FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Improving Flow Cache Efficiency in Network Flow Exporter
Student:	Damir Zainullin
Supervisor:	Ing. Jaroslav Pešek
Study program:	Informatics
Branch / specialization:	Information Security 2021
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

One of the fundamental concepts of network traffic monitoring and analysis for security purposes is the network flow, which represents aggregated information about packets associated with a particular connection. The network flows are then used for further analysis, such as network security threat detection and prevention. An application which implements the packet aggregation is generally called a flow exporter. The core functionality of each exporter is a flow cache — the memory that stores information about ongoing network flows and finds corresponding flow records for each packet. The implementation of the flow cache has a direct impact on exporter speed and quality of created flows.

Focus this bachelor thesis on optimising the open-source flow exporter tool ipfixprobe [1]. First, study the core concepts in flow exporters, focusing on flow cache and methods and techniques [2, 3, 4] for optimisation.

Analyse the ipfixprobe flow exporter [1] and research other exporters, focusing on the cache management techniques. Refactor the ipfixprobe's code, which implements the flow cache. Keep the interface and follow good practices and the style set in the project.

Identify the bottlenecks in the code and propose optimisations. Implement and experimentally evaluate these optimisations with commonly used metrics (time, number of steps, hits, and misses).

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 14 November 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

The thesis supervisor will supply test and benchmark data.

[1] https://github.com/CESNET/ipfixprobe
[2] Zadnik, Martin. "Optimization of network flow monitoring." Information Sciences and Technologies Bulletin of the ACM Slovakia 5.1 (2013): 6.
[3] Zadnik, Martin, and Marco Canini. "Evaluation and design of cache replacement policies under flooding attacks." 2011 7th International Wireless Communications and Mobile Computing Conference. IEEE, 2011.
[4] Li, He, et al. "FDRC: Flow-driven rule caching optimization in software defined networking." 2015 IEEE International Conference on Communications (ICC). IEEE, 2015.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 14 November 2023 in Prague.

Bachelor's thesis

IMPROVING FLOW CACHE EFFICIENCY IN NETWORK FLOW EXPORTER

Damir Zainullin

Faculty of Information Technology
Department of Information Security
Supervisor: Ing. Jaroslav Pešek
May 14, 2024

Czech Technical University in Prague
Faculty of Information Technology
© 2024 Damir Zainullin. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Zainullin Damir. *Improving Flow Cache Efficiency in Network Flow Exporter*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

I would like to express my sincere gratitude to my supervisor Ing. Jaroslav Pešek for his invaluable guidance and support throughout the completion of this work. His expertise and insightful feedback have been instrumental in shaping the outcome of this thesis.
I would also like to extend my thanks to the whole CESNET team for their valuable advice and contributions, which enriched the content and direction of this work.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 14, 2024

Abstract

This work is dedicated to the optimization of the currently used open-source network monitoring program – ipfixprobe developed primarily by CESNET, more precisely on its cache, as it's the core of flow-based monitoring. We will analyze the source code to identify approaches leading to performance issues and propose solutions for the identified problems. To prove the efficiency of our proposals we will experimentally test them, including a comparison of currently popular non-cryptographic hash functions and cache policies having good results adapted for specific conditions of the flow cache, usage of machine learning to create the best suiting policy for the exact type of traffic, and implementation of the DDoS detection algorithm. Successful proposals will become part of the project, increasing user service quality.

Keywords network monitoring, flow exporter cache optimization, cache replacement policy, flood detection, hash function test

Abstrakt

Tato práce je věnována optimalizaci v současnosti používaného open-source programu pro monitorování sítě – ipfixprobe vyvinutého primarně sdružením CESNET, přesněji řečeno jeho skryté paměti, neboť je jádrem monitoringu založenému na tocích. Bude analyzován zdrojový kód, abychom identifikovali přístupy vedoucí k snížení výkonu a navrhneme řešení pro identifikované problémy. Abychom prokázali účinnost našich návrhů, experimentálně je otestujeme, včetně srovnání aktuálně populárních nekryptografických hašovacích funkcí a správ skryté paměti přizpůsobených specifickým podmínkám skryté paměti síťových toků s dobrými výsledky, využití strojového učení k vytvoření nejvhodnější správy pro přesný typ provozu a implementaci detekčního algoritmu pro DDoS. Úspěšné návrhy se stanou součástí projektu a zvýší kvalitu uživatelských služeb.

Klíčová slova monitorování sítě, optimalizace skryté paměti exportéru toků, správa skryté paměti, detekce útoku zaplavou, test hashovací funkce

Introduction

Introduction

The global growth of the Internet due to its wide range of services means growth in users and devices [15]. Connecting all devices into one network means such a network's complexity and high link bandwidth. The Internet consists of many independent devices, which usually do not have much computer power and are designed for one, maximum few, tasks. Increased link bandwidths complicate the work of such devices, which results in the deterioration of services, degrading users' internet experience.

One of the most essential devices from the point of view of endpoint users is devices focused on security: there are many kinds of firewalls, Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), etc. Such devices must analyze passing data to detect network perimeter violations, anomalous traffic detection and filtration, inner network recourse accesses, etc.

Previously used solutions relied primarily on analyzing separate packets. Still, the rising amount of network devices and network traffic made it too expensive to analyze connections by analyzing packet payloads at the IP level, as the device with enough computing power would be expensive.

The second problem is the popularization of SSL/TLS network security protocols [8], which leads to the encryption of a significant fraction of data transferred by the Internet. A large amount of traffic is generated by web surfing, which is encrypted in almost all cases. Encrypted traffic cannot be decrypted without the decryption key, i.e., analysis is impossible without the key. Some solutions exist to confront the problem, like decryption and analysis of the traffic on network borders. Still, such solutions require user agreement and cooperation and thus could be used more in corporate than home networks.

New network loads and encryption required an innovative solution to provide security. The NetFlow protocol [5] was created to overcome the limitations of the packet payload-based analysis. It overcomes the weak sides of the payload-based approach by aggregating and analyzing only the metadata and packet headers of all connections, ignoring the payloads. This flow-based approach significantly reduces observing device load, but the device is still required to remember and manage all active flows. To process all passing packets in time, the observing device uses a space-efficient and time-efficient in-memory structure – cache. The cache is a core of the observing device, granting velocity and reliability of the exported data of the whole system. This work is focused on optimizations of existing open-source flow exporter ipfixprobe [4].

Objectives of this work are:

1. Research cache implementations of other flow exporters.
2. Analyse the source codes of the currently used ipfixprobe flow exporter, especially cache source codes.
3. Make the cache more readable and extensible by refactoring.

4. Identify parts of suboptimal code in the cache and propose possible improvements.
5. Experimentally test proposed improvements.
6. Compose a cache version with the best improvements and make a pull request back to the original repository.

Chapter 1
Network monitoring

In this work, we focus on a specific technique of network monitoring – IP flow tracking.

The flow is defined as a set of packets with the same properties (flow key) that can be observed on the observation point in a specified time interval [29]. These properties are usually source and destination IP addresses and ports, the transport layer protocol, and the time stamp from the defined time interval. Aggregated data are collected to analyze the flow: count of packets, count of transmitted bytes, sum of TCP flags, etc. Data can be collected for two separate unidirectional flows or one combined bidirectional. The flow record consists of the flow key and aggregated data, which can be analyzed and persistently stored. To standardize the flow record format, the NetFlow standard was created.

1.1 NetFlow

NetFlow is a Cisco protocol developed for network traffic monitoring. The NetFlow network architecture comprises exporters, collectors, analyzers, and communicating nodes. Usually, two unidirectional flows with opposite directions between the same nodes are combined into one bidirectional flow. The exporters are responsible for creating flows from passing packets and exporting them to one or more flow collectors. Exporters can be located on routers or separate devices. The main purpose of collectors is to gain flows from exporters and store and process them. The collected information can be used to see the overall network picture and detect anomalies and attacks. The scheme of the NetFlow architecture can be found in Figure 1.1.

If bidirectional flows are used, the fields are measured for source-to-destination and destination-to-source directions; in this work, by *flow* we mean *bidirectional flow*.

1.2 IPFIX

Based on Cisco's proprietary NetFlow protocol, a new open-source protocol was created: IPFIX [29]. The IPFIX protocol is very similar to the NetFlow standard of version 9. However, it has advantages over NetFlow, such as platform independence, custom export fields, and variable-length export records.

IPFIX and NetFlow have many common fields [29] [23]; exported fields can be found in Table 1.1.

Table 1.1 Export fields of NetFlow and IPFIX

Only NetFlow
The number of contiguous bits in the source and destination addresses subnet mask
IP multicast outgoing byte counter
Minimum and maximum IP packet length on incoming packets of the flow
Active and inactive timeouts in seconds
Source and destination MAC addresses
The fragment-offset value from fragmented IP packets
IPv4 or IPv6 next hop address
Bit-encoded field identifying IPv6 option headers found in the flow
Source and destination VLAN identifiers
Common
IP version number
Source and destination IP address
Packet counter. If a packet is fragmented, each fragment is counted as an individual packet.
Byte counter. The sum of the total length in bytes of all IP packets belonging to the flow.
Protocol type (TCP, UDP, ICMP ...)
Type of service octet (IPv4) or traffic class octet (IPv6)
TCP header flags
If the protocol type is TCP or UDP: source and destination TCP/UDP port number
Next hop IP address
Source and destination BGP Autonomous System numbers
Next hop BGP Autonomous System number
Multicast replication factor - the number of outgoing packets originating from a single incoming multicast packet.
Timestamp of the first packet of the flow
Timestamp of the last packet of the flow
In case of IPv6: Flow Label
If the protocol type is ICMP: ICMP type and code
Input and output interfaces
If sampling is used: sampling configuration
If MPLS is supported at the observation point: the top MPLS label or the corresponding forwarding equivalence class is bound to that label. The FEC is typically defined by an IP prefix.
Include the DiffServ Code Point that has a length of 6 bits.
Only IPFIX
Unique identifier of the observation point and exporting process
Time To Live (IPv4) or Hop Limit (IPv6)
IP header flags
Dropped packet counter at the observation point If a packet is fragmented, each fragment must be counted as an individual packet.
Fragmented packet counter counter of all packets for which the fragmented bit is set in the IP header

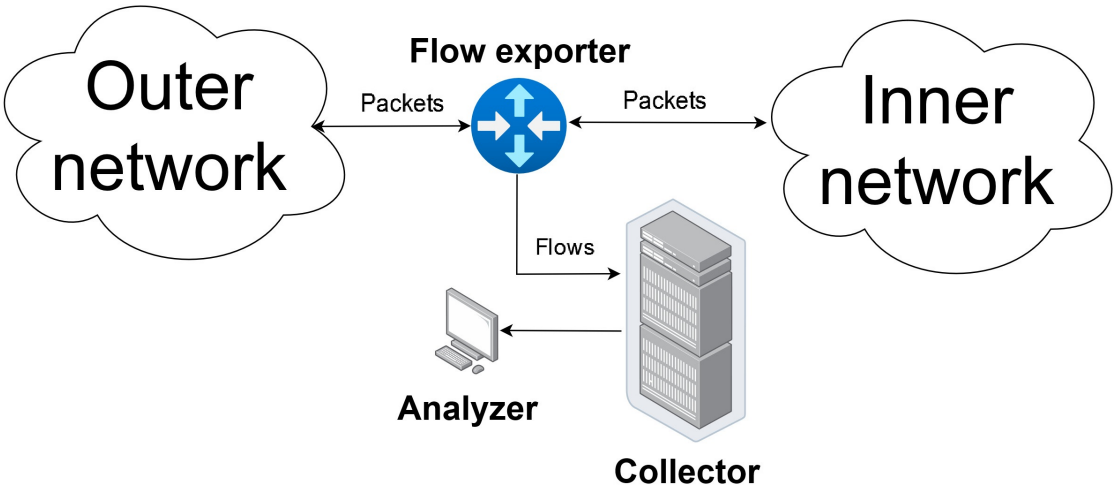

Figure 1.1 Scheme of the NetFlow architecture.

1.3 Exporter

Under typical circumstances, modern network bandwidth is measured in gigabits [2]. So, the exporters must collect packets into flows under high load because poor-quality exports significantly decrease the quality of every service that relies on exported data. This is why the NetFlow exporter is the bottleneck of the whole system.

1.3.1 Exporter implementation

To achieve its goal, the exporter uses a cache for flows. When a new packet arrives, the exporter makes a cache lookup; if a cache hit occurs, data of the flow record are updated, or a new flow is created; otherwise. For the best performance, the exporter must export every flow right after the last packet of a flow is processed. Practically, there is no way to reliably determine the end of a connection, except for TCP RST/FIN flags. This is the reason why inactive timeout was introduced. Since the last packet's timestamp has passed more than the established inactive timeout, the flow is considered as ended and being exported. Another problem occurs with flows that last too long. The flow data can't be analyzed until the flow ends, which allows various attack vectors. To avoid this, active timeout is introduced. The flow is exported since the first flow packet has passed more time than the active timeout.

1.3.2 Flow exporters

Before we start the analysis of our flow exporter, we analyze the implementations of caches of other flow exporters.

softflowd

The softflowd [16] is an old flow exporter written mainly in C. The main packet processing loop differs from the algorithm of ipfixprobe described in section 1.4. The main difference is the structure of a cache. softflowd uses two types of trees: red-black tree and splay tree, to keep its records. There is only one tree for cache, i.e., all records are kept in one big tree. When a

new packet arrives, it tries to find the flow in the tree, and if it is not found, it inserts a new record. After the maximum flow amount threshold is exceeded, the cache starts to export flows starting from flows that weren't accessed for the most time. The default cache size is 2^{13} records. A comparison by key fields is used to compare two flows, but hashing is not used. Both data structures consist of nodes, with pointers on left and right nodes, meaning that every access to the next node will call memory read on unpredictable address, which makes it hardware cache unfriendly. These tree-like data structures are simple and efficient for small amounts of flows, especially splay-tree, as its rules, with bubbling up to the top nodes that are accessed, are similar to the rules of LRU policy. Still, the high amount of active flows makes directing underlying data structures too expensive. The program structure doesn't support any plugins and can't be easily extended.

linux-flow-exporter

The linux-flow-exporter [27] has a cache written in C and uses the Linux kernel-provided Berkeley Packet Filter interface to create efficient storage. Implementation suggests only one possible type of storage – BPF hash-map. The usage of kernel storage allows kernel-level optimizations, such as network integration, allowing packets to be processed at the OS level and just-in-time compilation of user code that is executed in the kernel. There is a significant con to such an approach: there is no possibility to change the used hash function, the defined hash function is used to keep records, and we can't define our hash map – only choose from a few predefined types, which doesn't must to be optimal for our tasks. The default cache size is set to 8 records per CPU but can be increased to the maximum defined by BPF Linux subsystem limitations, which depend on the exact system. The program processes only TCP packets, which is a severe disadvantage. No support for custom plugins.

joy

The joy [14] is an open-source flow exporter created by CISCO. Every flow record in the storage is a node of 2 linked lists at once: a linked list of flows having the same hash value and a linked list of all flows in the order of their creation. To find the value, the hash is used, but in comparison to ipfixprobe, the hash value is an index to the head of the linked list of flows having that hash; comparison by key fields is used to find the exact flow in that list. To create the hash value, authors use their hash function, which is the sum of all 32-bit tuples of IP addresses, protocols, and ports multiplied by a constant. All IPv4 addresses are interpreted as IPv6, with a higher 96 bits set to 0. All flows are processed as unidirectional flow, but every flow record contains a "twin" pointer, pointing to the flow record in the opposite direction (if present). Custom plugins are not supported [26].

go-flows

The go-flows [6] flow exporter is written in the Go language. It uses default Go's hash-table implementation (called *map* in Go) to store its records [26]. The flow hash value is used as an index of the row, where all flows with colliding hash values are stored. To find exact flow comparison by key fields used. go-flows supports only bidirectional flows, and to find the flow identical to Algorithm 13 approach used, where we try to find flow by key fields. If the lookup isn't successful, a second search with exchanged source and destination addresses and ports occurs. This flow exporter allows the definition of any flow key, e.g., any combination of packet fields can be used to identify the flow. Exporter supports user-added extensions.

yaf

The yaf [24] flowmeter is written in C. To keep its record, it uses a hashtable-indexed pickable queue. It is a combination of hash table and queue, where each flow is a node of a linked list with pointers to its neighbors, while the node itself is in the hash table and can be randomly accessed by its hash. As implementation requires many allocations of the same flow record objects, a slab allocator is used, which maintains the pool of preallocated objects; when the object is freed, it is not returned to OS but back to the pool for future allocations. All flows are interpreted as bidirectional flows by default, and a second lookup is used to find possible reverse flows. User plugins are supported.

Vermont

The Vermont [9] flow exporter is written in C++. The hash table is used to keep records. The hash of the flow is used as the index to the hash table; every entry keeps a linked list of flows with the same hash value, where comparison by key fields is used to find the exact flow. A hash function is its implementation of the CRC32 hash. In biflow mode, a second hash of reversed key fields is used to find possible reversed flow. Any field set can be used as a flow key. The exporter is extensible by plugins.

CICFlowmeter

The CICFlowmeter [3] exporter, including its cache, is written in Java. The storage of active flows is implemented by default Java's *HashMap* implementation, where keys with the same hash values are added to the linked list. Flows are interpreted as bidirectional by default. Two lookups occur to find the flow for direct and reverse key tuples. The flow key is the string in the format "src_ip-dst_ip-src_port-dst_port-protocol" (or reversed), i.e., the key tuple is fixed and cannot be changed. Index to the hash table is the hash value of a string, which has a default implementation in Java: assume we have a string s, the hash value is $\sum_{i=0}^{s.length()-1} s[i] \cdot 31^{s.length()-1-i}$. Custom modules are not supported.

1.4 ipfixprobe

In this work, we try to find and negotiate problems with an existing open-source flow exporter ipfixprobe [4] developed primarily by CESNET.
The program is built of plugins: input, output, process, and storage. Input plugins are responsible for reading packets, e.g., from a PCAP file or an interface. Output plugin exports expired records in different formats, e.g., IPFIX or custom UniRec [25]. Process plugins can serve different purposes, such as exact protocol processing or statistics measurement. The main part of the exporter is the storage plugin, which is responsible for processing packets into flows. Architecture is described schematically in Figure 1.2
The hash table is divided into rows. The LRU replacement policy manages each row. The main problem with cache replacement policies is that they usually have no defined behavior if some elements in managed rows are externally deleted. In our case, it is flow exporting after the active or inactive timeout. The current implementation periodically scans part of memory and exports expired flows. This means that sometimes, unexpectedly, empty places appear in the managed row.

The main function that describes incoming packet processing on a high level is a *process_packet* from Algorithm 1 or in Figure 1.3.
Used simple functions and variables are defined in Table 1.2.

Generally, the algorithm of packet processing works as follows:

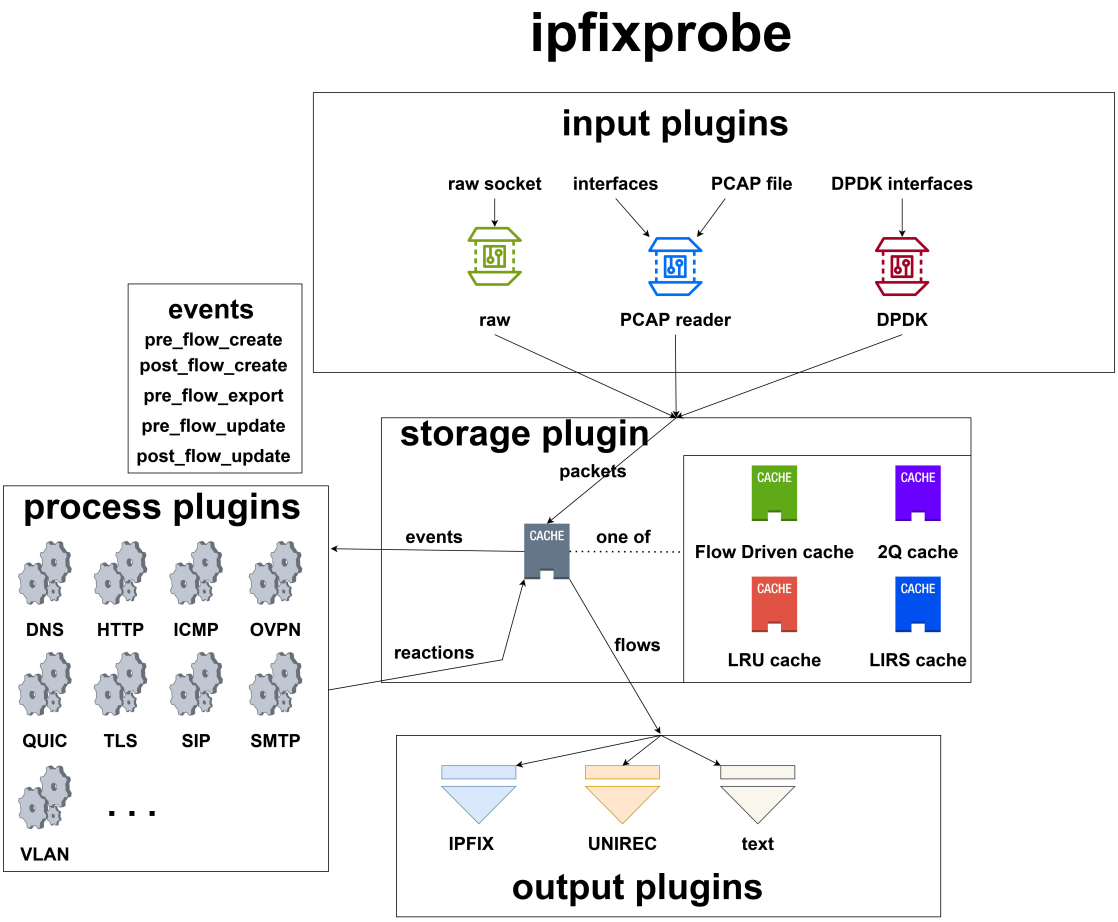

Figure 1.2 Architecture of ipfixprobe.

1. It tries to fill ports in case the packet is fragmented.
2. The probe must create a hash key of the packet. Currently, there is an XX64 [7] hash function that creates a hash from a direct flow key tuple and the second hash value from the reversed tuple to find the flow if the first packet of the flow was observed with exchanged addresses and ports.
3. Then it looks up in the hash table with created hash values. Lower bits are used as the index of the row.
4. After the row is found, it tries to find if the flow is already in memory by comparing hash values.

The second and third steps are the *find_flow* function, which described in Algorithm 13 If the flow is found, it updates the appropriate counters and moves the updated flow to the row begin, according to LRU policy. Shifts of the flows in a row are defined by *circular_shift* operation from Algorithm 17. Flow enhancement is described in Algorithm 14.

If the flow is not found, we must create a new one. New flow creation is in *create_new_flow* from Algorithm 15. Here is the main difference with classic LRU - it first tries to find an empty place in the current row (in Algorithm 16), and only if there is no empty place, it exports the last flow in the row, shifts all flow from the lower half of the row by one in the direction of the

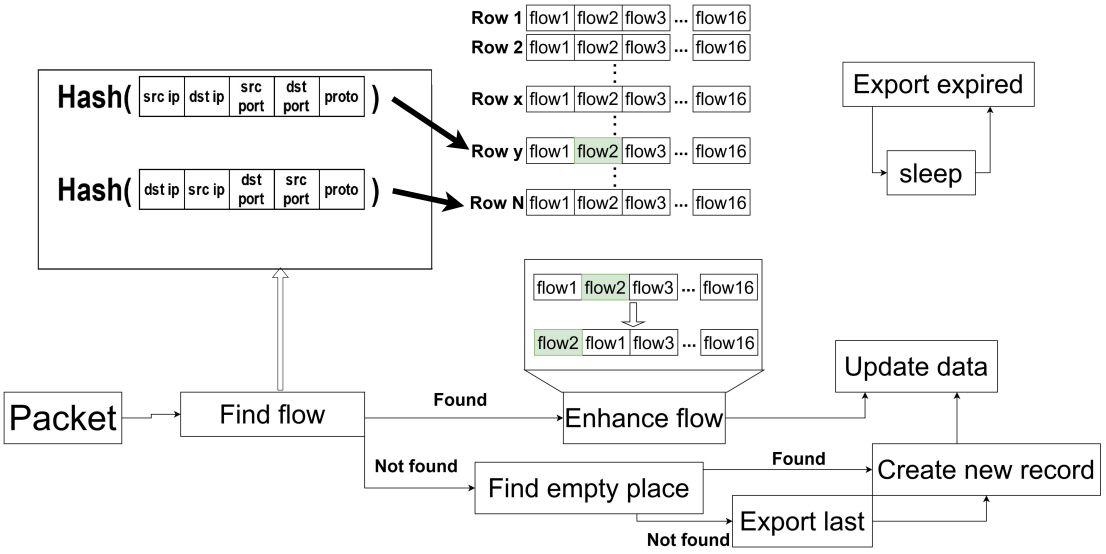

Figure 1.3 ipfixprobe packet processing.

row end, which defined as free_place_in_row from Algorithm 18. The new flow uses the created empty place in the middle of the row.

We analyzed the program run on Mawi PCAP (section 4.1) by kcachegrind. We must notice that the profiling results may significantly differ depending on the type of input traffic. The results are also impacted by compilation options – compilation with a higher level of optimization leads to significant changes in the program call tree, making data we are interested in unavailable. We chose a PCAP with a higher inter-packet distance and a small number of exports due to the lack of space. Results can be found in Figure 1.4

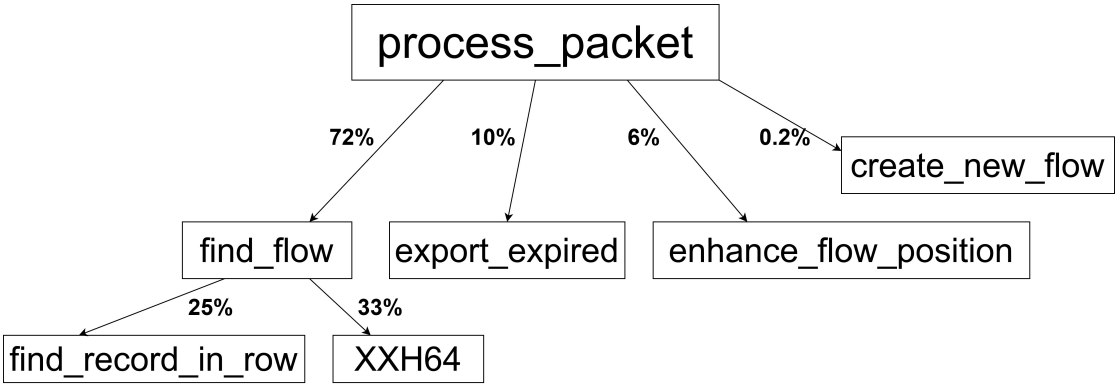

Figure 1.4 Results of profiling the Mawi PCAP by kcachegrind.

Algorithm 1 process_packet

```
Input: IP Packet packet
fill_ports_if_packet_is_fragmented(packet)
flow_index = 0
if find_flow(flow_index, packet) then
   if active_or_inactive_expired(flow_index) then
      export(flow_index)
      process_packet(packet)
      return
   end if
   enhance_flow_position(flow_index)
else
   row_begin = get_row_begin(flow_index)
   create_new_flow(row_begin, packet)
   export_expired()
end if
return
```

Table 1.2 Description of used variables and function in Algorithms 1 - 18.

Labels
fill_ports_if_packet_is_fragmented (packet) – a function that is used to increase the cache efficiency; it keeps the second internal smaller cache that keeps fragmented packet headers that will be completed after the next fragment arrives. It fills up missing packet ports, which allows for the precise determination of the packet flow.
active_or_inactive_expired (flow_index) – is a function that returns true if the flow pointed by $flow\_index$ has active or inactive timeout expired.
get_row_begin (flow_index) – used to get the flow record index of the first record in the row to which the $flow\_index$ belongs.
export_expired() – function check and exports, if record is expired, $\frac{line\_size}{2}$ records, starting from n, next function call will start at $n + \frac{line\_size}{2}$ index.
low_bits (int) – hash functions usually return long hash codes: 128 or 64 bits. We need only n bits to address the cache flow table with 2^n records. low_bits returns lower n bits of the int.
XXH64 – is a currently used hash function (defined in 1.1).
flow_table – is a table that keeps all records managed by the cache.

▶ **Definition 1.1.** *A hash function H is a function $\{0,1\}^n \rightarrow \{0,1\}^k$, where $n, k \in \mathbb{N}$, converting any input of length n bit to the output of the length k bit, having k fixed for each hash function.*

Chapter 2
Analysis

Firstly, the criteria for the improvement must be established. The idea of the cache itself suggests two options: hit rate and speed. Increasing the hit rate would mean increasing the count of flows successfully found with new incoming packets and decreasing the count of flows prematurely exported due to lack of place in a row. Increasing the cache speed means decreasing the average time the cache spends processing one packet. Analysis of the program by profiler (Figure 1.4) shows that the majority of time cache spends creating hash values of incoming packets and searching the flow with this hash value in memory, so our improvements are focused on this part of the program.

2.1 Double hashing

As described in section 1.4, the cache creates two hashes from the packet fields. First, to find the flow if the first packet of the flow was observed having the same source and destination addresses and ports, and second, if we are observing the response from the destination.

2.2 Hash function

The second possible source of problems is the hash function itself. These two hash function properties are the most important for our purposes: speed and collision resistance. Low hashing speed leads to overall slowing and, in the worst cases, to packet drops. On the other hand, no strict comparison of flows is used because of speed, i.e., comparisons are made only by hash values. Low collision resistance leads to combining many flows into one. After such a mixed flow is exported, none of the analyzing applications can come to a sensible conclusion from such data. The second problem is the hashable data itself. Almost all of its fields have low entropy:

1. The probe will probably be installed on the gateway router dividing the inner network and Internet, meaning that one of the source or destination IP addresses will be from a relatively small range.
2. Majority of the traffic on the Internet belongs to HTTP, which, along with many other applications, like DNS or SSH, uses well-known ports. Even if ports on the client side have a wide range of possible values, one of the source or destination ports is predictable.
3. Majority of the traffic is transported by, decreasingly, TCP, UDP, and ICMP protocols. Practically, the next-level protocol field has only a few possible values.

4. Finally, all of those values must be combined to create a value that is unique among, in order, hundreds of thousands of records.

To prove the point about entropy, we analyzed our PCAP files described in section 4.1, results are at Tables 2.2 and 2.1, *Port* and *Usage (%)* shows the proportion of how many packets contain such ports as source or destination. We assume that every application that uses port 443 is HTTPS, 80 – HTTP, 22 – SSH, and 53 – DNS.

As we can see, for every case, 90% of protocol field values belong to TCP and UDP; for ports, the situation is similar; HTTPS is a significant part of traffic (with the exception of Mawi PCAP).

Table 2.1 Usage of protocols by different PCAP files

PCAP file	Protocol	Usage (%)
Tul	TCP	81
	UDP	17.4
Traffic	TCP	66
	UDP	33
Sh	TCP	87.7
	UDP	11.8
Mawi	TCP	50.2
	UDP	39.7
	IPv6 encapsulation	8.6

Table 2.2 Usage of ports by different PCAP files

PCAP file	Port	Usage (%)
Tul	HTTPS	73
	HTTP	3.1
	SSH	1.6
Traffic	HTTPS	70
	HTTP	7
Sh	HTTPS	76
	HTTP	10.9
Mawi	HTTP	27.9
	SSH	7.4
	DNS	1.8

The currently used function is XXH64, a hash function from the XXH family [7]. The XXH hashes are widely used because of their speed and low collision count. The XXH family fulfills the criteria of the SMHasher test suit [1], designed to test the distribution, collision, and performance properties of non-cryptographic hash functions.

For further tests, we have chosen popular non-cryptographic hash functions nowadays: the fastest hash functions from the XXH family, such as XXH3_64bits and XXH3_128bits, CRC32c

hash function – CRC32 implementation of Google, FarmHash, Murmurhash, SuperFastHash, and Toeplitz hash functions.

2.3 Cache row policy

The next source of the slowdown is how flows are placed in the rows. The idea of a hash table divided into rows implies that only part of the target hash value is used as an index to the exact row, and any strategy could be used to find, remove, or create a new item. *process_packet* algorithm described at Algorithm 1 can be used as a template for any new row policy; we only need to redefine a few operations that the row policy must implement:

▶ **Definition 2.1.** ***Find flow****(row_begin, hash) - Operation that takes the index of the beginning of the row as a parameter and the hash of the flow to find. Returns the index of found flow or signals that the row doesn't contain the flow.*

▶ **Definition 2.2.** ***Find empty place****(row_begin) - Operation that takes the index of the beginning of the row. Returns the index of empty place in the row or signals that the row is full and contains no empty place.*

▶ **Definition 2.3.** ***Enhance flow****(flow_index) - Operation that takes the index of the flow and processes the row to improve the next* ***Find flow*** *operation for the flow which is currently on flow_index.*

Enhance flow is an important operation from the point of view of the temporal locality. We expect the recently accessed flow to be reaccessed soon, so we define an operation that helps to find this flow more efficient on the next access. By increasing the efficiency of the Find flow operation, we mean decreasing the overall time spent by the operation or increasing the chance that flow will be found, i.e., it was not exported due to lack of space since the last access.

▶ **Definition 2.4.** ***Free place in row (row_begin)*** *- Operation that takes the index of the beginning of the full row without any empty place and chooses and exports the most suitable record using an algorithm determined by the row policy.*

Operations above are presented in ipfixprobe (algorithms can be found in Appendix A): **Find flow** is a *find_flow* from Algorithm 13, **Enhance flow** is *enhance_flow* from Algorithm 14, **Free place in row** is a *free_place_in_row* defined in Algorithm 18, and **Find empty place** is *find_empty_place* from Algorithm 16.

LRU

The currently used implementation is a slightly improved LRU. LRU stores the flows sorted according to their last access time: the most active flow is in the first position of the row, and the least accessed is in the last. When flow inside the row is accessed, all flows, from the beginning of the row to the accessed flow position, are shifted down by one, and the accessed flow is placed to the freed position at the beginning of the row. When we need to add a new flow to the row, we use the first found empty place, or if there is no empty place, we export the last (least recently used) record, the lower half of the row is shifted down by one, and the freed place at the middle of the row is used. The definition of the operations above can be found in section 1.4

Improvement

The idea for the improvement is to implement and compare other possible row management policies. There are two important criteria for the comparison:

1. For obvious reasons, total time spent manipulating flows within defined operations.
2. Total count of exported flows - if the cache uses an unsuitable policy, there will be exports of still active flow, which leads to splitting up one flow into many. It's much harder to analyze such flows on the collector. As the collector is usually located on a simple device that doesn't have much computing power, we can't expect the collector to unite such split flows. On the other hand, updating the existing flow in our implementation is faster, so creating a new flow instead of updating the old one increases overall time.

Chapter 3
Improvements implementation

This chapter is dedicated to the solutions to problems highlighted in previous chapters. The source codes of all implementations below can be found in the public repository[1].

3.1 Refactoring

The first part of my work was to refactor existing code. The original version of the code was hardly readable and understandable. Also, there was almost no way to use C++ class inheritance, which was very significant for creating cache extensions. That was done by dividing large files with many classes and structures into small single-purpose files. After that, all functions were split into smaller parts. Part of the refactoring was also to make possible simple hash function replacement and rows manipulation functions part of the class interface. This led to simple extensions and modifications of hash functions and row policies. The ipfixprobe repository maintainer verified the results of refactoring. The refactored cache is schematically displayed in Figure 3.1.

3.2 Double hashing

Original code hashes every incoming packet twice - Algorithm 13: {source IP, destination IP, source port, destination port, transport layer protocol, IP version, VLAN ID} and {destination IP, source IP, destination port, source port, transport layer protocol, IP version, VLAN ID} and looks up twice, the second hash will find its flow if the flow is already in memory, but the initial packet was observed having current destination IP address as source. Assuming that we are interested in primarily bidirectional flows, which bring the most useful information for collectors, we can avoid double hashing. Instead of hashing data twice, we assume that hashing is an expansive operation so that we can hash sorted packet fields. If the source IP address, as a number, is bigger than the destination one, we swap them; ports are swapped, too. Only one hash lookup is enough to determine if the appropriate flow is already in memory. But then the next problem arises – after the packet fields were sorted, there is no way to determine the original order of the fields, i.e., no way to determine if the source-to-destination or destination-to-source packet was captured. Without that information, there is no way to update separate statistics for each direction. To solve that problem, we must extend the flow record structure with a boolean flag set to true if addresses and ports are swapped within sorting. Now original functionality is restored, what is the price for this improvement?

[1] https://github.com/Zadamsa/ipfixprobe-cache

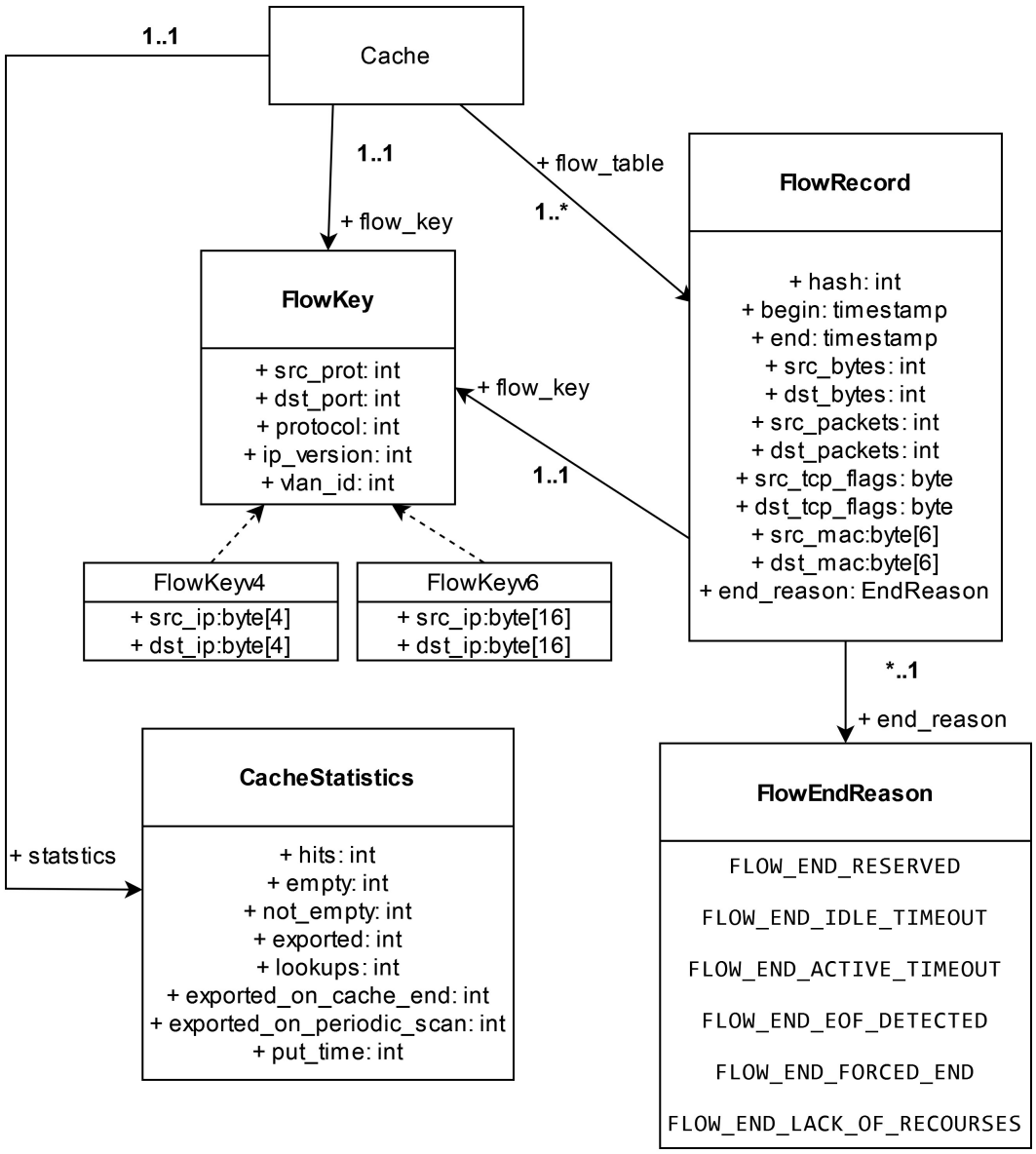

Figure 3.1 Class diagram of refactored ipfixprobe

Increasing total memory usage - every flow record is kept if its fields are swapped. In our case, the default cache size is 2^{17} records, but in practice may sometimes be increased to 2^{21}. Memory alignment leads to 128KB-1MB of additional space, which is not critical.

3.3 Hash function

As mentioned above, the XXH64 is currently used as a hash function; in this section, we compare other possible candidates to replace XXH64.

XXH3_64, **XXH3_128** [7] – both hash functions are from the XXH3 family, which is the improvement of the original XXH family. XXH family has built-in processor acceleration support. Both hash functions appeared in the same year, 2018, a few years after the active probe development stopped. XXH hashes are based on classic bit XOR and rotate operations.

Toeplitz hash [19] is based on linear operations with matrices. The data we want to hash are loaded into the Toeplitz matrix, multiplied by the vector, which is interpreted as the key. The result of the multiplication is interpreted as the hash. Assuming we have a matrix $n \cdot n$ and vector with length n, the time complexity of standard multiplication would be $O(n^2)$, but using a special Teoplitz matrix form, the multiplication can be done in $O(n \cdot log(n))$ time [22]. To hash new incoming packet fields, we must upload their fields to the Toeplitz matrix and multiply them by a predefined key. The DPDK, a big library for processing packets, widely uses this approach. In our tests, we use the implementation of DPDK.

CRC32c [11] is a hash function that calculates a 32-bit cyclic redundancy check optimized for the hardware. This implementation uses other than the default polynomial to provide better hash distribution and speed. Hashing itself is done by polynomial division of input data by polynomial over the GF(2). We use Google implementation for tests.

MurMurHash3 [1], the third version of MurMurHash, belongs to hash functions based on bit operations such as XOR and shifts. Optimized for x86 architecture. The main objective of that hash function is a good distribution with low collisions.

FarmHash [12] originally appeared as a concurrent for CityHash and MurMurHash. Has built-in processor acceleration support. Based on XORs and bit rotations. Implementation is much more complex compared to MurMurHash3 and CRC32.

SuperFastHash [17] – the old hash function appeared in the year 2001. Based on XOR and shifts. The main purpose was to create a simple hash function. Implementation of the hash function itself takes about 40 lines in C.

3.4 Row policy

This section discusses managing schemes of different row policies, which could show better results than the original LRU.

3.4.1 Heap LRU

The idea behind this improvement is to replace the array inside every row with a minimal heap. The key comparison for heap elements is a timestamp of the last observed packet. In this case, our operations are redefined as follows:

1. Find empty place – as for the original LRU implementation, we can't predict the following flow that will be exported because of timeouts, so empty places are randomly distributed over the whole row. The only way to find such empty records is to check all records one after one, as in Algorithm 16. The time complexity of this operation is $O(n)$, where n is the length of the row.

2. Find flow - here we can use a similar approach as in Algorithm 13, but in comparison to the original algorithm, the most active flows are at the lower half of the row now, so we search for the flow starting at the end of the row, ending at the beginning.

3. Enhance flow - here, we meet significant simplification of the heap's operations - as the new packet has the highest timestamp in comparison to other flow, so the target flow must be moved down by *bubble_down* (Algorithm 2) to the lowest possible position. After that, the heap is still in a valid state. In the worst case, we must bubble down the record from the top of the heap, as the heap has the height of $\log(n)$, where n is the count of records in one row, so the total complexity of this operation is $O(\log(n))$.

4. Free place in row - is called if the current row is full and empty place for inserting new flow is required. In this case, we export the element on the top of the heap, as it has the lowest last access time. As the newly freed place belongs to the new flow, it has a last access timestamp equal to the current time, i.e., maximal, compared to other flows in the row. To keep the heap in the valid state, we bubble_down (Algorithm 2) the element from the top of the heap. $O(1)$ for export and $O(\log(n))$ for bubble_down leads to overall $O(\log(n))$.

Algorithm 2 bubble_down

```
Input: index of the flow to move to the lowest layer flow_index
row_begin = get_row_begin(flow_index)
row_end = row_begin + cache_line_length
while (flow_index − row_begin + 1) · 2 ≤ cache_line_length do
  left_leaf_index = (flow_index − row_begin + 1) · 2 + row_begin − 1
  right_leaf_index = (flow_index − row_begin + 1) · 2 + row_begin
  swap_target = −1
  if right_leaf_index ≥ row_begin + cache_line_length then
    swap_target = left_leaf_index
  else if flow_table[left_leaf_index].is_empty() OR
          flow_table[left_leaf_index].is_empty() then
    if flow_table[left_leaf_index].is_empty() then
      swap_target = right_leaf_index
    else
      swap_target = left_leaf_index
    end if
  else
    if flow_table[left_leaf_index].last_access_timestamp <
    flow_table[right_leaf_index].last_access_timestamp then
      swap_target = left_leaf_index
    else
      swap_target = right_leaf_index
    end if
  end if
  swap(flow_index, swap_target)
  flow_index = swap_target
end while
return
```

As can be observed from Table 3.1, heap LRU has all operations at the same level or better. Of course, asymptotic complexity can't be the only criterion. The problem is a hidden multiplicative constant. The main advantage of the original LRU scheme is that the most active flows are at the beginning of the row, and there is a guarantee that the last accessed flow is in the first position

Table 3.1 Comparison of asymptotic complexities of LRU on heap and default LRU on array

Operation	Original LRU	Heap LRU
Find empty place	$O(n)$	$O(n)$
Find flow	$O(n)$	$O(n)$
Enhance flow	$O(n)$	$O(\log(n))$
Free place in row	$O(n)$	$O(\log(n))$

of the row. In a heap LRU, the most active flows are somewhere in the last layer of the heap, i.e., we need to check $\lceil n/2 \rceil$ flows to find it. We expect to see a higher average amount of tries to find records.

3.4.2 Flow-Driven Rule Caching

This policy was initially developed to effectively cache packet-processing rules on switches with Ternary Content Addressable Memory [20].

In the SDN architecture, all network switches are controlled by a centralized controller. When a switch can't find the appropriate rule for the flow to which the new incoming packet belongs, it must request the controller for the new rule. Those requests slow down the network, so the target of the work is to reduce the number of cache misses on switches caches, which also matches the purposes of this work. The authors of the original work came up with a heuristic that tries to predict when a new packet for every flow arrives. They divided all flows into two groups: predictable and unpredictable flows.

1. Predictable flows are flows for which we can reliably determine the timestamp of the next packet, e.g., the flows from deterministic network services.

2. Unpredictable flows are all other flows to which we can't determine the arrival time of the next packet.

In this work, all flows are threatened as unpredictable, as we expect packets to be encrypted, meaning we can't analyze signals from the packet body.

The algorithm works very similarly to the First-In-The-Future replacement strategy, but, for obvious reasons, values of heuristic functions are used instead of the actual timestamp of a packet from the future. The problem is that the original work doesn't mention a way to find a value with maximal value in the row. On SDN switches with CAM, this can be done trivially in $O(1)$, while in our linear software implementation, it would be $O(n)$ hard. This raises the question of how some effective implementation could be used. Still, neither heap nor sorted array as in the original LRU implementation can be used, as the values of timers change spurious, which means that when we access some row, we can't be sure that the sorted storage is in a valid state, meaning that we need to check and repair correctness of the row on each access, which seems not to have a simple solution. For this test, we use inefficient implementation only to test the statics of the proposed algorithm. Creating an optimized implementation only makes sense if this replacement strategy provides a good enough cache-hit ratio.

The heuristic function value is interpreted as the value of the timer. If the timer expires, a new value is calculated and assigned.

The only tunable parameter of the policy is T_{max}. Defined operations:

1. Find flow and Find empty place operations work like the original LRU.

2. Free place in row – the flow record with the highest heuristic value (timer) is removed, and trivial O(n), because of flow searching, implementation is used.
3. Insert new element – after getting an empty place, we set up a timer for the record with a T_{max} value.
4. Enhance flow – set timer value as the difference between the timestamp of the new incoming packet and the last packet of the flow.
5. The record with the highest timer value is exported.

To set the new value for the timer after it expires, we use Algorithm 3. The value of T_{max} was set to 2 seconds.

Algorithm 3 Update timer FDRC

Input: index of the flow with expired timer *flow_index*
timer_start_value = timer_started_with_value(*flow_index*)
if *timer_start_value* $\neq T_{max}$ **then**
  flow_table[*flow_index*].set_timer(max(2 · X,T_{max}))
else
  //If the initial expired timer value was T_{max}, the value of the timer is set to T_{max}, but the timer is not started again, and its value stays frozen at T_{max}, i.e., if the timer set to T_{max} expires, we consider that we have already processed the last packet that belongs to flow and flow can be exported.
  flow_table[*flow_index*].freeze_timer_with_value(T_{max})
end if

3.4.3 LRU2Q

The proposed algorithm is an improved version of the LRU-2, a particular case of LRU-K, which uses a time of K-th access to the block to sort the block in the row. The problem that the authors of the original work had with the LRU-K itself is that LRU was initially developed for hardware use, i.e., using Content Accessible Memory, we can find the required block or an empty place in the constant time while using LRU-K requires the heap data structure, which leads to the logarithmic complexity for same operations. In our case, it is not such a big deal, as our cache is implemented on the software level and has linear complexity. To overcome the logarithmic complexity of the LRU-2, LRU-2Q [18] was created. Both algorithms provide the same functionality, but LRU-2Q has constant time complexity with hardware implementation. In our case, it is still linear.

To achieve its purposes, LRU-2Q divides the row into parts. The parts size is the tunable part of the algorithm. We try to find the best parts sizes for our cache. The algorithm has two versions: simplified and full. We test both versions.

LRU2Q Simplified

The simplified version divides rows into two parts: main (Am) and A1. Firstly, data appear in the A1 queue, managed as FIFO. When a page inside an A1 queue is referenced again, it moves to the main buffer, which is a standard LRU buffer.

The authors of the original work expect that the first use of the block doesn't mean that the block is long-term used. If the block is shifted from the A1, it won't be accessed soon. However, if the block is accessed inside the A1, we place it in the Am as a potentially long-term used block. Operations are redefined as follows:

1. Find flow – similar to the original LRU.
2. Find an empty place – Linear lookup over the row. We return the first found empty place, regardless of whether it is inside A1 or Am.
3. Enhance flow – regardless of whether the target flow record is inside A1 or Am, it is moved to the beginning of the Am. If the Am is full, we need to export the last record. The description with pseudo-code is Algorithm 4.
4. Remove record – to free a place in the row, we remove the first record in the A1 and move an empty place to the last place in the A1 buffer using circular_shift from Algorithm 17.

Algorithm 4 enhance_flow

```
Input: index of the flow to enhance flow_index
row_begin = get_row_begin(flow_index)
if flow_is_inside_Am(flow_index) then
  circular_shift(row_begin, flow_index)
else
  empty_place = find_empty_place_in_Am(row_begin)
  if empty_place == NO_EMPTY_PLACE then
    last_record_in_Am = row_begin + Am_length − 1
    export(last_record_in_Am)
    empty_place = last_record_in_Am
  end if
  swap(flow_table[empty_place], flow_table[flow_index])
  circular_shift(row_begin, empty_place)
end if
```

The main target for this task is to determine the best parameters of the configuration, i.e., sizes of A1 and Am.

LRU2Q Full

The full version was proposed to improve the results of the simplified version. It uses three buffers: A1in, A1out, and Am. The improved version was proposed to beat "correlated reference." Correlated reference means repeated accesses to the same location in the cache shortly after the access but no accesses in the subsequent long-term period. To avoid adding blocks to Am on correlated references, the authors introduced Aout. Now, if we access the block in the A1in, we do nothing, ignoring the correlated reference.

Operations are redefined as follows:

1. Find flow – same as the original LRU.
2. Find empty place – linear lookup over the row.
3. Enhance flow – if the record is in A1in, do nothing. If it is inside the Aout or Am, move it to the beginning of Am, similar to Algorithm 4. If we move the record from Aout to the Am and there is no empty place, we export the last record from the Am.
4. Free place in a row – to make a free place in the row, we shift all records in Ain and Aout records, removing the last record from the Aout and making the last record from Ain first record in Aout. The new place is returned at the beginning of the Ain then.

The main target for this task is to determine the best values for sizes A1, Ain, and Aout.

3.4.4 LIRS

The Low Inter-reference Recency Set (LIRS) replacement policy [21] was invented to solve the problems of LRU. LIRS adds Inter-Reference Recency (IRR) to every flow record in the cache. IRR of the flow is calculated as the count of accepted packets between the last and second-to-last accesses to the flow. IRR is used as a heuristic, and the authors of the original work expect that if the flow has a high IRR, it will have a high IRR in the future, too. All flow records are then divided into Low IRR (LIR) and High IRR (HIR) blocks. LIR blocks are expected to be reaccessed soon. HIR blocks are divided into resident and non-resident blocks. In our implementation, we use the original row, which is divided into two parts:

1. The "list" part. This part of the row contains all flow records perceived as resident HIR blocks.

2. The "stack" part contains a mix of LIRS, resident, and non-resident HIR blocks.

Resident HIR blocks are blocks presented in a list; non-resident HIR blocks are contained only in the stack part.

The authors defined the operation "stack pruning" as removing all elements below the last LIR block, described in Algorithm 5. LIRS redefines operations as follows:

Algorithm 5 prune_stack

Input: index of the row beginning *row_begin*
for $row\_begin + cache\_line\_length - 1 \geq i \geq row\_begin + list\_size$ **do**
  if *flow_table*[*i*].is_empty() **then**
    continue
  else if *flow_table*[*i*].*reference_type*== *LIR* **then**
    return
  else
    export(*i*)
  end if
end for

1. Enhance flow – described by Algorithm 6. The strategy depends on the type of the flow.

   If the flow is LIR, it is moved to the top of the stack, and all HIR blocks before the new last LIR block are removed from the stack.

   If the flow is resident HIR, it depends on if the block is in the stack:

   If it is, we change its status to LIR and move it to the top of the stack. As it is not an HIR block anymore, we remove it from the list. The last block of LIR status changes its status to HIR and adds itself to the end of the list. Then, the stack pruning occurs. If it isn't, we move it to the end of the list.

   Upon accessing the non-resident HIR block, which can be found only on the stack, we change its status to LIR and move it to the top. Then, we switch the last LIR block status to the HIR and add it to the end of the list. Then stack pruning occurs.

2. Find flow – similar to the original LRU, linear search across the whole row.

3. Free place in row – used when the flow was not found, and an empty place is needed. To create an empty place, we delete the first item of the list and create a new record at the end of the list.

4. Find an empty place – linear lookup over the whole row. The newly created record gets an LIR status if the place is found in the stack. The new status is a (resident) HIR, if the place is found in the list.

Algorithm 6 enhance_flow

Input: index of the flow to enhance *flow_index*
//LIR block can get hit only in the stack
if *flow_table*[*flow_index*].*reference_type* == *LIR* **then**
  circular_shift(*row_begin* + *list_length*, *flow_index*)
else if *flow_index* < *row_begin* + *list_length* // Resident HIR block got hit in the list **then**
  circular_shift(*row_begin*, *flow_index*)
  flow_index_in_stack = find_in_stack(*flow_index*)
  if *flow_index_in_stack* ≠ *NOT_FOUND* **then**
    circular_shift(*row_begin* + *list_length*, *flow_index_in_stack*)
    flow_table[*flow_index_in_stack*].*reference_type* = *LIR*
    flow_table[*flow_index*].*clear*()
    last_lir_index = find_last_lir_index(*row_begin*)
    flow_table[*last_lir_index*].*reference_type* = *HIR*
    flow_table[*row_begin*] = *flow_table*[*last_lir_index*]
    prune_stack()
  end if
else
  //Non-resident HIR got hit
  flow_table[*flow_index*].*reference_type* = *LIR*
  circular_shift(*row_begin* + *list_length*, *flow_index*)
  empty_place_in_list = find_empty_place_in_list(*row_begin*)
  if *empty_place_in_list* == *NO_EMPTY_PLACE* **then**
    last_record_in_list = *row_begin* + *list_length* − 1
    export(*last_record_in_list*)
    circular_shift(*row_begin*, *last_record_in_list*)
  else
    circular_shift(*row_begin*, *empty_place_in_list*)
  end if
  last_lir_index = find_last_lir_index(*row_begin*)
  flow_table[*last_lir_index*].*reference_type* = *HIR*
  flow_table[*row_begin*] = *flow_table*[*last_lir_index*]
  prune_stack()
end if

3.4.5 Adaptive policy

Žadník proposed the original idea in their work [28]. The problem with all previously discussed policies is that too many possible types of traffic can be observed. The observed traffic kind differs for web surfing and torrent downloading, or the intensity of flows created by a small set of devices over the home and big corporate networks. Instead of trying to find the row policy suitable for every case, we can create the one that suits the most for some particular case.

The original work suggests creating a row policy that can be adapted to some exact example of network traffic.

Compared to other row policies suggested previously, we need to define some general policies that can be tuned to fulfill the requirements of the traffic type specific to some network type. Modification of LRU was suggested:

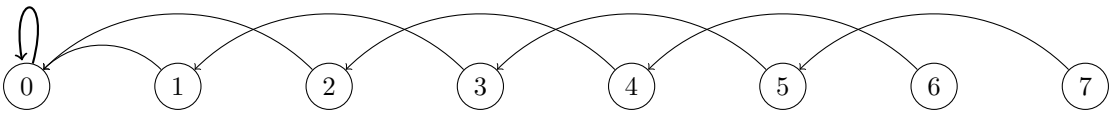

Figure 3.2 Configuration [{0, 3, false},{1, 5, true}] displayed graphically

1. Find flow or an empty place work the same as in the original LRU.
2. Free place in row – export of the last record in the row.
3. Insert new flow – insertion of a new element occurs to position i, a configuration parameter. i belongs to [0,s-1], where s is the length of the row.
4. Enhance flow – when the record on position x got hit, where x belongs to [0,s-1], the flow is moved to position $v[x]$, where v is an array of size s, defining a new position for hit flow as a value of v at the original flow position.

We aim to find methods to adapt parameters i and v.

As we can notice, the set of valid combinations of those parameters is huge. The total count of combinations is s^{s+1}, which is $16^{17} = 295\,147\,905\,179\,352\,825\,856$ for the default row length. Assuming that evaluation of every combination takes at least a minute, simple brute force of all possible values can't be done.

To avoid brute force, Žadnik [28] proposed and formally proved a few optimizations to reduce solution space:

▶ **Definition 3.1.** *Move tuple - is a tuple of* $\{t, c, i\}$*, where:*

- *t means the base target position of the enhanced flow.*
- *c means a count of flows in the tuple and belongs to* $[1, s]$.
- *i is an "increment" boolean flag that determines how the final place of the flow after enhancement is calculated. If i is false, every flow from the tuple gets position t after enhancement. Let the flow that gets enhancement is on the xth position from the first flow of its tuple, and i is set to true, then the new position of the flow is $t + x$.*

1. The flow position after enhancement can't be closer or at the same distance to the end of the row than before the enhancement, i.e., $v[x] < x$ (the only exception is the zero record of the row, which stays at its position after enhancement).
2. The flow can't "jump over" the other flow – if the flow on position x gets enhancement to the position $v[x]$, every flow at the position y, where $0 \leq y < x$, i.e., every flow, which is closer to the row beginning than x, has $v[y] \leq v[x]$, i.e. gets better (closer to the beginning) or same position after enhancement as $v[x]$.
3. Instead of determining optimal values of v for each place in the row, we group up part of the row as the "move tuple" (Definition 3.1). The first tuple can't set the increment flag to true, as it breaks the first optimization. We divided the row into $\lceil \frac{row\ length}{4} \rceil$ tuples. For example, if the row length is 8, it can be written by tuples as [{0, 3, false},{1, 5, true}]. This shortened policy form can be written down to the complete form, displayed in Figure 3.2, which graphically displays the enhanced position for every position.

The shortened form of the row significantly decreases solution space, and every flow in the shortened form still fulfills the optimizations.

To find exact parameter values, the authors of the original work proposed using a genetic algorithm.

A genetic algorithm is a method of searching a state space in optimization inspired by natural selection. The algorithm works with generations; every generation consists of possible solutions to the problem, and every solution is presented by its chromosome. In our case, the chromosome of a solution is the exact value of the parameters. The genetic algorithm defines operations in a generation:

1. Mutation – random changes in the chromosome of the solution, i.e., random change of the value of some parameter in configuration.
2. Crossover – an exchange of parts of chromosome between 2 solutions.

The genetic algorithm introduces a fitness function, which takes some solution and returns a number evaluating the quality of this solution to measure the impact of the genetic operations on the solution, and the objective function. As we want to minimize the objective function, we define it as the reciprocal of the fitness function. For our purposes, the objective function can be defined via the count of cache misses – the count of packets for which appropriate flow was not found and no empty place in a row was found or via the reciprocal of the count of cache hits. Both approaches create very similar solutions; we use the hit count as the fitness function.

The authors of the original work dedicated a significant part of the work to finding out what genetic operation can be used and what effect it brings. Results show that usage of crossover doesn't bring significant improvement to the searching process – with crossover, a bit better solution was generated in earlier stages of searching; at some count of generations, both algorithms, with crossover and without, converged to solutions with the same fitness function evaluation.

The final algorithm to find the best solution is described in Algorithm 7.

By comparing configurations with comparison operators, we mean a comparison of evaluations of the fitness function.

Creating a new generation is described in Algorithm 20, the binary tournament in Algorithm 19.

The main function that creates new configurations is an Algorithm 21, which consists of particular mutations and their fixes that are described in Algorithms 22-29.

We must also define a few functions responsible for generating random numbers in Table A.1.

Algorithm 7 Find best solution

Input: count of repeats *repeats*
Output: best found configuration *configuration*
generation = create_random_generation()
global_best_configuration = random_configuration()
for $0 \leq i <$ *repeats* **do**
  best_configuration = choose_best(*generation*)
  if *global_best_configuration* < *best_configuration* **then**
    global_best_configuration = *best_configuration*
  end if
  parent_configuration = binary_tournament(*generation*)
  generation = create_generation(*parent_configuration*)
end for
return *global_best_configuration*

Packet distance prediction

Besides this improvement, the idea is to improve the row management policy's performance by adding the prediction of when the next packet for this flow arrives. Intuitive solution seems to be analyzing the content of packet payloads. Practically, we can't rely on the packet's payload, as it is highly probable encrypted, or even if not, we must understand the scheme of payloads of many applications. Instead, we can rely on information that is always accessible, as it is needed to route the packet through the network – IP packet and TCP/UDP headers. From the study [28], the most information about the arrival of the next packet brings these fields:

1. IP protocol field.
2. IP total length field.
3. TCP flags.
4. TCP window size.

Even if we can find patterns in the arrival times of packets by observing the properties described above, we must display them numerically to use later in the cache.

The original Žádník [28] work used distance measurement in packets, i.e., the distance between two packets belonging to the same flow was measured in a count of packets belonging to other flows accepted between these two packets. From our view, this measurement must not be ideal, as the same flow with the same distances between packets is classified differently in different networks depending on the intensity of other simultaneously active flows. This makes the predictor trained on the one traffic example unprecise on the other example.

An improvement for this problem seems to be a distance measurement in seconds. In this case, distances for all packets inside one flow are calculated independently of packets in other flows, making these predictions more usable for other traffic samples.

We analyze packet distances in a PCAP file to create the dataset. We use the Weka library and its implementation of the J48 classification tree to generate the classification code to make the classification code. The textual form of the tree is converted to C code with the script. The output of the script is a bunch of nested if statements, i.e., has constant time complexity.

After our classification tree is converted to the code, a question arises about using the packet classifications. Packets classified as having a short distance to the next packet should be closer to the beginning of the row. On the contrary, packets with long distances to the next packet should be placed closer to the end of the row, reducing the probability of throwing out the records that will be accessed soon. But depending on its classification, we don't know the exact number of positions to promote the flow. To address this problem, the original work's author proposed extending the adaptive policy with new parameters – offsets – to move every flow after enhancement for every packet classification. Every offset is a whole number from $[\frac{line\_size}{2}, -\frac{line\_size}{2}]$ after the flow was enhanced to its position offset is also added, with restrictions that a new position can't be out of the current row. If it is negative, it can't leave the flow at the original position or even move it to a worse position. If, after adding the offset to enhanced flow, its new index is the same or worse, a new position after enhancement is chosen next to the current position.

3.4.6 Simulated annealing with taboo list

After analysis of the original policy search algorithm [28] described in subsection 3.4.5, we would like to introduce and test a new way of searching for best configurations. We test the usage of simulated annealing with taboo list optimization techniques.

We need to introduce a new definition to describe the following algorithm – the distance between 2 configurations. Its value expresses how much two configurations of the same row

Algorithm 8 calculate_distance_between_configurations

Input: the first configuration *left*, the second configuration *right*
Output: integer numerically describing distance between configurations
$distance = 0$
for $0 \leq tuple\_index < left.tuple\_count()$ **do**
  $distance$ += $|left.count - right.count| + |left.target - right.target|$
  if $left.increment \neq right.increment$ **then**
    $distance$ += 1
  end if
end for
$distance$ += $|left.offset\_of\_short - right.offset\_of\_short|$
$distance$ += $|left.offset\_of\_medium - right.offset\_of\_medium|$
$distance$ += $|left.offset\_of\_long - right.offset\_of\_long|$
$distance$ += $|left.offset\_of\_never - right.offset\_of\_never|$
$distance$ += $|left.insert\_position - right.insert\_position|$

differ. This value is calculated as described in Algorithm 8.
The core of the new algorithm is described in Algorithm 9 and consists of the following steps:

1. If we find a better solution, we simply take it.
2. If all configurations in the generation have worse statistics than a parent of this generation, we choose a random solution; the probability that the exact solution is chosen decreases with increasing distance between this solution and the parent. Probability also decreases with increasing generation number. We don't want to change the current solution significantly as the search ends. This step is described in Algorithm 10.

The second part of the improvement is a taboo list. Taboo list is a list of the last n configurations. The n was set to a generation size. When mutations of the best solution are generated, we also check that the newly generated solution is far enough from all solutions in the taboo list. This approach is included inside the function *create_generation_with_taboo_list* from Algorithm 11.

Algorithm 9 Simulated annealing

Input: count of repeats *repeats*
Output: best found configuration *configuration*
generation = create_random_generation()
global_best_configuration = random_configuration()
for $0 \leq current\_generation < repeats$ **do**
  best_configuration = choose_best(*generation*)
  if *global_best_configuration* < *best_configuration* **then**
    global_best_configuration = *best_configuration*
  end if
  chosen_configuration = choose(*generation*, *current_generation*, *chosen_configuration*)
  generation = create_generation_with_taboo_list(*chosen_configuration*)
end for

3.5 Multi thread optimization

The original implementation processes new packets and exports expired ones in the same thread. These tasks seem not directly connected so that we could divide them into separate threads.

Algorithm 10 choose

Input: generation to choose configuration from: *generation*, current generation number: *current_generation*, configuration from which *generation* was created: *parent_configuration*
Output: chosen configuration
sorted_generation = sort_by_efficiency(*generation*)
best_configuration = *sorted_generation*[0]
if *best_configuration* > *parent_configuration* **then**
  return *best_configuration*
end if
chosen_configuration = *best_configuration*
for $0 \leq i <$ *sorted_generation.size*() **do**
  configuration = *sorted_generation*[*i*]
  if random$(1 - e^{distance(configuration, best_configuration)/(6\cdot((current_generation+i)\cdot\frac{-3}{40}-2))})$ **then**
    chosen_configuration = *configuration*
  end if
end for
return *chosen_configuration*

Algorithm 11 create_generation_with_taboo_list

Input: configuration to create generation from *configuration*, taboo list *taboo_list*, generation number *current_generation*
Output: generation created from seed
output = *EMPTY_GENERATION*
while *output.size*() < *generation_size* **do**
  new_solution = *configuration*
  new_solution.mutate()
  output.add(new_solution)
  repeat
    repeat
      output.last_configuration().mutate()
      too_close = *false*
      for all *taboo_configuration* in *taboo_list* **do**
        too_close = distance(*output*.last_configuration(), *taboo_configuration*)<
        current_generation $\cdot -0.5 + 20$
        if *too_close* == *true* **then**
          break
        end if
      end for
    until *too_close* == *true*
  until NOT *output*.all_configurations_are_unique()
end while
return *output*

This approach has a problem when the export thread tries to export expired flows from the row currently manipulated by the main thread; the program's behavior is undefined. The standard solution for this problem is the usage of mutexes and locks to grant exclusive access to the memory. In this case, the main thread needs to gain the lock for every incoming packet, which means doing a system call with approximately hundreds of process cycles that would dramatically decrease the performance of the cache.

Instead, we use atomic variables with spin-waiting to grant exclusive access. The cache keeps variables containing currently processed lines by export and main threads. This variable is modified only atomically.

As the original implementation checked for the expiration of the flows after every newly created flow, we can expect fewer expired flows to be exported, increasing the count of cases where the cache decides that there is no empty place in the row. To address this problem, we improve the searching empty place algorithm to check if flows have expired and return that place after the export. Our task is determining how often the export thread must call the export function to achieve acceptable results.

3.6 Flood detection

In all previous sections, we discussed only the optimizations focused on the traffic generated by real users and their services; we expect this traffic to act according to typical user behavior. Of course, it is not the only type of traffic on the network. In this section, we want to address the problem of flood attacks on the network. From our point of view, the most important part is the influence of such an attack on our cache. A flood attack is a denial of service attack, which sends massive traffic to a particular server or service to exhaust its resources [10]. I.e., the definition itself doesn't define the exact way of carrying out an attack. Suppose we focus only on our cache, ignoring all other network parts. In that case, flood attacks with a low amount of simultaneously active flows, even bringing big payloads, don't affect the cache as it doesn't rely on payloads. On the other hand, a flood attack can consist of a high amount of flow, which is typical for TCP-SYN DDoS attacks. Such attacks create many new short flows consisting of few packets. We expect that the count of malicious flows significantly surpasses the cache size, and the cache tries to keep them all by exporting the regular user flows. Assuming that users will continue communicating during the attack, their export records will be divided into many short pieces, making their analysis impossible.

To react to the attack, we must be able to determine the start of the attack as soon as possible. To do it, we use an approach based on the measurements of flow exports per second.

Hofstede [13] proposed a system to detect a flood attack by monitoring sharp increases in the number of exported flows using exponentially weighted moving average extended by thresholds and a cumulative sum. Algorithm description:

1. $\overline{x}_t = \alpha \cdot x_t + (1 - \alpha) \cdot \overline{x}_{t-1}$, where $\overline{x}_t$ is weighted average at time t, $\overline{x}_{t-1}$ is weighted average calculated for previous time interval $t-1$, α is tunable parameter, describing how previous value is discarded. We forecast value $\overline{x}_{t+1}$ to be the same as $\overline{x}_t$.

2. After the actual value of x_{t+1} is known, we calculate the forecasting error as $e_{t+1} = x_{t+1} - \overline{x}_{t+1}$.

3. $\sigma_{e,t+1}$ is a standard deviation of previous forecasting errors, which is calculated as $\sqrt{\frac{\sum_{i=1}^{t} {e_i}^2 - 2 \cdot \frac{(\sum_{i=1}^{t} e_i)^2}{t} + (\frac{\sum_{i=1}^{t} e_i}{t})^2}{t}}$.

4. Then we calculate the threshold value for time $t+1$ as $T_{t+1} = \overline{x}_{t+1} + \max(c_{thr} \cdot \sigma_{e,t+1}, M_{min})$, where c_{thr} and M_{min} are defined constants. M_{min} is used to cancel flood detection on stages

of low cache loads when a significant increase of active flow count in relative numbers, but small in absolute, occurs.

5. We need to keep the sum of all differences between actual measurements and appropriate thresholds. This sum at time t is defined as $S_t = \max(S_{t-1} + (x_t - T_t), 0)$.

6. The flood is detected when S_t exceeds the threshold $T_{cusum,t}$ at time t, where $T_{cusum,t} = c_{cusum} \cdot \sigma_{e,t}$.

Calculating all required variables for time t is a relatively expensive operation, so the time interval can't be very short – a good solution seems to be 5 seconds. The c_{thr} was set to 5, M_{min} to 7000, α to 0.3.

The steps above are presented algorithmically in Algorithm 12.

Algorithm 12 flood_detected

Input: flood measurement data *data*
Output: true, if the flood attack detected, false otherwise
$data.measurement\_count$ += 1
$treshold = data.last\_mean + \max(data.threshold \cdot data.deviation, data.minimum)$
$data.cusum = max(data.cusum + data.cache\_misses\_in\_last\_interval - threshold, 0)$
$cusum\_threshold = data.cusum\_threshold \cdot data.deviation$
$data.last\_mean = \alpha \cdot data.cache\_misses\_in\_last\_interval + (1 - \alpha) \cdot data.last\_mean$
$forecasting\_error = data.cache\_misses\_in\_last\_interval - data.last\_mean$
$data.error\_sum$ += $forecasting\_error$
$data.error\_sum\_square$ += $forecasting\_error^2$
$error\_mean = \frac{data.error\_sum}{data.measurement\_count}$
$data.deviation = \sqrt{\frac{data.error\_sum\_square - 2 \cdot error\_mean \cdot data.error\_sum}{data.measurement\_count}}$
return $data.cusum > cusum\_threshold$
//Calculation is called every predefined time interval, 5 seconds in our case.

Chapter 4
Test results

4.1 PCAPs

After the new feature is added, the impact must be measured. The best solution is to use the traffic record, as the flow exporter works deterministically, returning the same output for the same input (with deviations in time), allowing us to measure the consequences of the added feature precisely.

We use PCAP files to measure the impact. PCAP file stores exact copies of the packets. If we test our feature on the PCAP file, we don't need to wait for time intervals between packets; we insert the next packet right after the previous one, which decreases test time. In our tests, we use these PCAP files:

1. Traffic – Is an evening traffic from backbone link of CESNET from 2023-06-05 21:15:32 to 2023-06-05 21:15:40. Length – 8 seconds, average inter-packet distance – 0.01sec. Content classification by protocols is in table 4.2. Graph of new flows per second can be found in Figure 4.1, packets per second in Figure 4.2.
2. Sh – Strahov dormitory, Prague, traffic of accommodated, the afternoon of Monday, 13.11.2023. Length – 1707s, average inter-packet distance – 0.34sec. Content classification by protocols is in Table 4.3. Graph of new flows per second can be found in figure 4.3, packets per second in Figure 4.4.
3. Tul - the Technical University of Liberec, normal high school traffic during the afternoon of 26.2.2024. Length – 332s, average inter-packet distance – 0.3sec. Content classification by protocols is in Table 4.4. Graph of new flows per second can be found in Figure 4.5, packets per second in Figure 4.6.
4. Mawi - is a PCAP sample from the Mawi archive. It describes the packets passing the Mawi trans-Pacific link on 2010/04/14 from 14:00 to 14:15. Inter-packet distance is 1,2sec. Content classification by protocols is in Table 4.5. Graph of new flows per second can be found in Figure 4.7, packets per second in Figure 4.8.
5. Mawi_flood - is a PCAP sample from the Mawi archive. It describes an interval of 2010/08/30 from 7:00 to 7:15. This record was labeled as containing anomalous traffic; we can observe a sharp increase in the count of active flows in a few seconds, which is highly likely marks the attack. Average inter-packet distance – 0.01sec. Content classification by protocols is in Table 4.6. Graph of new flows per second can be found in Figure 4.9, packets per second in Figure 4.10.

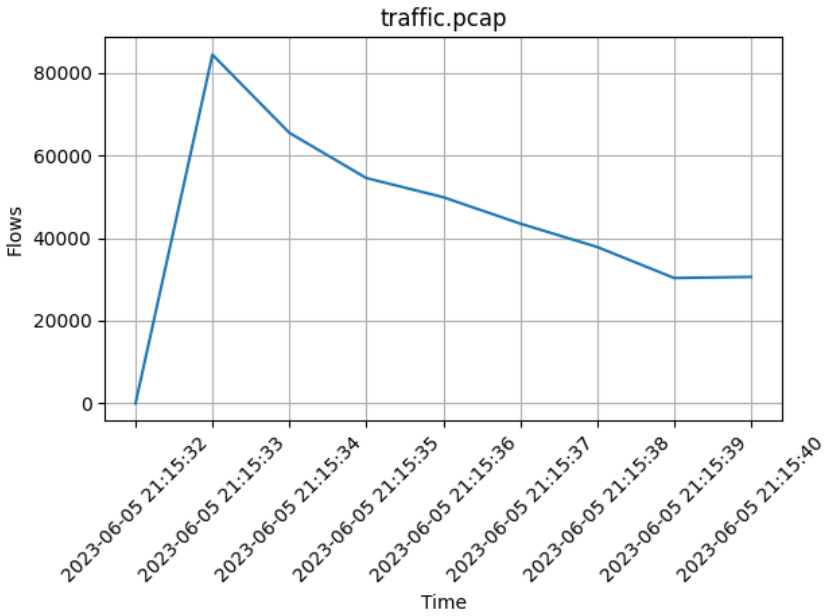

Figure 4.1 Count of new flows per second in Traffic PCAP

The results of the tests are described by labels defined in Table 4.1.

4.2 Double hashing

To test the impact of the flow sorting described in section 3.2 we use two configurations, both using LRU row policy and the XXH64 hash function, but one does hashing twice, and the second one sorts its flows, hashing only once.
Results are presented in Tables 4.7-4.10.

We can observe that cache statistics are almost unchanged, with the only exception being time. This improvement brings speedup 7.5%-19%. We can admit this improvement as successful.

4.3 Hash function

We use an improved version of hashing to test the hash functions, as described in the previous section. Result are in Tables 4.11-4.15.

The count of cache hits by Toeplitz hash for Traffic PCAP (Table 4.11) is anomaly high. A subsequent test, where the comparison of flows by hash value was replaced with strict comparison by key fields, showed that Toeplitz hash creates many collisions (Table 4.12).

XXH3_64bits is the fastest hash function that surpasses other hash functions. Statistics are almost the same for all hash functions. The hash function of the cache is replaced with XXH3_64bits.

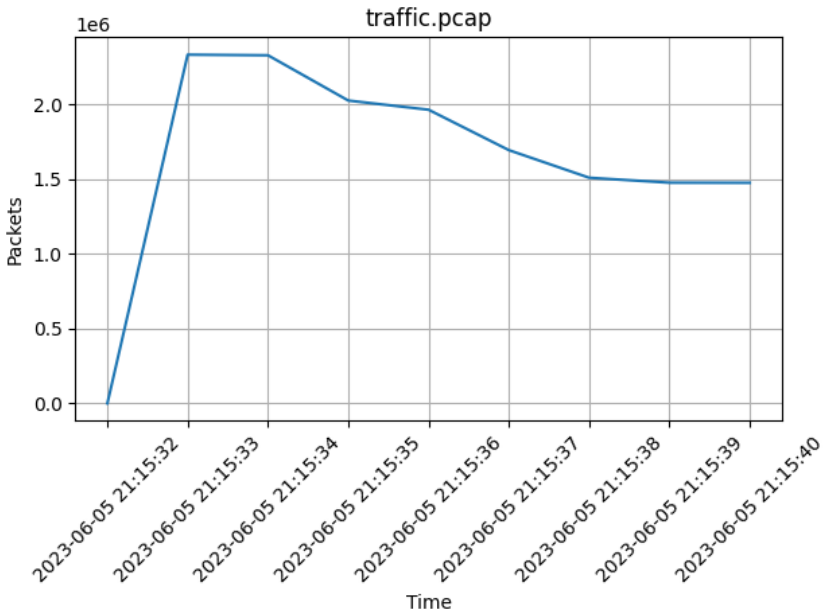

Figure 4.2 Count of new packets per second in Traffic PCAP

4.4 Row policy

In this section, we test different row policies. To hash flows, we use XXH3_64bits with sorted flows.

The original LRU policy is used as a reference. Results are presented in Tables 4.16-4.19.

Seeing LRU implementation with a heap slower than the original LRU is unexpected. Analyzing the cache runs with the profiler in Figure 1.4, we found that almost half of the cache's work time is spent searching for the flow in the row. The character of heap implementation means that the flow with the lowest last access time is on the top of the heap, while the most active flows are randomly shuffled on the lower half of the row. In the default implementation, flows in the row are sorted by their last access time, while heap LRU disrupts the rule of locality. This effect is revealed by increasing the average lookup and lookup variance statistics.

According to the original LIRS paper, the "list" part must be tiny, approximately 1% of managed space. In our case, it is impossible to set that small value, as from 16 cells in a row, we can set the list to be minimal one cell long, or 6.25%. Setting the list part size to 0 degrades the LIRS to default LRU. Increasing the list part size to 2 makes statistics worse, so the size of 1 seems to be the best.

The 2Q simplified algorithm shows promising results. The division of the row to proportion 1:3 has the worst hits statistics while having the best work time, average lookup time, and lookup variance compared to other 2Q-simplified policies. The 1:1 and 3:1 options show a difference in cache hits of less than 0.1%; for many types of traffic, 2Q is faster than the default LRU, except for the Traffic PCAP file. The best proportions can't be chosen, as it depends only on the type of traffic, which makes this policy unsuitable for the general-purpose flow exporters, as we can't know the traffic where the exporter is installed.

The full version of the 2Q algorithm generally can't overcome the simplified version, having time statistics higher and cache hit statistics worse. We can notice that cache hit statistics are almost the same for all presented proportions. The flow-driven replacement policy in this test uses an inefficient implementation, so the time is not essential for us, but statistics: the policy

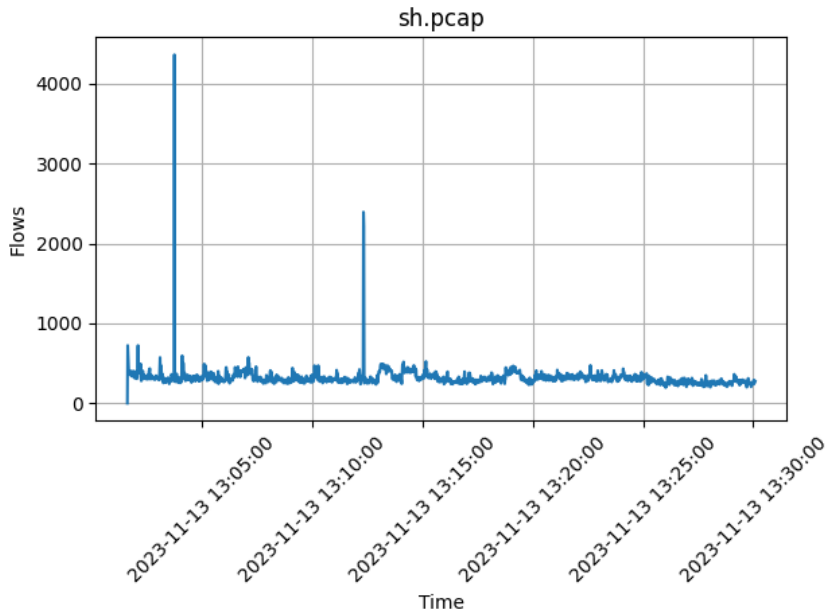

Figure 4.3 Count of new flows per second in Sh PCAP

can't achieve the hit rate of the LRU, showing a slight decrease of cache hits. Unexpectedly, it overcomes the LRU on the Mawi PCAP file and gains more hits.

The default LRU policy looks pretty efficient, combining a good hit ratio and run time, compared to other policies for all provided traffic samples. We don't see enough reasons to replace the LRU policy with another policy.

4.4.1 Adaptive row policy

This subsection is dedicated to the comparison of the results of the configurations found in the genetic algorithm and simulated annealing with the taboo list. To create the configurations, the search algorithm created 80 generations with eight configurations each, and the first generation was set to have more configurations – 16. Practically, improvements after the 60th generation are minor; increasing the generation count to 200 doesn't bring any improvement. For many PCAP files, such as Mawi and Sh, the count of cache misses is too low to test adaptive policies, so the cache size for these records was reduced to 2^{11} (default cache size used for other cases is 2^{17}).

The main problem with searching the configuration is the required time. For our PCAP files, creating and evaluating 80 generations requires 4-12 hours. According to the work of Žadník [28], sampling could be used to reduce evaluation time. We tested reducing PCAP files to deterministically every 2nd and every 3rd packet: as we can see in the result tables, creating generation on original PCAP records makes configuration pretty close to the original LRU, while configurations created on sampled PCAPs may show better results on sampled PCAP, but worse on the original record. By reducing PCAP to every 2nd packet, we managed to generate relatively successful configurations for some PCAP files, while by reducing input to every 3rd packet, we didn't find any successful configuration. To create our configurations, we do not use sampled PCAPs.

The XXH3_64bits (section 4.3) with sorted flows (section 3.2) were used to reduce the evaluation time of generations. We present results comparing found configurations for the genetic

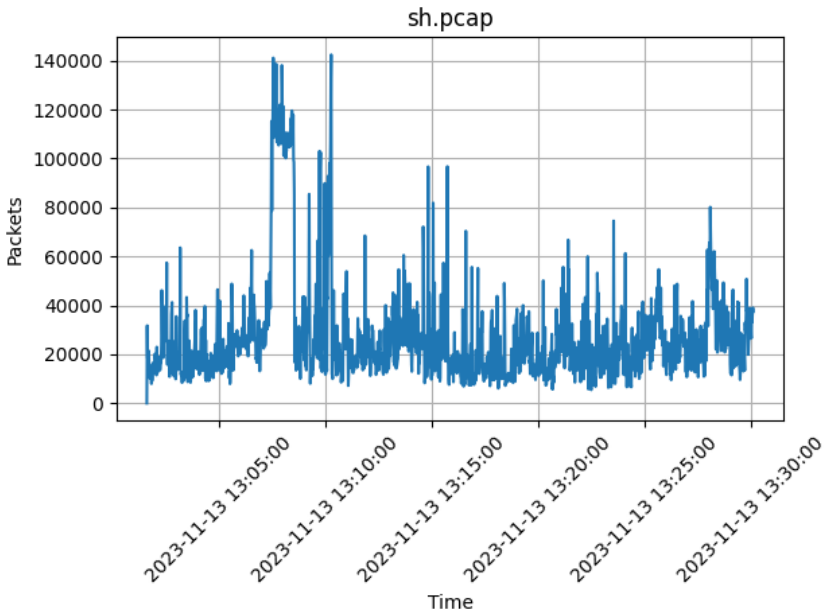

Figure 4.4 Count of new packets per second in Sh PCAP

algorithm and simulated annealing in Table 4.20-4.23.

We can notice that for the majority of PCAP files, adaptive policies brought a slowdown. The genetic algorithm or simulated annealing configurations itself cannot slow down the cache in runtime, as their impact on the run is few memory and arithmetic operations for every packet. The source of the slowdown is the distance predictor for PCAP files, where found configurations did not significantly differ from the original LRU configuration, and the increase of cache hits did not pay off the price of prediction for every packet.

Algorithms found the configurations with better hit/miss statistics for all PCAP files, while the significant improvement is shown only on Mawi PCAP (Table 4.23).

Every presented configuration for the genetic algorithm and simulated annealing is the best of the five generated configurations. Still, all configurations differed only slightly, and to achieve almost the same results, we could create only one configuration for every PCAP file. Also, we can notice almost no difference between the results of configurations provided by the genetic algorithm and those of simulated annealing with taboo list.

4.5 Multi thread optimization

In this section, we use configuration with the default LRU, using XXH3_64bits hash function to hash sorted flows. Results of the comparison are presented in Table 4.24.

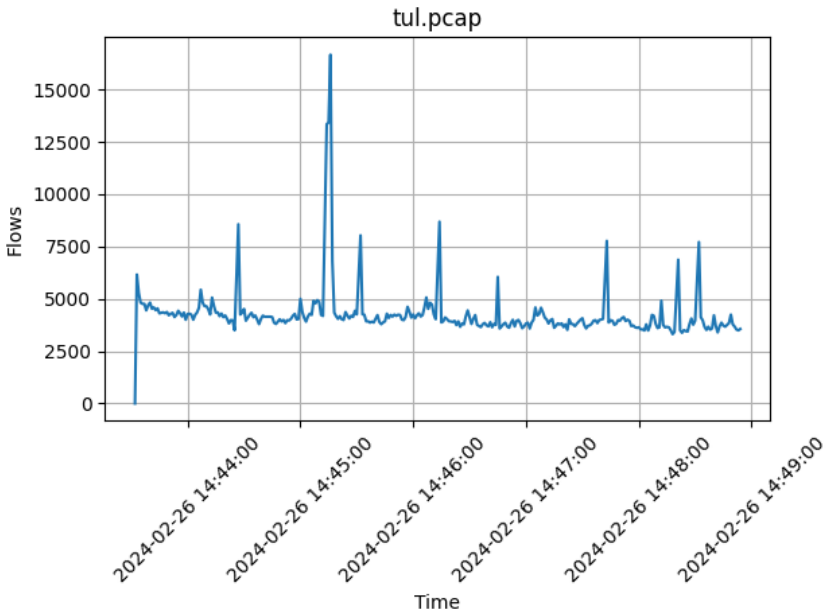

Figure 4.5 Count of new flows per second in Tul PCAP

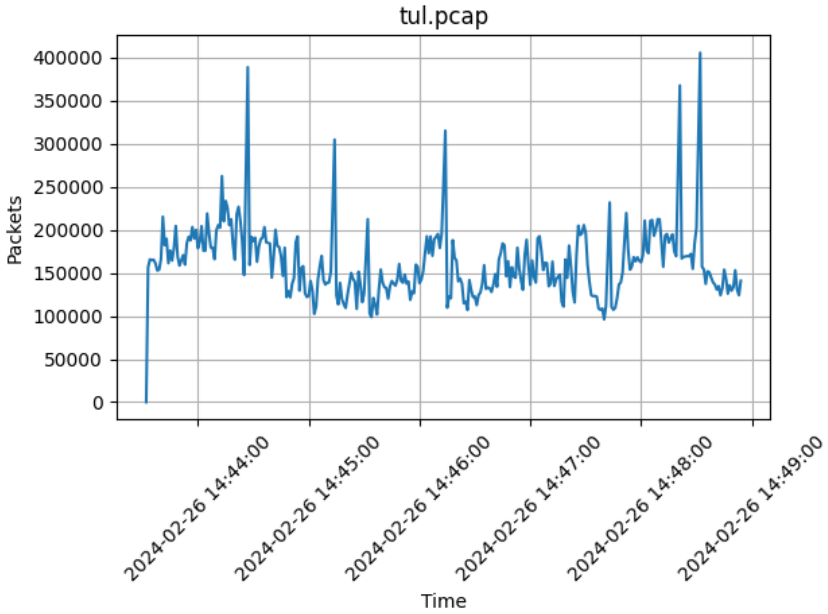

Figure 4.6 Count of new packets per second in Tul PCAP

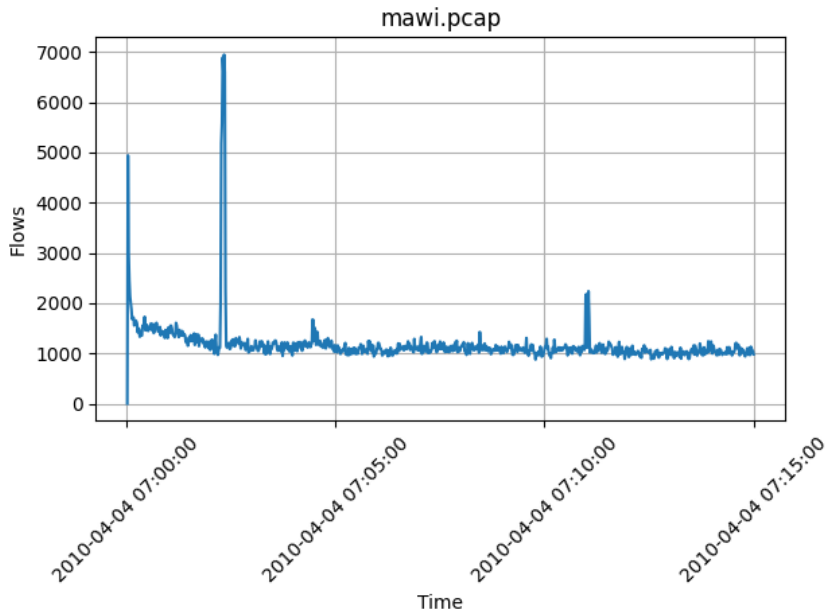

Figure 4.7 Count of new flows per second in Mawi PCAP

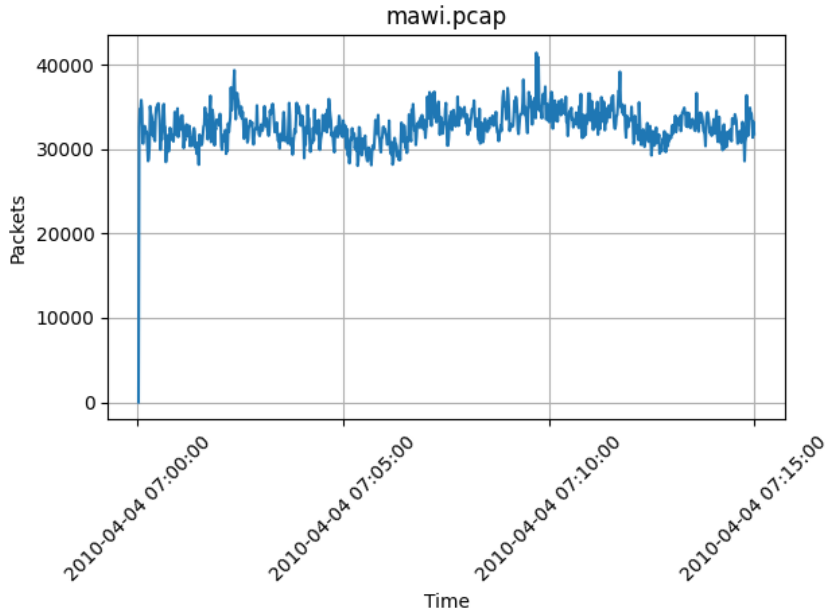

Figure 4.8 Count of new packets per second in Mawi PCAP

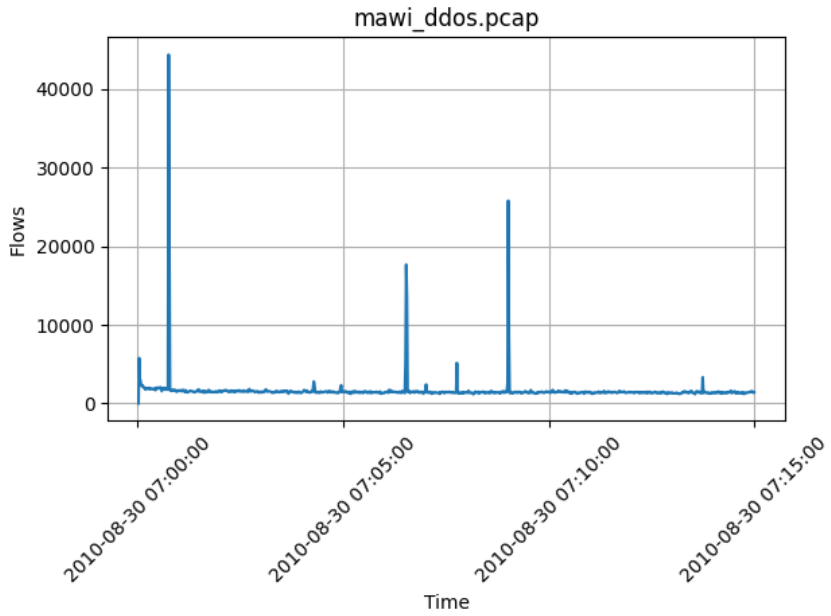

Figure 4.9 Count of new flows per second in Mawi_flood PCAP

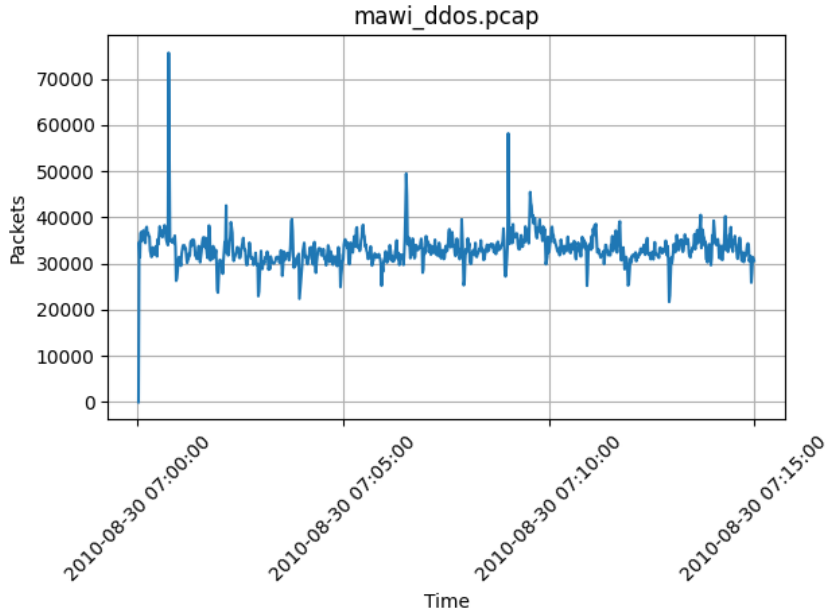

Figure 4.10 Count of new packets per second in Mawi_flood PCAP

Table 4.1 Labels used to compare cache results

Hits – count of packets where the proper flow was successfully found in the cache.
Empty – count of packets, where proper flow was not found, but the row had an empty place, and we didn't need to export any flow to place new flow to the row.
Not empty – count of packets where the proper flow was not found, and the row had no empty place, and we needed to export flow to place new flow to the row.
Exported – overall count of exported flows, including exports of expired flows or due to lack of empty place in a row.
Avg. lookup – average count of checked cells in a row, before the proper flow was found. Unsuccessful searches are not counted.
Var. lookup – variance of the average lookup.
Periodic – count flow flows exported in periodical exports of function *export_expired*().
On finish – count of flows, including active and expired, that was in the cache when the exit signal was accepted.
Time – average count of seconds spent in the cache. Packet processing and exporting time included. Packet reading from input and writing to output excluded. Count of runs to average – 5.

Table 4.2 Traffic PCAP statistics

Protocol	Flows	Packets	Bytes
TCP	281 305	10 134 741	9 466 938 155
TCP (%)	69.6	66.6	69.6
UDP	112 510	4 977 507	4 129 882 362
UDP (%)	27.8	32.7	30.3
ICMP	5 323	12 236	912 928
ICMP (%)	1.12	0.08	0.003
DNS	54 456	91 249	15 925 411
DNS (%)	13.4	0.6	0.1
SSL/TLS	83 687	6 890 495	6 622 891 749
SSL/TLS (%)	20.7	45.3	48.7
Total	403 972	15 204 703	13 598 309 979

Table 4.3 Sh PCAP statistics

Protocol	Flows	Packets	Bytes
TCP	398 216	40 837 004	44 507 978 384
TCP (%)	72.8	87.6	93.7
UDP	121 469	5 505 412	2 977 366 682
UDP (%)	22.2	11.8	6.2
ICMP	27 024	217 643	4 018 179
ICMP (%)	4.9	0.4	0.008
DNS	70 600	138 130	14 115 324
DNS (%)	12.9	0.2	0.02
SSL/TLS	13 185	29 257 645	33 026 567 520
SSL/TLS (%)	2.4	62.8	69.5
Total	546 772	46 567 434	47 489 385 503

Table 4.4 Tul PCAP statistics

Protocol	Flows	Packets	Bytes
TCP	1 035 007	41 178 500	1 008 741 494
TCP (%)	77.4	81.09	93.1
UDP	256 602	8 780 681	70 639 299
UDP (%)	19.1	17.2	6.5
ICMP	43 538	129 258	3 048 034
ICMP (%)	3.2	0.2	0.2
DNS	468	468	53 331
DNS (%)	0.03	0.0004	0.002
SSL/TLS	15	381	35 467
SSL/TLS (%)	0.0014	0.0007	0.003
Total	1 337 211	50 781 008	1 082 600 789

Table 4.5 Mawi PCAP statistics

Protocol	Flows	Packets	Bytes
TCP	508 725	15 841 821	410 695 428
TCP (%)	47.9	53.6	78.4
UDP	480 836	12 809 017	107 885 132
UDP (%)	45.3	43.3	20.6
ICMP	10 670	73 254	492 764
ICMP (%)	1.005	0.2	0.08
DNS	261 934	451 080	9 021 576
DNS (%)	24.6	1.5	18.8
Total	1 061 040	29 535 029	523 396 592

Table 4.6 Mawi_flood PCAP statistics

Protocol	Flows	Packets	Bytes
TCP	664 922	21 719 710	571 003 272
TCP (%)	45.3	72.9	88.4
UDP	712 037	7 075 196	68 891 392
UDP (%)	48.6	23.7	10.6
ICMP	18 069	240 589	1 011 196
ICMP (%)	1.2	0.8	0.15
DNS	465 360	1 024 139	20 482 756
DNS (%)	31.7	3.4	3.1
Total	1 464 887	29 790 632	645 897 256

Table 4.7 Comparison of flow hashing with and without sorting on Traffic PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
Unsorted	14 719 870	131 083	353 761	484 833	1.21	1.18	5.9
Sorted	14 719 801	131 083	353 830	484 902	1.21	1.18	5.3

Table 4.8 Comparison of flow hashing with and without sorting on Sh PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
Unsorted	45 852 498	719 233	0	719 190	1.0	0.01	12.4
Sorted	45 852 452	719 235	0	719 192	1.0	0.01	11.5

Table 4.9 Comparison of flow hashing with and without sorting on Tul PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
Unsorted	48 854 366	766 063	1 163 324	1 929 336	1.09	0.75	19.7
Sorted	48 854 559	767 514	1 163 700	1 929 123	1.09	0.75	16.5

Table 4.10 Comparison of flow hashing with and without sorting on Mawi PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
Unsorted	28 285 118	1 248 250	1476	1 247 388	1.22	0.99	11.4
Sorted	28 285 141	1 248 132	1597	1 247 391	1.22	0.99	10.0

Table 4.11 Comparison of different hash functions on Traffic PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
XXH64	14 719 801	131 083	353 830	484 902	1.21	1.18	5.3
Toeplitz	14 755 882	131 749	317 749	448 821	1.2	1.12	5.9
CRC32c	14 719 537	131 083	354 094	485 166	1.21	1.18	6.3
XXH3_64bits	14 719 855	131 084	353 776	484 848	1.21	1.18	4.6
XXH3_128bits	14 719 810	131 083	353 821	484 893	1.21	1.18	4.8
MurMurHash	14 719 673	131 156	353 958	485 030	1.21	1.18	6.9
FarmHash	14 719 742	131 154	353 889	484 961	1.21	1.18	5.6
SuperFastHash	14 719 792	131 157	353 839	484 911	1.21	1.19	6.9

Table 4.12 Results of ipfixprobe run on Traffic PCAP with strict flow comparison by key fields

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup
Toeplitz hash with strict comparison	14 715 010	131 083	358 621	489 693	1.27	1.51

Table 4.13 Comparison of different hash functions on Sh PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
XXH64	45 852 452	719 235	0	719 192	1.0	0.01	11.5
Toeplitz	45 852 499	719 387	0	719 188	1.0	0.01	17.1
CRC32c	45 852 542	719 235	0	719 192	1.0	0.01	14.3
XXH3_64bits	45 852 482	719 235	0	719 192	1.0	0.01	9.3
XXH3_128bits	45 852 468	719 235	0	719 192	1.0	0.01	9.5
MurMurHash	45 852 497	719 391	0	719 192	1.0	0.01	15.8
FarmHash	45 852 481	719 391	0	719 192	1.0	0.01	13.0
SuperFastHash	45 852 512	719 386	0	719 187	1.0	0.01	14.7

Table 4.14 Comparison of different hash functions on Tul PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
XXH64	48 854 559	767 514	1 163 700	1 929 123	1.09	0.75	16.5
Toeplitz	48 854 686	765 422	1 163 684	1 929 055	1.09	0.75	18.9
CRC32c	48 854 470	765 862	1 163 410	1 929 220	1.09	0.75	19.1
XXH3_64bits	48 854 637	765 809	1 163 314	1 929 072	1.09	0.75	12.8
XXH3_128bits	48 854 331	765 739	1 163 638	1 929 327	1.09	0.75	14.0
MurMurHash	48 854 325	766 650	1 164 811	1 929 373	1.09	0.75	19.1
FarmHash	48 854 369	767 478	1 163 936	1 929 323	1.09	0.75	15.9
SuperFastHash	48 854 385	768 285	1 163 094	1 929 284	1.09	0.75	17.8

Table 4.15 Comparison of different hash functions on Mawi PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
XXH64	28 285 141	1 248 132	1597	1 247 391	1.22	0.99	10.0
Toeplitz	28 295 152	1 240 217	900	1 237 305	1.22	0.98	11.0
CRC32c	28 285 146	1 248 712	986	1 247 358	1.22	0.99	11.1
XXH3_64bits	28 285 051	1 248 091	1652	1 247 405	1.22	0.99	8,6
XXH3_128bits	28 285 079	1 248 086	1640	1 247 388	1.22	0.99	8,9
MurMurHash	28 285 128	1 256 681	1587	1 247 380	1.22	0.99	11.8
FarmHash	28 285 128	1 256 632	1664	1 247 408	1.22	0.99	9.8
SuperFastHash	28 285 105	1 256 595	1683	1 247 383	1.22	0.99	12.7

Table 4.16 Comparison of different row policies on Traffic PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
LRU	14 719 801	131 083	353 830	484 902	1.21	1.18	5.3
LRU - heap	14 712 720	131 154	360 911	491 983	3.23	5.5	5.9
LIRS(1:15)	14 499 773	128 617	576 559	708 915	1.8	0.6	7.4
LRU - 2Q - simplified(1:1)	14 710 690	268 711	225 403	494 013	1.24	1.5	5.0
LRU - 2Q - simplified(1:3)	14 666 492	331 365	206 905	538 211	1.2	1.1	4.8
LRU - 2Q - simplified(3:1)	14 718 996	250 398	235 406	485 707	1.26	2.0	5.2
LRU - 2Q - full(5:5:6)	14 713 047	168 952	322 785	498 812	7.6	19.5	5.9
LRU - 2Q - full(2:1:1)	14 706 709	165 555	332 521	504 330	8.2	27.5	5.9
LRU - 2Q - full(1:2:1)	14 703 743	198 062	302 970	508 904	8.8	29.5	5.3
LRU - 2Q - full(1:1:2)	14 713 951	165 648	325 186	497 420	6.5	12.1	5.8
Flow-driven	14 652 373	131 458	425 144	556 216	6.0	24.9	11.8

Table 4.17 Comparison of different row policies on Sh PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
LRU	45 852 452	719 235	0	719 192	1.0	0.01	11.5
LRU - heap	45 852 452	719 391	0	719 192	2.2	1.5	15.8
LIRS(1:15)	45 851 396	720 710	0	918 242	1.8	0.12	20.7
LRU - 2Q - simplified(1:1)	45 860 512	715 148	0	712 858	1.06	0.92	13.4
LRU - 2Q - simplified(1:3)	45 862 959	715 024	0	712 720	1.06	0.92	12.4
LRU - 2Q - simplified(3:1)	45 859 343	715 299	0	713 251	1.06	0.92	14.4
LRU - 2Q - full(5:5:6)	45 852 459	719 387	0	723 123	2.2	2.4	14.2
LRU - 2Q - full(2:1:1)	45 852 452	719 391	0	719 255	2.2	1.5	13.3
LRU - 2Q - full(1:2:1)	45 852 503	719 376	0	730 758	2.5	5.9	12,7
LRU - 2Q - full(1:1:2)	45 852 503	719 376	0	730 728	2.3	3.0	14.2
Flow-driven	45 848 909	726 924	0	731 882	2.0	1.5	24.5

Table 4.18 Comparison of different row policies on Tul PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
LRU	48 854 559	767 514	1 163 700	1 929 123	1.09	0.75	16.5
LRU - heap	48 839 271	719 391	1 688 819	1 942 761	3.3	6.0	19.4
LIRS(1:15)	48 768 200	1 158 234	865 095	2 030 254	1.8	0.6	22.9
LRU - 2Q - simplified(1:1)	48 853 095	939 740	994 757	1 929 930	1.09	0.77	14.8
LRU - 2Q - simplified(1:3)	48 814 557	941 892	1 028 081	1 966 928	1.07	0.55	14.7
LRU - 2Q - simplified(3:1)	48 848 316	1 100 131	839 811	1 935 663	1.11	1.15	16.6
LRU - 2Q - full(5:5:6)	48 849 066	708 100	1 228 684	1 940 834	8.3	17.5	18.7
LRU - 2Q - full(2:1:1)	48 843 505	636 025	1 305 556	1 945 230	9.1	26.1	17.9
LRU - 2Q - full(1:2:1)	48 845 101	673 763	1 266 157	1 945 283	9.8	25.9	16.8
LRU - 2Q - full(1:1:2)	48 847 310	794 650	1 144 004	1 941 877	7.13	10.6	18.1
Flow-driven	48 808 366	1 028 793	979 030	2 006 332	6.2	18.2	32.4

Table 4.19 Comparison of different row policies on Mawi PCAP

Hashing	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
LRU	28 285 141	1 248 132	1597	1 247 391	1.22	0.99	10.0
LRU - heap	28 285 218	1 255 763	2438	1 247 313	2.6	4.7	10.7
LIRS(1:15)	28 281 284	1 263 536	2888	1 380 331	2.1	0.85	14.2
LRU - 2Q - simplified(1:1)	28 318 367	1 252 540	964	1 213 247	1.5	5.3	9.8
LRU - 2Q - simplified(1:3)	28 257 153	1 298 764	350	1 276 214	1.4	5.3	9.0
LRU - 2Q - simplified(3:1)	28 316 730	1 250 889	1234	1 214 387	1.5	5.3	9.5
LRU - 2Q - full(5:5:6)	28 298 868	1 252 215	1538	1 304 653	3.8	14.7	9.8
LRU - 2Q - full(2:1:1)	28 287 860	1 255 539	1622	1 287 123	3.3	10.8	9.7
LRU - 2Q - full(1:2:1)	28 294 863	1 256 626	968	1 306 702	4.5	24.7	10.8
LRU - 2Q - full(1:1:2)	28 304 334	1 250 511	1603	1 296 226	3.6	11.6	10.0
Flow-driven	28 353 325	1 354 645	2355	1 357 378	3.1	5.4	20.1

Table 4.20 Comparison of configurations found by genetic algorithm and simulated annealing against the default LRU policy on Traffic PCAP

Algorithm	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
Genetic algorithm	14 720 426	131 154	353 205	484 277	1.23	1.46	4.9
Simulated annealing	14 720 434	131 154	353 197	484 269	1.23	1.46	4.9
Original LRU	14 719 855	131 084	353 776	484 848	1.21	1.18	4.6

Table 4.21 Comparison of configurations found by genetic algorithm and simulated annealing against the default LRU policy on Sh PCAP with cache size reduced to 2^{11}

Algorithm	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
Genetic algorithm	45 735 221	3327	828 988	832 135	1.1	0.67	9.8
Simulated annealing	45 735 031	3092	829 413	832 324	1.1	0.66	9.8
Original LRU	45 727 525	2636	837 348	839 807	1.1	0.59	9.3

Table 4.22 Comparison of configurations found by genetic algorithm and simulated annealing against the default LRU policy on Tul PCAP

Algorithm	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
Genetic algorithm	48 860 695	727 291	1 202 999	1 927 393	1.4	1.19	13.5
Simulated annealing	48 860 495	767 791	1 162 866	1 928 240	1.27	0.87	13.2
Original LRU	48 854 637	765 809	1 163 314	1 929 072	1.09	0.75	12.8

Table 4.23 Comparison of configurations found by genetic algorithm and simulated annealing against the default LRU policy on Mawi PCAP with cache size reduced to 2^{11}

Algorithm	Hits	Empty	Not empty	Exported	Avg. lookup	Var. lookup	Time (s)
Genetic algorithm	25 402 836	6100	4 118 823	4 120 993	3.22	15.4	9.1
Simulated annealing	25 398 664	6245	4 122 996	4 125 165	3.2	15.7	9.1
Original LRU	25 058 603	4154	4 463 114	4 465 193	3.09	13.2	11.1

Table 4.24 Comparison of different export periods with original implementation on PCAP files

PCAP File	Implementation	Hits	Empty	Not empty	Exported	Periodic	On finish	Avg. lookup	Var. lookup	Time (s)
Traffic	Original	14 719 855	131 084	353 776	484 848	0	131 072	1.21	1.18	4.6
	Export thread, 1x per 1700ms	14 719 855	131 155	353 776	484 848	0	131 072	1.21	1.18	3.4
Sh	Original	45 852 482	719 235	0	719 192	702 958	11 647	1.0	0.01	9.3
	Export thread, 1x per 1700ms	45 860 224	719 389	0	718 932	560 595	13 587	1.0	0.01	7.4
	Export thread, 1x per 900ms	45 851 041	719 391	0	719 176	699 973	11 234	1.0	0.01	8.8
Tul	Original	48 854 637	765 809	1 163 314	1 929 072	637 114	125 821	1.09	0.75	12.8
	Export thread, 1x per 1700ms	48 859 805	770 698	1 160 468	1 928 545	99 358	129 614	1.09	0.75	9.3
	Export thread, 1x per 900ms	48 859 491	770 743	1 160 416	1 928 210	172 061	128 826	1.09	0.75	9.4
	Export thread, 1x per 10ms	48 853 223	770 660	1 160 556	1 927 523	618 643	125 887	1.09	0.75	12.1
Mawi	Original	28 285 051	1 248 091	1652	1 247 405	1 189 669	47 410	1.22	0.99	8.6
	Export thread, 1x per 1700ms	28 307 381	1 256 712	1581	1 246 269	690 014	53 578	1.22	0.99	6.0
	Export thread, 1x per 900ms	28 301 282	1 256 706	1581	1 245 801	832 848	51 114	1.22	0.99	6.3
	Export thread, 1x per 10ms	28 283 152	1 256 712	1582	1 247 095	1 162 456	47 409	1.22	0.98	8.0

As Traffic PCAP does not contain any expired flow, the only interesting part is the time, where we can observe a significant improvement of 35%. The Sh PCAP file has good statistics, too; exporting once per 1700ms is enough to achieve results similar to the original implementation while having an advantage in time over 24%. Exporting once per 900ms is enough to achieve almost equal results to the original implementation, still having an advantage in time. For the Tul PCAP, higher export frequency is required to achieve original results. Export once per 10ms achieves original results while still having an advantage in time of 5%.

Mawi PCAP acts very similar to Tul PCAP.

Multithread optimization seems useful; we add this feature for live test builds with a sleep time of 1500ms.

4.6 Flood detection

As we can see on a graph of new flows (Figure 4.9), the record contains one peek of attacks – a first peek at 7:00:46. Two other peeks seem to be significant too, but the count of new flow is comparable to the peek of Tul PCAP (Figure 4.5).

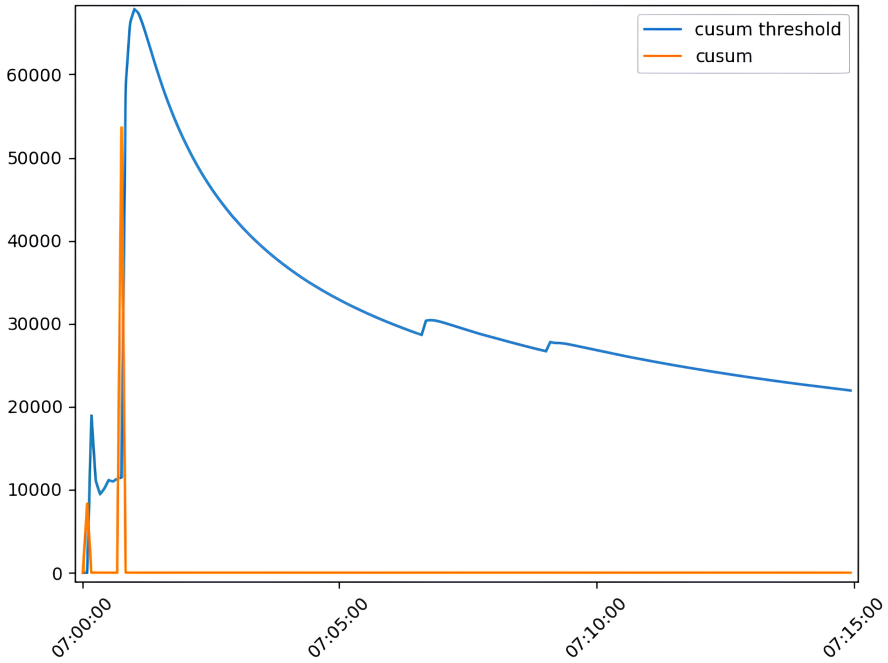

Figure 4.11 Illustration of flood detection.

Comparison of the S_t and the threshold $T_{cusum,t}$ illustrated in Figure 4.11. The algorithm detects (ignoring the initial peak) only the attack at 7:00:46 but not the subsequent peaks. This behavior occurs because of the algorithm design: the calculations of the cumulative sum are connected with the average count of cache misses, which is very high after the first peak, meaning that the first high peak blocks the detection of other subsequent peaks with a lower count of cache misses.

Chapter 5
Conclusion

In our work, we analyzed the cache of the ipfixprobe flow exporter with a focus on the performance of the current implementation. We compared it with other existing solutions and identified performance issues. To address found issues, we proposed a few approaches:

- We experimentally tested many hash functions: XXH3_64bits, XXH3_128bits, XXH64, CRC32c, FarmHash, Murmurhash, SuperFastHash, and Toeplitz hash on samples of real traffic and found the best non-cryptographic hash function for the task of flow hashing.
- Row management policies: LRU, LRU on the heap, LIRS, 2Q in simplified and full variants, flow-driven rule caching; all replacement policies were tested on real traffic records.
- We implemented an adaptive replacement policy based on a genetic algorithm with the predictor of the next packet arrival time. Based on that policy, we created our adaptive replacement policy, which uses simulated annealing with a taboo list to search configurations. Both configurations were tested on traffic records and compared.
- We implemented and tested multi-thread optimization that effectively divides the work of the cache into separate threads.
- Flood detection system using exponentially weighted moving average extended by thresholds and a cumulative sum was implemented and successfully tested on the record of DDoS attacks.

The most efficient and universal changes were added to the test version. According to the work objectives, the cache was refactored. Finally, the cache, improved, refactored, and approved by its maintainers, was sent back to the original repository with a pull request. After the effectivity of the new version is tested on high loads, changes will become a part of the original repository, making a new version that brings a higher cache hit ratio, lower working time, and more features, i.e., increased service level, available for users.

Due to theme complexity and size, we didn't manage to cover many other essential aspects of flow cache optimizations, such as more profound research of possible usages of machine learning in the area of flow processing or processor acceleration – usage of special fast processor instruction in the task of flow caching, which can be a good theme for further researches in this area.

Appendix A
Appendix

This chapter is dedicated to the less important data from the main chapters.

Detailed description of inner algorithms of ipfixprobe's process_packet can be found in algorithms 13-18

Algorithm 13 find_flow

Input: Integer *flow_index*, Packet *packet*
Output: Integer *flow_index*, Boolean *found* - true if the flow was found
hash_direct = XXH64(*packet.src_ip*, *packet.dst_ip*, *packet.src_port*, *packet.dst_port*, *packet.protocol*, *packet.ip_version*, *packet.vlan_id*)
row_begin = get_row_begin(low_bits(*hash_direct*))
for *row_begin* ≤ *i* < *row_begin* + *cache_line_length* **do**
  if *flow_table*[*i*].*hash* == *hash_direct* **then**
    flow_index = *i*
    return *true*
  end if
end for
hash_reversed = XXH64(*packet.dst_ip*, *packet.src_ip*, *packet.dst_port*, *packet.src_port*, *packet.protocol*, *packet.ip_version*, *packet.vlan_id*)
row_begin = get_row_begin(low_bits(*hash_reversed*))
for *row_begin* ≤ *i* < *row_begin* + *cache_line_length* **do**
  if *flow_table*[*i*].*hash* == *hash_reversed* **then**
    flow_index = *i*
    return *true*
  end if
end for
return *false*
return

Additional algorithms for adaptive row policy are at table A.1 and alg. 19-29

Algorithm 14 enhance_flow_position

Input: index of the flow to enhance *flow_index*
row_begin = get_row_begin(*flow_index*)
circular_shift(*row_begin*, *flow_index*)
//Flow that was on position *flow_index* is now on *row_begin*
return

Algorithm 15 create_new_flow

Input: index of the row start *row_begin*, IP Packet *packet*
empty_place = find_empty_place(*row_begin*)
if *empty_place* == *NO_EMPTY_PLACE* **then**
  empty_place = free_place_in_row(*row_begin*)
end if
create(*empty_place*, *packet*)
return

Algorithm 16 find_empty_place

Input: index of the row start *row_begin*
Output: index of the empty cell
for *row_begin* $\leq i <$ *row_begin*+*cache_line_length* **do**
  if *flow_table*[i].*is_empty()* **then**
    return *i*
  end if
end for

Algorithm 17 circular_shift

Input: index of the shift start *start*,index of the shift end *end*
temporary = $flow\_table[start]$
for *start* $< i \leq$ *end* **do**
  flow_table[i] = *flow_table*[$i - 1$]
end for
flow_table[start] = *temporary*
return

Algorithm 18 free_place_in_row

Input: index of the row start *row_begin*
Output: index of the empty cell
last_row_record = *row_begin* + *cache_line_length* − 1
row_middle_index = *row_begin* + $\frac{cache\_line\_length}{2}$
circular_shift(*row_middle_index*, *last_row_record*)
export(*row_middle_index*)
return *row_middle_index*

Algorithm 19 binary_tournament

Input: generation to choose from *generation*
Output: best of two randomly chosen configurations
first = *generation*[random(0, *generation.size()*−1)]
second = *generation*[random(0, *generation.size()*−1)]
if *first* < *second* **then**
  return *first*
else
  return *second*
end if

Algorithm 20 create_generation

Input: configuration to create generation from *configuration*
Output: generation created from seed
output = *EMPTY_GENERATION*
while *output.size()* < *generation_size* **do**
  new_solution = *configuration*
  new_solution.mutate()
  output.add(*new_solution*)
  repeat
    output.last_configuration().mutate()
    //If we discover, that a duplicate to an existing configuration was generated, we call mutation operation until the configuration is unique.
  until NOT *output.all_configurations_are_unique()*
end while
return *output*

Algorithm 21 mutate

Input: configuration to mutate *configuration*
Output: mutated_configuration *configuration*
repeat
  new_solution = *configuration*
  new_solution.mutate_counts()
  new_solution.fix_counts()
  new_solution.mutate_counts_by_one()
  new_solution.fix_counts()
  new_solution.mutate_increment()
  new_solution.fix_targets()
  new_solution.mutate_targets()
  new_solution.fix_targets()
  new_solution.mutate_targets_by_one()
  new_solution.fix_targets()
  new_solution.mutate_insert_position()
until *new_solution* == *configuration*
return *new_solution*

Algorithm 22 mutate_counts

Input: configuration to mutate *configuration*
Output: mutated configuration *configuration*
flows_in_row = *configuration.length()*
for all *tuple* in *configuration* **do**
  if random(0.2) **then**
    tuple.count = random(1, $\frac{flows\_in\_row}{4}$)
  end if
end for

Algorithm 23 fix_counts

Input: configuration to fix *configuration*
Output: fixed configuration *configuration*
flows_in_row = *configuration.length()*
repeat
  sum = 0
  for all *tuple* in *configuration* **do**
    sum += *tuple.count*
  end for
  random_tuple_index = random(0, *configuration.tuple_count()* − 1)
  if *sum* > *flows_in_row* AND *configuration.tuples*[*random_tuple_index*].*count* > 1 **then**
    configuration.tuples[*random_tuple_index*].*count* -= 1
  else if *sum* < *flows_in_row* **then**
    configuration.tuples[*random_tuple_index*].*count* += 1
  end if
until *sum* ≠ *flows_in_row*

Algorithm 24 mutate_counts_by_one

Input: configuration to mutate *configuration*
Output: mutated configuration *configuration*
for all *tuple* in *configuration* **do**
  if random(0.4) **then**
    if random(0.5) AND *tuple.count* ≠ 1 **then**
      tuple.count -= 1
    else
      tuple.count += 1
    end if
  end if
end for

Algorithm 25 mutate_increment

Input: configuration to mutate *configuration*
Output: mutated configuration *configuration*
for all *tuple* in *configuration* **do**
  if random(0.3) **then**
    tuple.increment = NOT *tuple.increment*
  end if
end for

Algorithm 26 fix_targets

Input: configuration to fix *configuration*
Output: fixed configuration *configuration*
for $0 \leq$ *tuple_index* $<$ *configuration.tuple_count()* $- 1$ **do**
  current_tuple = *configuration.tuples*[*tuple_index*]
  next_tuple = *configuration.tuples*[*tuple_index* $+ 1$]
  if *current_tuple.increment* == *true* AND *current_tuple.target* + *current_tuple.count* $>$ *next_tuple.target* **then**
    next_tuple.target = *current_tuple.target* + *current_tuple.count*
  else if *current_tuple.target* $>$ *next_tuple.target* **then**
    swap(*current_tuple.target*, *next_tuple.target*)
  end if
  if *next_tuple.target* $>$ *current_tuple.target* **then**
    next_tuple.target = *current_tuple.target* + *current_tuple.count*
  end if
end for

Algorithm 27 mutate_targets

Input: configuration to mutate *configuration*
Output: mutated configuration *configuration*
possible_target_maximum = 0
for all *tuple* in *configuration* **do**
  possible_target_maximum += *tuple.count*
  if random(0.2) **then**
    tuple.count = random(0,*possible_target_maximum* $- 1$)
  end if
end for

Algorithm 28 mutate_targets_by_one

Input: configuration to mutate *configuration*
Output: mutated configuration *configuration*
//Exclude the first and the last tuple from the loop
for $1 \leq i <$ *configuration.tuples_count()* $- 1$ **do**
  tuple = *configuration.tuples*[*i*]
  if random(0.4) **then**
    if random(0.5) **then**
      tuple.target -= 1
    else
      tuple.target += 1
    end if
  end if
end for
last_tuple = *configuration.tuples*[*configuration.tuples_count()* $- 1$]
if random(0.4) **then**
  tuple.target -= 1
end if

Table A.1 Random generation functions.

The function **random(p)** returns true with probability p.
The function **random(a,b)** returns a random integer number from the uniform distribution over the interval [a, b].

Algorithm 29 mutate_insert_position

Input: configuration to mutate $configuration$
Output: mutated configuration $configuration$
flows_in_row = *configuration.length()*
if random(0.2) **then**
  configuration.insert_position = random(0, *flows_in_row* − 1)
end if
if random(0.2) **then**
  configuration.offset_short = random($-\frac{cache\_line\_length}{2}$, $\frac{cache\_line\_length}{2}$)
end if
if random(0.2) **then**
  configuration.offset_medium = random(*configuration.offset_short*, $\frac{cache\_line\_length}{2}$)
else
  configuration.offset_medium =
  max(*configuration.offset_medium*, *configuration.offset_short*)
end if
if random(0.2) **then**
  configuration.offset_long = random(*configuration.offset_medium*, $\frac{cache\_line\_length}{2}$)
else
  configuration.offset_long =
  max(*configuration.offset_long*, *configuration.offset_medium*)
end if
if random(0.2) **then**
  configuration.offset_never = random(*configuration.offset_long*, $\frac{cache\_line\_length}{2}$)
else
  configuration.offset_never =
  max(*configuration.offset_never*, *configuration.offset_long*)
end if

Appendix B
Appendix

This section is dedicated to compilation and run of ipfixprobe.

B.1 Compilation

Code listing B.1 Steps required to compile ipfixprobe

```
autoreconf -i
./configure # or ./configure --with-pcap to compile with pcap plugin
make
sudo make install
```

B.2 Run

Code listing B.2 Examples of ipfixprobe run with different plugins

```
# Capture from eth0 interface using raw sockets, print flows to console
./ipfixprobe -i 'raw;ifc=eth0' -o 'text'

#Read input by pcap plugin from eth0 using cache with cache size 2^16
    and row length 2^4 with output including mac addresses printed to
    console
./ipfixprobe -i 'pcap;ifc=eth0' -s 'cache;' -o 'text;m'

#Read input by pcap plugin from eth0 using cache with cache size 2^17
    and row length 2^3 with output including mac addresses printed to
    console
./ipfixprobe -i 'pcap;ifc=eth0' -s 'cache;s=17;l=3' -o 'text;m'
```

Bibliography

[1] aappleby. *aappleby/smhasher*. original-date: 2015-03-16T19:14:44Z. Apr. 2024. URL: https://github.com/aappleby/smhasher (visited on 04/25/2024).

[2] Tomas Benes, Jaroslav Pesek, and Tomas Cejka. "Look at my Network: An Insight into the ISP Backbone Traffic". In: *2023 19th International Conference on Network and Service Management (CNSM)*. ISSN: 2165-963X. Oct. 2023, pp. 1–7. DOI: 10.23919/CNSM59352.2023.10327823. URL: https://ieeexplore.ieee.org/document/10327823 (visited on 04/27/2024).

[3] Behavior-Centric Cybersecurity Center (BCCC). *ahlashkari/CICFlowMeter*. original-date: 2018-02-12T16:57:30Z. May 2024. URL: https://github.com/ahlashkari/CICFlowMeter (visited on 05/04/2024).

[4] *CESNET/ipfixprobe*. original-date: 2017-10-31T12:55:06Z. Apr. 2024. URL: https://github.com/CESNET/ipfixprobe (visited on 04/29/2024).

[5] *Cisco Systems NetFlow Services Export Version 9*. URL: https://www.ietf.org/rfc/rfc3954.txt (visited on 05/10/2024).

[6] *CN-TU/go-flows*. original-date: 2018-01-23T13:18:13Z. Feb. 2024. URL: https://github.com/CN-TU/go-flows (visited on 04/29/2024).

[7] Yann Collet. *Cyan4973/xxHash*. original-date: 2014-04-30T23:32:49Z. Apr. 2024. URL: https://github.com/Cyan4973/xxHash (visited on 04/25/2024).

[8] *Encrypted Traffic, Once Thought Safe, Now Responsible For Most Cyberthreats*. en. URL: https://www.darkreading.com/application-security/encrypted-traffic-once-thought-safe-now-responsible-for-most-cyberthreats (visited on 05/10/2024).

[9] felixe. *felixe/ccsVermont*. original-date: 2015-05-21T15:35:49Z. June 2018. URL: https://github.com/felixe/ccsVermont (visited on 04/29/2024).

[10] *Flooding Attack - an overview — ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/computer-science/flooding-attack (visited on 04/23/2024).

[11] *google/crc32c*. original-date: 2017-08-04T17:15:20Z. Apr. 2024. URL: https://github.com/google/crc32c (visited on 04/23/2024).

[12] *google/farmhash*. original-date: 2015-08-14T21:01:50Z. Apr. 2024. URL: https://github.com/google/farmhash (visited on 04/25/2024).

[13] Rick Hofstede et al. "Towards real-time intrusion detection for NetFlow and IPFIX". In: *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*. ISSN: 2165-963X. Oct. 2013, pp. 227–234. DOI: 10.1109/CNSM.2013.6727841. URL: https://ieeexplore.ieee.org/document/6727841 (visited on 04/23/2024).

[14] *https://github.com/cisco/joy/tree/master*. URL: https://github.com/cisco/joy/tree/master (visited on 04/29/2024).

[15] *ICT Statistics*. URL: https://www.itu.int/ITU-D/ict/statistics/ict/ (visited on 05/10/2024).

[16] Hitoshi Irino. *irino/softflowd*. original-date: 2016-07-31T00:22:47Z. Apr. 2024. URL: https://github.com/irino/softflowd (visited on 04/29/2024).

[17] Manish Jethani. *mjethani/superfasthash*. original-date: 2019-03-23T12:17:44Z. Feb. 2024. URL: https://github.com/mjethani/superfasthash (visited on 04/25/2024).

[18] T. Johnson and D. Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm". In: Sept. 1994. URL: https://www.semanticscholar.org/paper/2Q%3A-A-Low-Overhead-High-Performance-Buffer-Johnson-Shasha/5fa357b43c8351a5d8e7124429e538ad7d687abc (visited on 04/23/2024).

[19] Hugo Krawczyk. "New Hash Functions for Message Authentication". en. In: *Advances in Cryptology — EUROCRYPT '95*. Ed. by Louis C. Guillou and Jean-Jacques Quisquater. Berlin, Heidelberg: Springer, 1995, pp. 301–310. ISBN: 978-3-540-49264-1. DOI: 10.1007/3-540-49264-X_24.

[20] He Li et al. "FDRC: Flow-Driven Rule Caching Optimization in Software Defined Networking". In: June 2015, pp. 5777–5782. DOI: 10.1109/ICC.2015.7249243.

[21] *LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance: ACM SIGMETRICS Performance Evaluation Review: Vol 30, No 1*. URL: https://dl.acm.org/doi/10.1145/511399.511340 (visited on 04/23/2024).

[22] *Multiplying a Toeplitz matrix by a vector - Alin Tomescu*. URL: https://alinush.github.io/2020/03/19/multiplying-a-vector-by-a-toeplitz-matrix.html (visited on 04/23/2024).

[23] *NetFlow Version 9 Flow-Record Format [IP Application Services]*. en. URL: http://www.cisco.com/en/US/technologies/tk648/tk362/technologies_white_paper09186a00800a3db9.html (visited on 04/30/2024).

[24] Brian Trammell. *britram/cert-netsa-yaf*. original-date: 2019-01-28T11:21:33Z. June 2023. URL: https://github.com/britram/cert-netsa-yaf (visited on 04/29/2024).

[25] *UniRec: UniRec*. URL: https://nemea.liberouter.org/doc/unirec/md_doc_intro.html (visited on 05/10/2024).

[26] Gernot Vormayr, Joachim Fabini, and Tanja Zseby. "Why are My Flows Different? A Tutorial on Flow Exporters". In: *IEEE Communications Surveys & Tutorials* 22.3 (2020). Conference Name: IEEE Communications Surveys & Tutorials, pp. 2064–2103. ISSN: 1553-877X. DOI: 10.1109/COMST.2020.2989695. URL: https://ieeexplore.ieee.org/abstract/document/9076310 (visited on 04/29/2024).

[27] *wide-vsix/linux-flow-exporter*. original-date: 2022-06-26T11:43:33Z. Apr. 2024. URL: https://github.com/wide-vsix/linux-flow-exporter (visited on 04/24/2024).

[28] Martin Žádník. "Optimalizace sledování síťových toků". cs. In: (June 2019). Publisher: Vysoké učení technické v Brně. Fakulta informačních technologií. URL: http://hdl.handle.net/11012/63224 (visited on 04/23/2024).

[29] Tanja Zseby et al. *Requirements for IP Flow Information Export (IPFIX)*. Request for Comments RFC 3917. Num Pages: 33. Internet Engineering Task Force, Oct. 2004. DOI: 10.17487/RFC3917. URL: https://datatracker.ietf.org/doc/rfc3917 (visited on 04/23/2024).

Consents of the attachment

- `readme.txt` ..... short description of the media content
- `exe` ..... directory with binary form of ipfixprobe
- `src`
  - `impl` ..... source codes
  - `scripts` ..... Used scripts source codes
  - `thesis` ..... LaTeX source codes
- `text` ..... text of the work
  - `thesis.pdf` ..... text of the work in the PDF format