
Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich
{emmericp|pudelko|scheitle|carle}@in.tum.de

***Abstract*—Analyzing large amounts of traffic at the packet or flow level is an important part of managing and monitoring cloud network infrastructure. Common scenarios that require low-level packet analysis are troubleshooting problems, accounting traffic, and security applications such as intrusion detection systems or firewalls. Moreover, researchers often analyze traffic for scientific purposes. For such low-level traffic analyses, tracking flows is a feature required for both commercial and scientific purposes. However, there is no good shared library available to implement this functionality in an efficient, configurable, and dynamic way that is suitable for real-time analysis. We implement a high-performant generic flow tracker that can track millions of simultaenous flows based on arbitrarily complex definitions of a flow. We make this implementation available as open source in our traffic analysis tool FlowScope. The highly efficient real-time tracking of flows by arbitrarily complex user-defined flow criteria and filters is enabled by just-in-time (JIT) compilation of flow tracking rules. The code and evaluation scripts are available as free and open source at:**

https://github.com/emmericp/FlowScope

***Keywords*—DPDK, traffic analysis, flows**

I. Introduction

Traffic analysis in real time is required in cloud network infrastructure to troubleshoot problems and for security applications like intrusion detection systems (IDS) and firewalls. Beside these commercial applications, researchers also look at packet data on a flow level. All of these applications require efficient tracking of flows by various configurable parameters. This core functionality is currently re-implemented on a case-by-case basis for each new tool.

We provide an extensible framework as a building block for applications that consume raw packets and need to track state on a per-flow basis. Examples for such applications are tools calculating statistics for monitoring, accounting, or DoS detection, as well as tools for security analysis. Our flow tracker is integrated in our FlowScope [1] tool which already provides other functionality for these scenarios. For example, our QQ data structure can be used for time traveling network debugging and security analysis together with the new flow tracker. All crucial parts of the flow tracker are under full control of the application built on top of it: extracting the relevant header fields to identify flows is done in JIT-compiled user-defined code. The user's application is informed via callbacks of all packets and events such as (configurable) flow timeouts. High performance is achieved by building on DPDK [2] for fast access to hardware and on libmoon [3], [4] to provide a fast and flexible scripting interface and protocol stack implementations [5].

Our main contributions are the first fast and flexible open source implementation of flow tracking in software on commodity hardware. Compared to other implementations, our implementation handles arbitrary flow types and identifiers (from simple 5 tuple to complex protocol-level tracking) while achieving packet rates of above 10 Gbit/s with minimum sized packets with millions of flows. We provide a performance analysis with both real-world traces and a worst-case analysis with small packets and 10 million flows that are tracked bidirectionally. Moreover, we provide detailed example programs for the presented use cases in our git repository [6]. We publish all of our code used for the experiments to make this paper fully reproducible.

The remainder of the paper is structured as follows. We start with a dive into the background of flow-level analysis and tracking by looking at related work in the next section. We then describe the architecture and implementation of our framework in Section III followed by an explanation of one of our example analyzer module in Section IV. Section V looks at the performance of our implementation. We conclude in Section VI by showing how you can reproduce our research by deploying and running FlowScope with our new flow tracker.

II. Related Work

This work builds on our tool FlowScope [1], [6], a scriptable network traffic recorder and analyzer. FlowScope builds on the metaphor of a digital storage oscilloscope: it continuously records traffic in an in-memory ring buffer and dumps specific traffic to disk flows to disk on user-defined trigger events. This effectively allows time travel to find root causes of anomalous flows. It is designed for both debugging networks and security applications in mind. However, despite the name, it featured no flow tracking or any flow-specific functionality in the version published in 2017. The previous paper focused on the underlying data structure for efficient packet storage.

Flow tracking for monitoring is often done via the IPFIX [7] flow exporter protocol, a deployment consists of two parts: the exporter extracting flow information from raw traffic and the collector receiving only the summarized flows. The exporter is typically implemented either directly on routers or

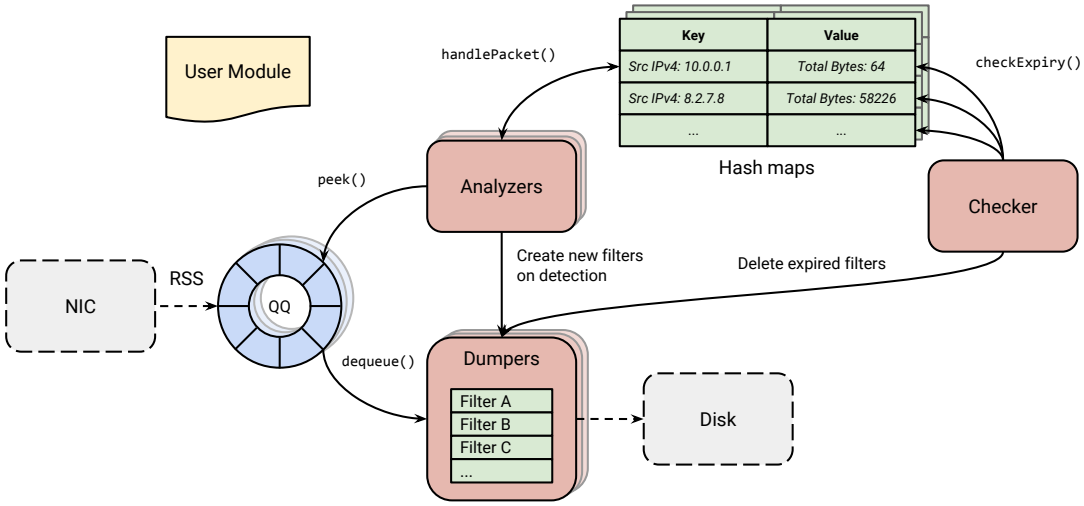

Figure 1. New FlowScope architecture

in specialized appliances that are either based on hardware or software [8]. However, modern cloud networks aim to run on softwarized network deployments on commodity hardware. Several open source IPFIX implementations exist, most notably the C/C++ library `libfc` [9] (last updated in 2014). None of the open source software implementations is tuned for performance and real-world deployments therefore built on purpose-built appliances. Building an IPFIX exporter is one of the usage scenarios for our framework. We include an example (`liveStatistician.lua`) of a simple per-flow statistics tool with similar semantics to IPFIX to showcase how to build such a tool in a high-speed version with our platform.

Examples of analyzers featuring flow tracking are DPDK-Stat [10] and FlowMon-DPDK [11]. Both are limited to hard-coded flow identifications and cannot track more complex protocols. FlowMon-DPDK's design also prohibits bidirectional tracking of flows, limiting the analyses that can be performed. Further, they are limited to a low number of flows, no evaluation is given for more than 200k parallel flows. The limiting factor for both flexibility and performance is the custom fixed-size hash table implementation relying on partially offloading hash calculation to the NIC (which has no concept of bidirectional flows or high-level protocols). The aim of our implementation is to track several million concurrent flows with arbitrary flow identifiers. Moreover, we also provide support for dumping selected flows to disk.

Security applications such as Bro [12] also implement exhaustive flow tracking, however these have a narrow focus and are typically built to only track and match on header fields. Bro also leverages tracking of TCP sequence numbers to assert whether flows have been observed in full, or whether packets are missing. We extend this approach by providing generic, scriptable flow tracking, permitting arbitrary rules to match and track flows.

Other efforts in this area primarily focus on offline analysis of data streams, for example improved heavy hitter analysis [13]. These analyses are not fast enough for real-time analysis in bigger networks. These research efforts are orthogonal to our framework. Another seemingly similar, but very different scenario is load balancing flows. This is typically implemented by hashing over a defined set of protocol fields and distributing flows by hash. These techniques can be found all over the protocol stack, from high-level load balancers to low-level bonding of several network links via LACP [14] or ECMP [15]. However, they cannot keep individual state per flow, they merely use the hash to distribute traffic.

III. Architecture

Besides the original QQ data structure [6], most parts of FlowScope required major changes to achieve arbitrary flow tracking. Figure 1 shows the updated architecture of FlowScope as a high-level overview. Packets are received at one or multiple NICs and are enqueued in one or multiple QQ ring buffers. All packets are then analyzed in a separate *analyzer thread*: the first step is to look up or create the corresponding flow state in a hash map via a user-defined flow key extraction function. The packet is then passed to a user-defined function together with its flow state. This user-defined function can decide whether the flow shall be dumped to hard disk for persistent storage by informing the *dumper threads* with filter functions. Concurrently, a *checker thread* performs garbage collection on inactive flows based on a configurable timeout. All customizable functions such as flow state definition, the flow key extraction, and the analyzer code are loaded from a user module.

Each of these functions runs in a separate thread, and each step can be handled by multiple threads to allow for multi-core scaling at all steps. We support multi-threaded reading from both multiple and single input NICs by supporting the RSS (receive side scaling) hardware feature.

A. QQ (Queue-in-Queue) ring buffer

FlowScope can use the QQ ring buffer to keep packets in memory instead of discarding them after the analyzer step. QQ is a high-speed circular buffer data structure optimized for large memory sizes ($\geq$ 100 GB) at large packet and data rates (tested at $\geq$ 120 Gbit/s). This data structure was the focus of our previous publication [1] and is therefore only discussed briefly here.

QQ consists of two levels of nested queues; the outer queue holding references to the inner ones, and the inner queues actually storing the data. This layered design improves multi-threaded performance by reducing the contention for the mutex guarding access. Instead of acquiring the lock on every insert/dequeue, threads get a complete inner queue exclusively for a short time. This model works nicely with the common receive-side scaling (RSS) optimization in NICs, because QQ preserves intra-flow packet ordering. The inner queues store variably sized objects (like packets) tightly packed without losing useful properties like random access. Further, QQ allows access to elements at the queue head without dequeuing them, permitting analyzers to see new packets as soon as possible. Contrary to other buffers, QQ aims to be as filled as possible. It does not allow dequeuing packets until the fill level reaches the high water mark, so that as many packets as possible are available in case of detected anomalies.

Using QQ is optional in FlowScope. Enabling it allows using the time traveling dumper feature. It can also improve performance as the work done for packet reception in the DPDK driver is shifted to a separate thread. Minimizing work done in the thread performing the actual reception by using QQ can also help with packet drops compared to the usual run-to-completion model of DPDK [11].

B. Hash map

The hash map is a central part of FlowScope: it contains an entry for each flow consisting of the flow identifier key and a user-defined flow state. It needs to be accessed at least once per packet and usually a second time to update the flow state making a potential performance bottleneck. Multiple hash maps are supported to track different flow types simultaneously (e.g., IPv4 and IPv6 flows).

Our requirements for the hash map are not only high performance with a large number of entries, it also needs to handle concurrent access from multiple threads. Packet analyzers often rely on hardware flow splitting features like RSS to ensure that packets belonging to one flow are always processed by the same core to avoid expensive and complicated handling of multi-thread safety (e.g., [11]). FlowScope features a completely user-defined flow identifier key that might not be handled by the RSS hardware flow splitter. A simple example is tracking both directions of a flow on a mirror port, a splitting function typically not supported by hardware. In this case, full multi-threading safety is required.

We evaluate several well-known hash tables for use in FlowScope: Google's sparsehash [16], Facebook's folly [17], Jeff Preshing's Junction concurrent maps [18], and `concurrent_hash_map` from Intel's Thread Building Blocks (TBB) library [19]. Table I lists our requirements (see [18] for a detailed performance analysis) and whether the hash maps can fulfill them. The Junction maps are the fastest, however, they gain that speed at the cost of only supporting 64 bit keys. As we aim to support arbitrary, user-defined flow-identifiers, arbitrary sized keys are a requirement for us.

Table I
HASH MAP COMPARISON

Implementation	Speed	Lock-free	Arbitrary key size
sparsehash [16]	+	✗	✓
folly [17]	-	✓	✓
Junction [18]	++	✓	✗
TBB [19]	+	✓	✓

We select Intel's TBB library for the hash table based on these results. TBB, like most C++ data structures, requires the key size as a template parameter, i.e., as a compile-time constant. User-defined keys in FlowScope are completely dynamic and determined at run time. We solve this by pre-instantiating the C++ template with all relevant sizes and we chose the correct implementation at run time. This only affects the size of our executable, there is no run-time overhead of the unused implementations in memory. Only the classes specialized to the sizes actually being required are used.

C. Analyzer threads

Analyzers dequeue packets either directly from a NIC or through an intermediary QQ ring buffer (QQ) as soon as they arrive. The user module defines an `extractFlowKey()` function that is called for each packet to classify it into one of the flow tables by extracting its identifying flow feature (e.g., 5-tuple or VXLAN VNI). This process does not yet involve any state or expensive hash map access: Basic pre-filtering can be performed very cheaply here, e.g., by discarding IPv4 traffic in an IPv6-only measurement. The function therefore returns three values: if a packet is interesting at all, to which flow table (i.e., hash map) it belongs, and the flow identification key to use for the lookup.

Using the flow identifier key, the flow state is looked up in the corresponding hash map. This locks the entry for exclusive read-write access until the user module function `handlePacket()` returns. `handlePacket()` can perform arbitrary flow analysis based on the flows previous state and the current packet and updates the flow state.

Should a threshold be passed or an anomaly be identified, the function can request the on-disk archival of this flow for forensic purposes if the QQ module is enabled. In this case, `buildPacketFilter()` in the user module is tasked with transforming a flow into a pcap filter expression (or arbitrary JIT-compiled code) to match packets for this flow in the dumper thread.

D. Checker thread

Since the module's `handlePacket()` function is only called for arriving packets, there would be no way to detect and

handle inactive flows (e.g., to implement an IPFIX exporter's inactivity timeout). Therefore, a checker thread iterates over all flows in the hash maps in regular intervals and passes them to the `checkExpiry()` user module function. Here the user can decide if a flow is still active, e.g., by keeping the timestamp of the last seen packet or protocol specific flags (such as TCP FIN). Should a flow be deemed inactive, it is purged from the hash map and the dumpers are instructed to forget its matching filter rule when reaching the current time stamp in the buffer.

E. Dumper thread

Note that the dumper thread existed for our previous publication, but it was limited to a single simple filter in the past. We significantly extended its capabilities to permit multiple filters and threads.

Dumpers dequeue packets from the QQ ring buffer as late as possible to maximize the amount of information available in case of a detected anomaly. Due to their delayed processing of the packets, rules can not be immediately discarded once a flow is inactive or the capture of interesting flows could end early, leading to missing packets. If the checkers requests deletion of of a flow rule, the timestamp of the last seen packet is also included. With this information dumpers know exactly when it is safe to finally forget about a rule; if the other dequeued packets in QQ are older than the timestamp.

The pflua [20] framework is used to facilitate high performance packet matching with the familiar PCAP filter syntax. pflua works by compiling filters to Lua which is then JIT-compiled by LuaJIT [21] for performance. This approach to packet filtering has proved itself for packet filtering and switching in other implementations [22], [23].

Another new feature of the dumper thread is that it can now also be triggered externally via a JSON REST API. We are using this feature to build a distributed anomaly detection system that can respond to events in remote locations.

F. JIT compilation of user-defined functions

All previously mentioned user-defined functions can be written in the Lua programming language and JIT-compiled with LuaJIT [21]. Using a scripting language here maximizes the flexibility as user modules can be swapped without requiring recompilation of the code. Lua was chosen because of the availability of LuaJIT, a very fast JIT compiler and its fast and simple integration with existing C/C++ code. This makes Lua a good choice even for performance critical functions that need to access every single packet in the user module. LuaJIT has proven itself suitable for performance-critical packet processing tasks in the past [4], [22].

Another advantage of LuaJIT is that it can call into C/C++ functions without marshalling overhead in most cases. This makes it possible to write all user module functions in C/C++ if Lua is not suitable for a given task or if pre-existing libraries are required.

```
function module.handlePacket(key, state, buf, isFirst)
    local t = lm.getTime()
    state.packets_interval = state.packets_interval + 1
    state.bytes_interval = state.bytes_interval + buf.size
    state.packets_total = state.packets_total + 1
    state.bytes_total = state.bytes_total + buf.size
    if isFirstPacket then
        state.first_seen = t
        state.interval_start = t
    end
    state.last_seen = t
end
```

Listing 1: `handlePacket()` in the stats example

IV. Example scripts

All references to line numbers in files in this section refer to commit `c7b22db1371d` in the FlowScope repository [6]. Code excerpts in the paper are slightly modified and shortened to fit the format.

FlowScope comes with example modules meant as a basis for custom user modules and analyzers. The following is a walkthrough of the code in our `examples/liveStatistician.lua` example module to aid understanding how FlowScope processes a packet and interacts with the user module. This example script is a simple flow statistics module with IPFIX-like semantics.

The module starts by importing utility functions (e.g., protocol parsing) from libmoon and flowscope in lines 1-6. Next, the modules defines local variables that are not exposed to the FlowScope driver. This can be configuration or values required for calculations.

A. Statistics: Analyzer

Next come the required module definitions so that FlowScope knows what to initialize and which functions to call. Lines 12 to 22 define a C struct which encapsulates the state of a flow. `module.stateType` (line 25) exposes this type, such that FlowScope can instantiate the hash maps with this as the value type on startup. It is possible to provide a default state for new flows (default is zero-initialized) via the `module.defaultState` (line 26) configuration. Lastly the flow key types are defined. Since FlowScope already comes with a IP 5-tuple key and extraction function, it is reused from the `tuple` lib. This default implementation, found in `lua/tuple.lua`, already handles both IPv4 and IPv6 and tracks flows bidirectionally.

FlowScope uses this information to to look up or create the flow state in the central hash map. It then calls the `handlePacket` function reproduced in Listing 1. The fields available in state are directly mapped to the C struct defined in `module.stateType` previously using a LuaJIT FFI struct. Accesses to these fields are directly compiled to memory accesses to the corresponding hash map entry, there is no overhead or copying from using a scripting language.

B. Statistics: Checker

Next is the checker thread is configured: `module.checkInterval` (line 42) sets how often it should run, we configure it to run every 5 seconds here.

Optionally, `checkInitializer()` (line 44) can perform setup actions for the checker state. It is run once per checker run before any flow is accessed. Here, it sets up counters and prepares the list of the current top flows (line 45-49).

The `checkExpiry()` (line 67 et seq.) function is then called for every flow currently present in the hash maps. In its arguments it holds the flow key, flow state, checker state and to which table the flow belongs. The example module calculates basic traffic metrics in lines 70 to 72: throughput in bytes and packets. Note that the checker updates the flow state by resetting the `_interval` fields.

Finally, it calculates the requested statistics and generates a top list of currently active flows. The timestamp set previously by the analyzer is used to decide whether a flow is still active (line 85 et seq.). At the end of a checker run, the collected data is printed out in the `checkFinalizer()` (line 93 et seq.) function.

C. Further use cases

Beside the statistics script, we also include a more sophisticated tracking scenario as an example: tracking QUIC flows based on the QUIC connection ID instead of the usual 5-tuple. The file `quicDetect.lua` contains all necessary definitions to track QUIC flows for further analysis. Running this on real Internet traffic on a mirror port at an ISP reveals a problem: we encountered a large number of flows from and to VPN service providers that tried to disguise their traffic as QUIC. QUIC is almost completely encrypted and it is therefore not possible to distinguish between real QUIC traffic and other traffic on UDP port 443 in many cases.

More sophisticated analyzers are also feasible with FlowScope. We used a FlowScope user module to track anomalies in the IP TTL field on a mirror port of an ISP. FlowScope allowed us to identify flows where the TTL value has changed, possibly indicating an attack on a user via spoofed traffic. QQ then allowed us to selectively and retroactively dump the affected flows to disk. Capturing all flows to find anomalous TTL behavior later would have been prohibitive due to the large bandwidth of the monitored link.

V. Evaluation

We validate the functionality of FlowScope by running it on a replay of the Abilene traces [24]. This suffices as a validation of the functionality and for a quick demonstration of FlowScope's capabilities. However, synthetic traffic is more suitable for a performance analysis. We use MoonGen [4] to generate worst-case traffic with minimum-sized UDP packets and a configurable number of flows.

We use a server with an Intel Xeon E5-2620 v3 CPU with 6 cores at 2.4 GHz, 128 GB RAM, and an Intel XL710 40 Gbit/s NIC and an Intel X540 10 Gbit/s NIC for the evaluation here.

A. Multi-thread scaling

Figure 2 shows how the processing rate of FlowScope scales with the number of analyzer threads when handling 1 M flows. To minimize influence of user module functions, the benchmark was created with the `noop.lua` module. This module extracts ordered 5-tuples and tracks them in a hash map, but does not perform more specific calculations. This measurements hence assesses the upper bound for the performance of a user module tracking bidirectional 5-tuple flows.

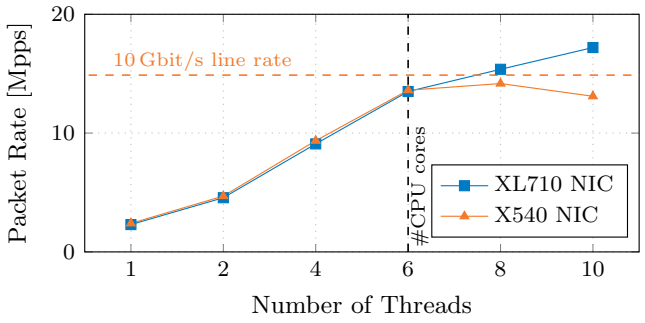

Figure 2. Processing rates for no-op analysis function with 1 million flows with 64 byte packets, 6 core (+ HT) CPU

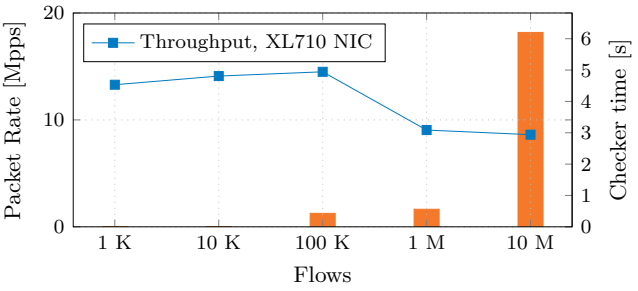

Figure 3. Processing rates for no-op analysis function with 4 threads (logarithmic x axis)

Performance increases linearly as more analyzer threads are added until the number of physical CPU cores is exhausted. Beyond the number of physical CPU cores, only virtual hyper-threading cores are available, leading to slightly diminishing performance gains on the 40 Gbit/s XL710 NIC. The X540 NIC hits the line rate limit and adding more virtual cores hurts performance slightly.

B. Number of flows

Increasing the number of flows extends the performance benchmark beyond the analyzer threads: The hash map, growing in size, slows down as cache misses increase, and the checker thread needs to iterate over more entries.

Figure 3 shows how the performance with four analyzer threads changes as the number of flows is increased exponentially. Performance initially increases as there are fewer memory access conflicts between the threads. When stepping from 100 K flows to 1 M flows, performance drops from 14.1 Mpps to 9.5 Mpps. This drop can be explained by the fact the the hash map does not fit into the CPU's L3 cache (15 MiB) anymore at that flow rate. Each flow entry consumes about 40 byte of memory, which includes flow identifier, flow state, and a hash map overhead. Hence, tests with 1 M or more

flows can not store the hash map in the L3 cache any more and need to perform accesses to main memory, which are slower by several orders of magnitude. This is a worst-case scenario with a uniform distribution of flow sizes. Traffic observed at ISPs or IXPs typically follows a Zipf distribution for the flow – the top flows are accessed more often and therefore more likely to be found in the CPU's cache.

Besides this effect of the hash map growing larger than CPU cache, another component of our system is affected when more flows are observed: The checker thread, which periodically iterates over the hash map to identify timed out flows, will take more time for its end-to-end pass. Figure 3 also plots the time required by the checker to iterate over the whole hash map with one thread. This sets a lower bound for the timeout of inactive flows and for periodic checks such as statistics reports. Note that there is currently only a single thread in FlowScope, extending this to multiple threads is possible but not necessary given these performance results.

Based on these results, we claim FlowScope to be suitable for tracking about 10 million flows without further optimizations. This is an order of magnitude more than other flow trackers [11] (≈200 K) while simultaneously offering more features and flexibility.

C. Validation with Real-World Flow Levels

We compare out benchmark against flow levels observed at a 80% utilized 10 Gbit/s ISP uplink. In one 24-hours period[1], we observe 960M flows, including TCP, UDP and other protocols. 97% of flows lasted less than 1 minute, hence we derive a worst-case median flow length of 1 minute. This results in an average of 666k parallel flows, which is easily within the 10 M flows our system is capable of, even when assuming peak loads of 2-3 times the average load.

We also observed an incoming DDoS attack on this link generating 3.4 M new flows per minute lasting for 53 minutes. Handling such attack scenarios is especially important: these unusual network conditions are exactly the time a analyzer or forensic tool like FlowScope is required. This shows that our improved performance is relevant for real-world scenarios.

VI. Conclusions

We presented a flexible flow tracker integrated into our FlowScope framework running on commodity hardware that is completely user programmable. FlowScope is not restricted to hard-coded flow types and can use arbitrary identifier keys for flows. As we anticipate 5-tuples as the most common use case, we ship flow key extractors for bidirectional 5-tuples on both IPv4 and IPv6 to simplify deployment. Despite the increased flexibility, we can track at least an order of magnitude more flows than other implementations. We have already used custom FlowScope modules on a uplink in a carrier-grade network for anomaly detection.

Reproducible Research Our code is available as open source on GitHub [6] under the MIT license. The repository's README file contains further information on running FlowScope as well as the raw data used here.

[1]Covering a Tuesday, i.e., a normal load

Acknowledgments

This research was supported by the German BMBF projects X-CHECK (16KIS0530), DecADe (16KIS0538), and SENDATE-PLANETS (16KIS0472).

References

[1] P. Emmerich, M. Pudelko, S. Gallenmüller, and G. Carle, "FlowScope: Efficient Packet Capture and Storage in 100 Gbit/s Networks," in *IFIP Networking 2017*, Stockholm, Sweden, Jun. 2017.

[2] Intel, "Data Plane Development Kit," `http://dpdk.org`. Last visited 2017-01-15.

[3] P. Emmerich, "libmoon," `https://github.com/libmoon/libmoon`, Technical University of Munich.

[4] P. Emmerich, "MoonGen," Technical University of Munich, `https://github.com/emmericp/MoonGen`.

[5] D. Scholz, P. Emmerich, and G. Carle, "Efficient Handling of Protocol Stacks for Dynamic Software Packet Processing in the Cloud," *Under submission to IEEE CloudNet 2018*, 2018.

[6] P. Emmerich, "FlowScope," `https://github.com/emmericp/FlowScope`, Technical University of Munich.

[7] B. Claise, B. Trammell, and P. Aitken, "RFC 7011: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information," 2013.

[8] ntop, "nBox: An Embedded NetFlow v5/v9/IPFIX Probe (IPv4, IPv6, MPLS)," `https://www.ntop.org/products/netflow/nbox/`.

[9] B. Trammell and S. Neuhaus, "libfc," `https://github.com/britram/libfc`, ETH Zurich.

[10] M. Trevisan, M. Mellia, M. Munafò, and D. Rossi, "DPDKStat: 40Gbps Statistical Traffic Analysis with Off-the-Shelf Hardware," 2016.

[11] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, and D. Rossi, "FlowMon-DPDK: Parsimonious per-flow software monitoring at line rate," in *Network Traffic Measurement and Analysis Conference 2018 (TMA'18)*, 2018.

[12] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[13] K. Cho, "Recursive lattice search: Hierarchical heavy hitters revisited," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17. New York, NY, USA: ACM, 2017, pp. 283–289. [Online]. Available: http://doi.acm.org/10.1145/3131365.3131377

[14] IEEE, "802.3ad-2000 - Aggregation of Multiple Link Segments," 2000.

[15] IEEE, "802.1Qbp - Equal Cost Multiple Paths," 2011.

[16] Google, "sparsehash," 2005, `https://github.com/sparsehash/sparsehash`.

[17] Facebook, "Folly: Facebook open-source library," 2017, `https://github.com/facebook/folly`.

[18] J. Preshing, "New Concurrent Hash Maps for C++," 2016, `http://preshing.com/20160201/new-concurrent-hash-maps-for-cpp/`.

[19] Intel, "Thread Building Blocks (TBB)," `https://www.threadingbuildingblocks.org/`. Last visited 2017-01-15.

[20] K. Barone-Adesi, A. Wingo, D. Pino, J. Muñoz, and P. Melnichenko, "pflua: Packet filtering in Lua," 2018, `https://github.com/Igalia/pflua`.

[21] M. Pall, "LuaJIT," `http://luajit.org/`, last visited 2017-01-15.

[22] L. Gorrie, "Snabb: Simple and fast packet networking," 2018, `https://github.com/snabbco/snabb`.

[23] J. Fanguede, M. Paolino, D. Dimitrov, and D. R. Virtual, "A novel pflua-based openflow implementation for vosyswitch," in *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, April 2018, pp. 43–49.

[24] WAND Group, "WITS: Waikato Internet Traffic Storage," `https://wand.net.nz/wits/index.php`.