HyPaFilter – A Versatile Hybrid FPGA Packet Filter

Andreas Fiessler[†] Sven Hager[‡] Björn Scheuermann[‡] Andrew W. Moore[§]

[†]genua mbH, Germany
andreas_fiessler@genua.de

[‡]Humboldt University of Berlin, Germany
{hagersve,scheuermann}@informatik.hu-berlin.de

[§]University of Cambridge, UK
andrew.moore@cl.cam.ac.uk

ABSTRACT

With network traffic rates continuously growing, security systems like firewalls are facing increasing challenges to process incoming packets at line speed without sacrificing protection. Accordingly, specialized hardware firewalls are increasingly used in high-speed environments. Hardware solutions, though, are inherently limited in terms of the complexity of the policies they can implement, often forcing users to choose between throughput and comprehensive analysis. On the contrary, complex rules typically constitute only a small fraction of the rule set. This motivates the combination of massively parallel, yet complexity-limited specialized circuitry with a slower, but semantically powerful software firewall. The key challenge in such a design arises from the dependencies between classification rules due to their relative priorities within the rule set: complex rules requiring software-based processing may be interleaved at arbitrary positions between those where hardware processing is feasible. We therefore discuss approaches for partitioning and transforming rule sets for hybrid packet processing, and propose HyPaFilter, a hybrid classification system based on tailored circuitry on an FPGA as an accelerator for a Linux `netfilter` firewall. Our evaluation demonstrates 30-fold performance gains in comparison to software-only processing.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: Security and protection

Keywords

Packet classification, FPGA hardware accelerator, Firewall

ANCS '16, March 17 - 18, 2016, Santa Clara, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4183-7/16/03. . . $15.00
DOI: http://dx.doi.org/10.1145/2881025.2881033

1. INTRODUCTION

Software firewalls like `netfilter/iptables` [3], `pf` [4], or `ipfw` [2] are widely used in practice, in both standalone applications and as a basis for professional security appliances [1]. Their main advantages are flexibility and powerful filtering options, as well as their easy setup and handling, since they can be used on top of common operating systems with commercial off-the-shelf (COTS) hardware. These CPU-based architectures, however, hardly meet the line rate packet processing requirements for high link speeds such as 40 Gbps or beyond, which leave only small processing time frames of 8 ns or less for each packet in the worst case [21]. In contrast, packet classification systems based on special purpose hardware, such as network processors (NPUs) [24,26], field-programmable gate arrays (FPGAs) [11, 17, 20, 21], graphics processing units (GPUs) [35], or application-specific circuits (ASICs) [8] provide an abundant amount of parallelism which can be used to process many network packets at once. Furthermore, the matching process for every single packet is often parallelized, which leads to large achievable throughputs of up to 640 Gbit/s [8].

However, dedicated hardware is significantly more constrained with respect to the expressiveness of the supported rule set semantics: while the functionality of software-based classification systems ranges from stateful connection tracking over probabilistic matching to deep packet inspection [2–4], specialized hardware engines are often restricted to simple stateless packet classification with no or only limited connection tracking capabilities [8, 11, 17, 21, 35]. Moreover, while software firewalls can utilize a virtually unlimited amount of memory for storing policies and connection states, hardware firewalls have to operate within fixed boundaries.

In order to combine the advantages of massively parallel matching hardware and powerful inspection capabilities of software-based packet filters, we propose *HyPaFilter*, a hybrid packet classification concept. The HyPaFilter approach aims to reach the packet rate and processing latency of a dedicated hardware firewall for common, easy to classify traffic, while providing the flexibility and functionality of a software firewall for packets which require complex processing. To this end, HyPaFilter partitions a user-defined packet processing policy into a simple part manageable by specialized matching hardware, and a complex part, which is handled in software. We found that a key challenge in such a hybrid design, regardless of its concrete implementation, is the proper handling of dependencies between different rules in the specified policy: if the hardware detects a rule match of an incoming packet in the simple part of the policy, it must ensure that the packet does not match a more highly prioritized rule installed in the software filter *before* the action specified by the hardware-detected rule is applied. However, it is desirable to avoid a full-fledged software packet classification whenever possible in order to achieve the full hardware speedup for a large number of packets. In order to overcome this challenge, the HyPaFilter approach determines the largest rule index in the simple rule set up to which a hardware-only classification is safely possible. Furthermore, even if complex processing for a packet is required, the matching information from the hardware can be reused in order to narrow down the set of rules the software filter has to match against this packet. We also address policy updates, as both the simple and complex part of a policy can change at arbitrary positions after an initial system setup.

As FPGA-based systems are suitable candidates for high performance, low latency network applications [25], we prototyped the HyPaFilter approach using a standard Linux host using `netfilter/iptables` as the software packet filter, combined with the NetFPGA SUME [37] platform. In this setup, the NetFPGA SUME is configured with tailored logic which matches packets against every simple rule in parallel, allowing it to perform basic firewalling tasks without involving the host at all at speeds of up to 40 Gbps for 64 byte frames. Complex rules and policy updates are implemented in `netfilter` in order to allow for comprehensive packet analysis as well as short rule update latencies. Whenever possible, updates that involve simple rules are moved from the software filter to the hardware filter during the next hardware configuration phase.

Of course, the achievable performance of HyPaFilter is dependent on both the structure of the implemented policy as well as on network traffic characteristics. However, previous examination of real-world traffic in [6] showed that the fraction of traffic which can be analyzed by simple packet filter rules is large enough to expect a significant performance gain in practical applications. Our evaluation results indicate that the HyPaFilter system can significantly increase the maximum achievable classification throughput over a software-only approach even for policies with many and widely spread complex rules. In the current state of development, stateful firewalling relies on the software firewall only. Therefore, the intended use case prefers scenarios like bridging firewalls, denial-of-service protection, or demilitarized zone configurations, where many policies can be implemented by stateless firewall rules.

To sum up, the main contributions of this paper are: (1) We present a hybrid packet classification concept which combines the benefits of dedicated matching hardware with powerful matching semantics typically found in software-only approaches. We prototyped this concept on a combination of a NetFPGA SUME and a Linux host system. However, at its core, the HyPaFilter approach does not make any assumptions on the used hardware and can be used with other kinds of hardware, such as GPUs or NPUs. (2) We describe different strategies in order to achieve a good policy separation, which is one of the key challenges in such hybrid designs. (3) We provide a detailed study on how the structure of the implemented policies affects the achievable throughput in our hybrid system.

The remainder of this paper is structured as follows: in Section 2, we discuss related work in this field of research. Next, we briefly introduce the packet classification problem in Section 3. Sections 4 and 5 describe the hybrid matching algorithm as well as the architecture of the HyPaFilter system, respectively. Finally, we present our evaluation results in Section 6 and conclude this paper in Section 7.

2. RELATED WORK

Network packet classification has been of major interest to the research community due to its importance for packet-switched networks [13, 32]. Most of the scientific work in this area focuses on the geometric variant of the packet classification problem, which considers a limited number of packet header fields and does not take other criteria into account, such as packet payloads or connection states. These research efforts can be roughly split into the following categories: *classification algorithms*, *hardware architectures*, and *rule set transformations*.

Classification algorithms traverse an algorithm-specific data structure in order to find the highest prioritized rule that matches on all relevant header fields of an incoming packet. Such approaches exist in many different flavors, such as decision tree algorithms [12, 30, 34], bit vector searches [7, 22], or techniques based on hash maps [31]. In comparison to a straightforward linear search through the rule set, these advanced classification algorithms provide significantly faster classification performances [13]. Despite this fact, many practically used packet classification systems, such as `netfilter` [3] and `pf` [4], implement a linear search in order to discriminate network packets and thus generally suffer from low classification performance [16]. However, these systems also provide powerful rule set semantics which are more expressive than plain stateless header field inspection.

Specialized hardware architectures used for packet classification typically employ large amounts of parallelism in order to achieve high throughput rates. The most common hardware architecture used for packet classification is *Ternary Content Addressable Memory (TCAM)*, which matches the entire rule set in parallel against incoming packet headers [28] and can thus process every incoming packet in a small, fixed number of clock cycles. On the downside, TCAMs are expensive, power-intensive, and cannot natively represent rules with range or negation tests [17]. Other widely used implementation platforms for packet classification are FPGAs [11, 17, 18, 20], NPUs [24, 26], and GPUs [35], which typically also employ a full parallel match [17] or implement a classification algorithm which is amenable for parallelization/pipelining [11, 18, 20, 24, 26, 35]. Although significantly faster than software-based systems, these approaches only support limited stateless matching semantics. In contrast, the HyPaFilter design combines the flexibility of existing software engines with the processing speed of dedicated hardware.

Rule set transformation techniques are orthogonal to the employed classification algorithm/architecture. The goal is to transform an initial rule set $\mathscr{R}$ into an equivalent rule set $\mathscr{R}'$ which can be traversed faster for incoming network packets. Existing approaches for rule set transformation are rule set minimization [23] or the encoding of decision tree data structures into the rule set [16]. HyPaFilter utilizes the latter transformation variant to install complex rules in the software filter which can reuse the hardware classification result in order to accelerate the software matching.

The possibility of hybrid packet filters for FPGA/`netfilter` and NPU/`netfilter` combinations has been previously addressed in [10] and [6], respectively. However, these works do not answer the following key questions: (1) How should a packet processing policy be deployed in a hybrid system in order to reach high classification performance? (2) How does the hybrid system implement rule set updates? In order to provide an answer to these questions, we present three rule set partitioning schemes as well as update mechanisms to handle rule set changes.

3. PROBLEM STATEMENT

In this section, we first introduce the packet classification problem, which serves as the vantage point for the extended packet classification problem, which we define subsequently.

3.1 Packet Classification Problem

The packet classification problem, as it is most often seen in the literature [7, 12, 17, 31], can be formally defined as follows: let

$\mathscr{H} = (H_1 \in D_1, \ldots, H_K \in D_K)$ be a tuple of *header values* and $\mathscr{R} = \langle R_1, \ldots, R_N \rangle$ be an ordered list of *rules* R_i, which is called the *rule set*. Here, D_j is called the *domain* of the jth header field. For the rest of this paper, we assume that each D_j is a range of non-negative integers, in order to cover common header fields like IP addresses, ports, or protocol numbers. Every rule R_i consists of K *checks* $C_i^j : D_j \to \{\text{true}, \text{false}\}$ with

$$R_i = C_i^1 \wedge \ldots \wedge C_i^K.$$

R_i is said to *match* the header tuple $\mathscr{H}$ if $C_i^j(H_j)$ yields true for all $j \in \{1, \ldots, K\}$. The goal of the packet classification problem is to find the smallest index i^* such that rule R_{i^*} matches $\mathscr{H}$. This index can subsequently be used in order to look up and execute an *action* for the corresponding matching rule, such as DROP or ACCEPT. Here, the checks C_i^j are often simple equality, range, or subnet tests. An example for an `iptables` rule which consists of these basic tests is

```
-p tcp --dport 80 --dst 1.2.3.4 -j ACCEPT
```

which accepts incoming TCP packets with destination port 80 addressed to 1.2.3.4.

The most straightforward way to solve the packet classification problem, which is implemented by many practically used classification systems [3, 4], is a linear search through the rule set $\mathscr{R}$. Although a linear search is simple to implement and memory efficient, it does not provide good classification performance. Faster classification algorithms, such as [7, 12, 22], exploit the simplicity of the above mentioned tests in order to translate the rule set $\mathscr{R}$ into search data structures for fast traversal at runtime. However, this is independent of the semantic specification of the rule set, which is typically still done through a linear list of rules, ordered by priority. Any other representation used by the algorithm must not change these rule set semantics, so that HyPaFilter can remain independent of the specific algorithms and data structures used internally in the software classificator.

3.2 Extended Packet Classification Problem

Practical packet filter implementations, such as `netfilter` [3], `pf` [4], and `ipfw` [2], support advanced matching criteria in order to increase the expressiveness of an implemented filtering policy. Examples for such sophisticated checks are connection tracking, rate limiting, unicast reverse path forwarding (URPF) verification, probability-based matching, or deep packet inspection. When used in conjunction with the previously defined basic checks, these tests can greatly foster both the robustness and effectiveness of the used packet filtering policy. In such a system, a rule R_i can be modelled as

$$R_i = C_i^1 \wedge \ldots \wedge C_i^K \wedge A_i,$$

where A_i is a rule-specific combination of advanced matching criteria. An example `iptables` rule which scans the packet payload in addition to basic header checks could look like

```
-p tcp --dport 80 -m string --string "BAD" --algo
                  bm -j DROP.
```

Here, the A part of this rule is the test for the string “BAD”. In contrast to most other existing hardware-accelerated classification systems, the HyPaFilter approach tackles both the basic and the extended packet classification problems.

4. MATCHING ALGORITHM

In order to support good classification performance, small rule set update latencies and expressive rule set semantics, the HyPaFilter system relies on a hybrid matching algorithm which first processes every incoming packet on the FPGA. After the packet is matched, the FPGA circuitry decides whether the packet requires further, potentially more complex processing in the host-based `netfilter` classification system. For the remainder of this paper, we follow the nomenclature of [36] and denote packets forwarded by the FPGA as *forwarded*, while those diverted to the host are called *shunted*. Although software-based packet processing is significantly more expensive than classification on the FPGA, we will present a strategy how the software rules can be structured in order to keep the number of traversed rules as small as possible in case of a packet shunt. In the remainder of this section, we will explain the architecture of the FPGA-based filter, the dispatch logic which decides whether a processed packet must be shunted to the host, as well as the rule set structure implemented in `netfilter`.

4.1 Hardware Filter

Let $\mathscr{R}_S \subseteq \mathscr{R}$ be the sublist containing the simple rules without advanced matching criteria, and $\mathscr{R}_C \subseteq \mathscr{R}$ be the sublist of complex rules with advanced matching criteria. That is, $\mathscr{R}_S \cup \mathscr{R}_C = \mathscr{R}$ and $\mathscr{R}_S \cap \mathscr{R}_C = \emptyset$. The classification system implemented on the FPGA solves the classic packet classification problem on $\mathscr{R}_S$, as introduced in Section 3.1. It therefore implements every $R_i \in \mathscr{R}_S$. In order to achieve high matching performance on the FPGA with a low deterministic processing latency per packet, we decided to use a rule set specific parallel matching engine, which is generated by translating every rule $R_i \in \mathscr{R}_S$ at setup time into a specialized match unit M_j specified in VHDL, similar to the technique proposed in [17]. Here, j is the index of rule R_i within the sublist $\mathscr{R}_S$. This process is illustrated in Figure 1. Since each rule in $\mathscr{R}_S$ is a conjunction of simple checks, such as subnet tests or port range tests, the match units are composed of a small number of basic comparator circuits. For example, a rule which matches TCP packets if the source IP address is in the subnet 203.0.0.0/8 with destination port 80 is translated into three specific comparator circuits: the first one compares the packet's transport protocol field against the TCP transport protocol number 6, while the second and third comparators compare the first octet of the packet's source IP address against 203 and the packet's destination port against 80, respectively. Finally, the results of these comparators are connected with an AND gate in order to determine whether the rule matches or not. As the match units are arranged in parallel, incoming network packets can be matched against the entire simple rule set $\mathscr{R}_S$ in a single clock cycle, which yields a result bit vector V of size $|\mathscr{R}_S|$. Here, the ith position V_i of the result vector V stores a 1 if rule R_i matches the current packet, otherwise V_i is set to 0. As we are interested in the most highly prioritized matching rule, we employ a priority encoder in order to determine the index of the first enabled bit in

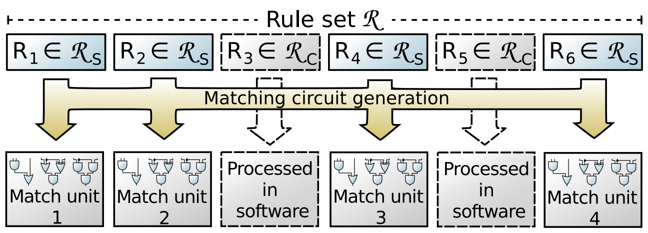


Figure 1: Translating simple rules into matching circuitry.

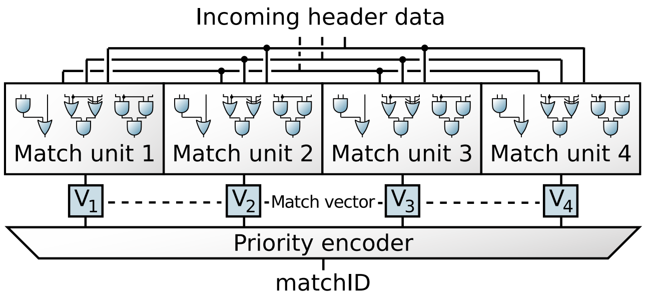


Figure 2: Parallel match of packet header data against $\mathscr{R}_S$.

V, which we will refer to as *matchID* in the following. The entire hardware matching process is sketched in Figure 2.

Up to this point, the packet classification problem is solved for the simple rule set $\mathscr{R}_S$ solely in hardware, as the matchID can be used in order to quickly look up the action $A_{\mathscr{R}_S,P}$ which must be applied for the current packet P. If the installed rule set $\mathscr{R}$ does not specify any rules with complex checks, i. e., if $\mathscr{R}_C = \emptyset$ and thus $\mathscr{R}_S = \mathscr{R}$, then the classification is complete at this point and $A_{\mathscr{R}_S,P}$ is applied to the current packet. However, if $\mathscr{R}_C \neq \emptyset$, then additional processing may be required by the software filter residing on the host system. This is the case when the matchID is greater or equal to the smallest index of a rule in $\mathscr{R}$ that specifies complex checks. In the following, we denote the smallest index of a rule in $\mathscr{R}$ with complex checks by the term *validID*.

For instance, consider the case that the hardware matching circuit for the rule set sketched in Figure 1 computes that matchID is 3 for an incoming packet P (that is, the packet matches the simple rule R_4). In this case, our hardware classification might be incorrect, as rule $R_3 \in \mathscr{R}_C$ is more highly prioritized than rule $R_4 \in \mathscr{R}_S$, and R_3 might also match on the packet P. Thus, whenever matchID $\geq$ validID, we shunt the classified packet to the host for further processing, as described in the next section.

4.2 Software Filter

The task of the software filter running on the host computer (which is `netfilter` in our case) is to classify every shunted packet which cannot be handled solely in hardware. However, simply installing only the complex rule set $\mathscr{R}_C$ in the software filter is not enough, since for every shunted packet P, the hardware classification might have been correct. This is the case when P is not matched by any complex rule with a higher priority than the first matching simple rule. As a consequence, the software filter must be able to reproduce the hardware classification result iff the most highly prioritized matching rule is in $\mathscr{R}_S$ and not in $\mathscr{R}_C$. In the remainder of this section, we present three different strategies how the rule set in the software filter can be organized in order to achieve this goal.

Full set strategy.

The most straightforward way to setup the software filter, which we call the *full set strategy*, is to simply install the entire rule set $\mathscr{R}$. That way, forwarded packets will always traverse rules in the correct order until the first matching rule is found, as sketched in Figure 3a for the example rule set from Figure 1. This approach allows for quick rule updates, since only one rule in the rule set installed in the software filter has to be changed in addition to a possible update of the validID register on the FPGA. This strategy is simple, but comes at the cost of a major disadvantage: the software filter may process a large number of rules for every shunted packet, including simple rules. It thus repeats significant work already done in hardware. In contrast to the full parallel match in the hardware filter, this can be rather expensive, as the rules are processed linearly in most existing software packet filters.

Cut set strategy.

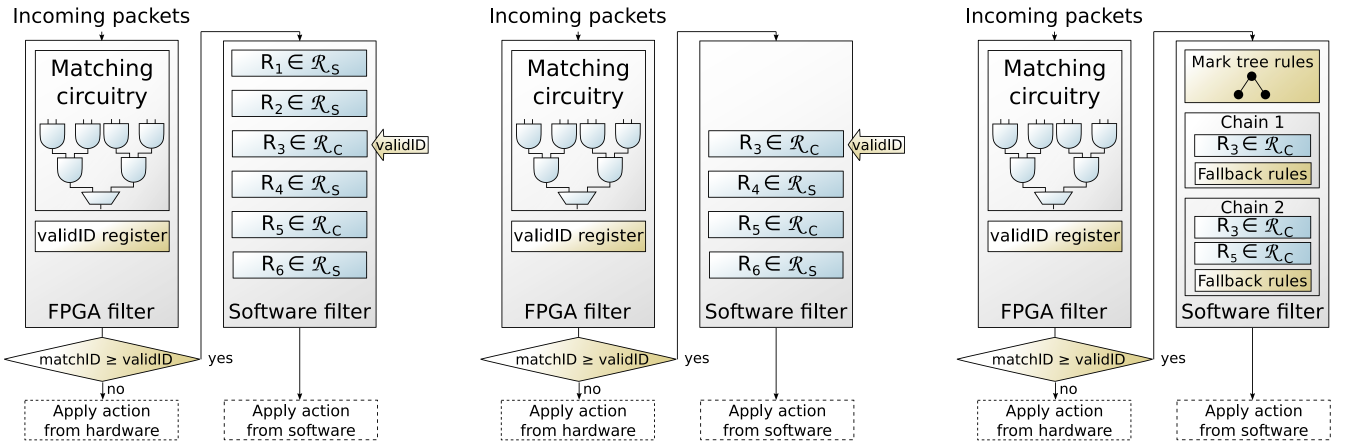
The amount of redundant work which is done in software for shunted packets can be reduced with a simple modification. We already know that no simple rule with an index less than validID can match a packet which has been forwarded to the software filter—otherwise, the packet would have been processed solely on the FPGA. For example, consider the rule set from Figure 1 and a packet P with matchID 3. As matchID is equal to validID (which is also 3 in the example), P will be forwarded to the software filter, which will superfluously once again test rules R_1 and R_2 against P. In order to avoid this potential extra work on the host system, the *cut set strategy* installs only those rules $R_i \in \mathscr{R}$ in the software filter where $i \geq$ validID, as sketched in Figure 3b.

In comparison to the full set strategy, the cut set strategy has higher rule update costs, as a potentially large number of rules must be inserted or removed from the software filter in case of an update. For instance, if the current validID is 300, and a rule is updated at position 100, then the 200 rules $R_i \in \mathscr{R}$ with $100 \leq i \leq 299$ must be inserted in the software filter. However, our evaluation demonstrates that the update effort can clearly pay off in terms of classification performance, as the software filter will test $|\mathscr{R}| - \text{validID} + 1$ rules at most, in contrast to the worst case of testing $|\mathscr{R}|$ rules in the full set strategy.

Interval strategy.

All strategies described so far implement rule sets in the software filter which are agnostic to the partial classification result tuple <matchID, $A_{\mathscr{R}_S,P}$> previously computed on the FPGA for every shunted packet P. This results in wasted effort on the software side and inflates the software-side rule set—also in case of the cut set strategy. To avoid the recomputation effort, the *interval strategy* relies on metadata handed over from the FPGA to the matching software when a packet is shunted, i. e., the match index and action tuple <matchID, $A_{\mathscr{R}_S,P}$>. Simply put, the goal of the interval strategy is that shunted packets should only be tested against a fraction of the complex rules $\mathscr{R}_C$ and none of the rules in $\mathscr{R}_S$ again in software.

The basic idea behind the interval strategy is that groups of consecutive simple rules $G_k = \{R_i, \ldots, R_{i+\alpha}\}$ in $\mathscr{R}$ can be mapped to intervals $I_k = [M(R_i), M(R_{i+\alpha})]$, where $M(R_i)$ is the index of the generated match unit for R_i. For instance, the simple rules from the example rule set in Figure 4 form three groups $G_1 = \{R_1, R_2\}, G_2 = \{R_4\}$, and $G_3 = \{R_6\}$, with the corresponding intervals $I_1 = [1,2], I_2 = [3,3]$, and $I_3 = [4,4]$, respectively. Each interval represents a range of matchIDs, which may be computed by the FPGA for an incoming packet P. It is important to note that, if P is shunted to the host, then the matchID computed on the FPGA falls into exactly one of these intervals. The interval strategy exploits this fact by precomputing the chain of complex rules C_k for every interval I_k, that could potentially contain a more highly prioritized matching rule for a packet P whose hardware-computed matchID falls into interval I_k (i. e., P matches a simple rule in group G_k). In the example shown in Figure 4, C_1 is empty, since there are no complex rules in $\mathscr{R}$ that are more highly prioritized than the simple rules R_1 and R_2. In contrast, $C_2 = \{R_3\}$, as the complex rule R_3 is more highly prioritized than the simple rule R_4 and thus could match on packets which have been assigned to R_4 by the FPGA. Similarly, C_3 would be set to $\{R_3, R_5\}$, as both complex rules R_3 and R_5 are more highly prioritized than the simple rule R_6.



(a) Full set strategy. (b) Cut set strategy. (c) Interval strategy.

Figure 3: Different strategies to implement the complex rule set $\mathcal{R}_A$ in the software filter.

Now, whenever a packet P is shunted to the host, the FPGA driver fetches the <matchID, $A_{\mathcal{R}_S,P}$> tuple from the hardware, which are 28 and 4 bit values, respectively. Then, the FPGA driver code on the host uses the matchID to perform a binary search over the precomputed intervals in order to find the index k of the interval I_k that contains the matchID. Before the actual `netfilter` packet classification starts, the index k as well as the hardware action code $A_{\mathcal{R}_S,P}$ are written to the most significant 28 and least significant 4 bits of the `netfilter mark` field, which is a 32 bit metadata field attached to the packet P.

These efforts are justified by the fact that `netfilter` supports tests on the `mark` field. We exploit this fact in order to achieve two goals: first, we want to limit the set of complex rules which must be tested in `netfilter` to only those which are more highly prioritized than the first matching simple rule. Second, we want to apply the hardware-computed action $A_{\mathcal{R}_S,P}$ in `netfilter` *without* the need to re-traverse any simple rule in software.

To this end, the rules which are installed in `netfilter` for the interval strategy are generated as follows: the `netfilter` rule set starts with a sequence of rules which implement a binary search over the interval index k encoded in the most significant 28 bits of the `mark` field. This is done in order to quickly locate the chain of relevant complex rules C_k during the matching process, as sketched in Figure 3c. The generated rule set also contains each chain C_i as a linear list, which contains the complex rules that are mapped to interval I_i. Finally, the last rule in every chain C_i ends with a jump to a small set of fallback rules (one for each possible action), which use the least significant four bits of the `mark` field in order to apply the action $A_{\mathcal{R}_S,P}$ to the shunted packet P if no complex rule matches.

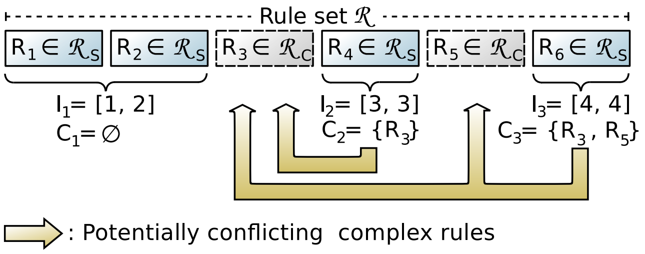


Figure 4: Intervals in the rule set $\mathcal{R}$.

In comparison to the full set and cut set strategies, the interval strategy requires more complex preprocessing in case of a rule update, as the intervals for the complex rules have to be re-computed and communicated to the hardware driver. Furthermore, the `net-filter` binary search tree encoded in the filter rules must be regenerated. However, this strategy provides the best classification performance in software, as the number of traversed rules for each shunted packet P can be orders of magnitude smaller than in the full set and cut set strategies, as indicated by our evaluation. Furthermore, this approach does not require a change of the `netfilter` source code in order to use the hardware-computed matching information. Instead, we completely rely on existing `netfilter` match functionality in order to accelerate the software match process.

5. SYSTEM ARCHITECTURE

We implemented the hybrid algorithm on our HyPaFilter system, which consists of two functional units. One part is a standard host system, used to run the software firewall and the toolchain for managing the system. This can even be an already existing firewall appliance which should be upgraded in terms of performance. This system is extended by the second part, a general purpose FPGA

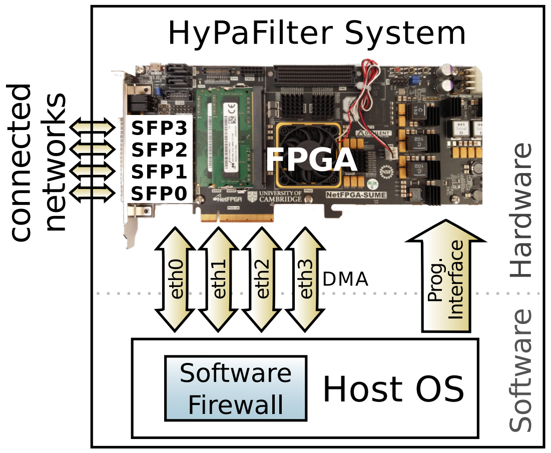


Figure 5: Proposed structure of a HyPaFilter system. The host can be any COTS system capable of carrying the additional FPGA NIC.

addon card, as shown in Figure 5. These units must provide a sufficient communication path for transferring data and settings between them.

FPGA Networking Card

This card is a suitable FPGA platform which can provide the required interfaces to both communicate with external Ethernet networks as well as acting as a regular network interface card in regard to the host system. Both FPGA plug-in cards we used during our evaluation – the VC709 [5] and the NetFPGA SUME – have proven to be suitable. They provide multiple network ports and can be plugged into a COTS system via PCI Express (PCIe). This card acts as the primary network interface connected to both internal (e. g., LAN) and external network (e. g., Internet). The hardware based filtering is handled exclusively on the FPGA on the card.

Host System

The host system carries the FPGA NIC and communicates with it, for example via PCIe. The host runs the operating system where the back-end `netfilter` with `iptables` is installed, supplies the tools to configure the FPGA and provides a user interface for administrating HyPaFilter.

These two units are connected through several communication channels. For quick and simple settings, the host system is able to set and read predefined 32 bit registers on the FPGA. Network traffic between FPGA and host is handled through *direct memory access (DMA)*. On the host side, a driver provides the functionality and interfaces so that the operating system can access the FPGA like a regular NIC. This is important since we do not want to rely on non-standard customizations to `netfilter` or other core components for HyPaFilter to work. By using a programming interface, the configuration of the FPGA can be updated. A software toolchain of the FPGA vendor, in our case Xilinx Vivado, is used to generate the FPGA configuration based on a given rule set. For convenience, it is also installed on the host.

Operation

Packets received from any connected network are first matched against the rules implemented on the FPGA. Based on its decision and the validID, packets are either dropped, forwarded directly (without interaction of the host system), or shunted to the host for further processing. The host can send packets through the supplied driver interface, which applies to both packets shunted through the software firewall and packets generated by the host itself. These packets are directly forwarded by the FPGA to the corresponding network interface. The flow of packets through this structure is visualized in Figure 6.

The two reasons for packets being shunted are a) not synthesizable rules appearing at positions in the rule set that would otherwise be forwarded, or b) an update to the rule set at these positions. For updating, this technique copes with the problem that changes to logic-level optimized rule sets cannot be selectively integrated. While a new filter's source code can be quickly generated, the synthesis and implementation of the new FPGA bitfile requires a significant amount of time – in our test setup about 45 minutes.

The information about which index in the rule set matched on the hardware is not discarded, as it is needed to remove the redundancy by applying the interval strategy. For operation, the administrator uses a central management tool. In our implementation, it is a Python command line interface. The general workflow for using HyPaFilter is shown in Figure 7.

5.1 Rule Set-Specialized Hardware Filter

The dataflow through the FPGA can be shown in two layers. The underlying structure for general networking and communication tasks is based on the NetFPGA SUME pipeline. The actual core which is responsible for filtering is embedded into this pipeline and connected via the AXI4 stream protocol as shown in Figure 8.

Internally, the HyPaFilter core uses a data bus width of 512 bits, with the pipeline running at 180 MHz. The theoretically achievable throughput of 85.83 Gbit/s is therefore enough to fully saturate all four 10 Gbit/s Ethernet ports. The NetFPGA SUME currently uses a bus width of only 256 bits which are converted before and after the hardware core. Packets coming into the hardware core are first distributed (cloned) into a classification path and a data path, with the latter being a simple FIFO queue of 64 kB. In the classification path, the *Header Parser* extracts relevant information from incoming packets. For a versatile operation, the header parser should take care of the data alignment due to VLAN tags or various header lengths. It is therefore implemented as a multi-stage non-blocking pipeline architecture. The preprocessed data is forwarded to the filtering module, which is automatically generated by the management toolchain. After the classification, the decision is forwarded to the *Output Processing*, where the determined action is executed: DROP (read from FIFO and discard), FORWARD, or SHUNT by adapting the output port field in the packet's metadata. The register interface can be accessed from the host directly via PCIe. Figure 9 shows the described parts in the module. The match logic is able to classify packets in constant time. Hence, a reader might note that the separation into data and classification path yields no advantages in terms of maximum throughput. However, as we aim to support more complex decisions in hardware in our future work, this structure allows for more flexible development. Since the hardware filtering logic contains no components that could cause a congestion, it is clear that the HyPaFilter hardware core is never the limiting

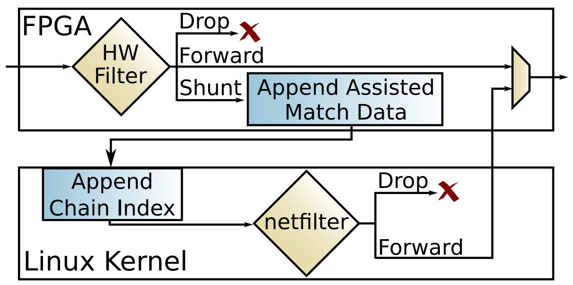


Figure 6: Flow of packets through the HyPaFilter system.

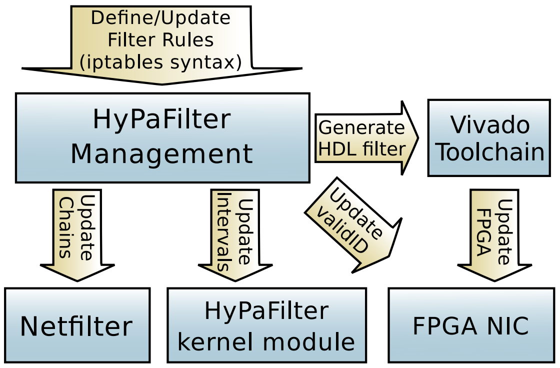


Figure 7: HyPaFilter workflow with the central management tool.

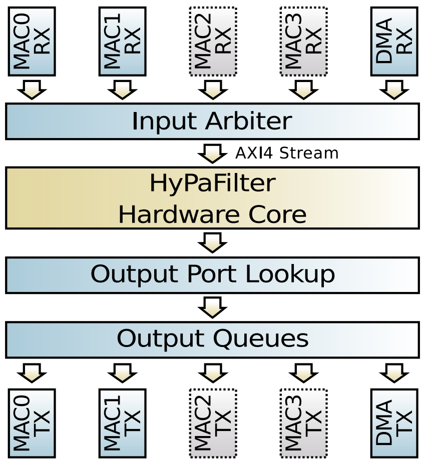


Figure 8: Simplified dataflow structure of the NetFPGA SUME. The dashed elements are available, but not used in our evaluation.

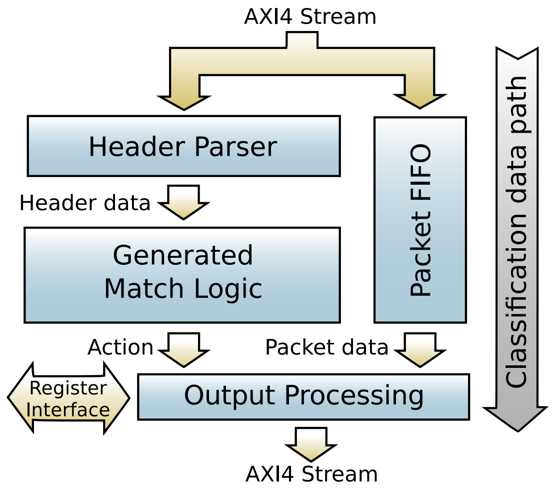


Figure 9: Dataflow inside the HyPaFilter hardware core.

factor for raw data throughput in this setup. The hardware filter core is able to extract and classify incoming packets against a variety of parameters like IPv4 address, IPv6 address, protocol type (UDP, TCP, ICMP, ARP), media access control (MAC) address, TCP/UDP ports, and several flags.

Previous work has shown that the resource utilization of typical rule sets on FPGAs can be significantly reduced by including the actual rule set in the logic optimization process, rather than using a generalized filtering logic [15, 17]. As firewall rule sets in general are not static, an FPGA's reconfigurability allows to exploit this potential in practical applications. Thus, in our HyPaFilter prototype, we combine such a rule set tailored hardware filter with a generic software filter residing on the host. However, the proposed HyPaFilter approach does not strictly rely on this type of specialized hardware filter and could also utilize another hardware matcher (e. g., a TCAM), as long as the driver interface which provides the matching information is maintained.

5.2 Software Filter for Incremental Updates and Complex Rules

Although hardware-based parallel packet filtering, as explained in the previous section, can achieve high throughputs of 40 Gbps or higher, it suffers from two fundamental drawbacks: on the one hand, incremental updates to the rule set are time-consuming. On the other hand, the flexibility of hardware-based filters is much more constrained than that of software-based filters. Most hardware-based approaches are restricted to simple equality, range, or prefix checks on incoming header fields [17, 19, 20]. In contrast, many software-based packet filters, such as `netfilter` [3] or `pf` [4], support much more complex matching criteria. Examples are the state of the flow that corresponds to the examined packet, probability-based matches for rate limiting, or even arbitrary Berkeley Packet Filter (BPF) expressions, to name only a few. Furthermore, new filtering functionality like, e. g., support for a new protocol or the addition of further options for an existing one, can be added relatively quickly. However, a full-fledged in-circuit implementation which supports all of these features is notoriously difficult, as dedicated hardware it typically optimized for a very limited functionality.

In order to achieve the advantages of both software- and hardware-based packet processing, i. e., (1) high throughput, (2) fast incremental updates, and (3) powerful rule set semantics, we combine fast specialized matching circuitry with the flexibility and expressiveness of a CPU-based packet filter, namely `netfilter`.

6. EVALUATION

In our evaluation, we focus on three of the most important performance metrics for packet classification architectures: packet rate, rule set update latencies and consumption of resources. This stands in contrast to raw data throughput measurements, which are more targeted at the data flow structure. Therefore, we conducted the following experiments:

- determining the maximum number of rules which can be fitted onto the FPGA,
- measuring the maximum packet rate of the NetFPGA SUME architecture,
- measuring the performance of the HyPaFilter system and comparing the impact of rule updates using different strategies,
- measuring the network latency,
- measuring delays of the update process and number of rules and
- comparing against a commercial OpenFlow software-defined networking (SDN) setup.

To generate a high workload on the classification engine, we used small packets at a high rate. Packets carry just five arbitrarily chosen bytes as payload. For our evaluation, we set up a typical bridging firewall scenario as shown in Figure 10.

Traffic is generated and received by two dedicated machines, based on Intel Core i7 960 with a dual-port 10 Gbit/s NIC. The hosts run Ubuntu Linux. We generated rule sets and traffic using the ClassBench suite [33], which is widely used in this context. The system is easily capable of saturating the connected networks with traffic. These sender and receiver hosts are connected to the HyPaFilter system via optical fibre. We counted the number of packets received by the MAC-Core `MAC0` on the NetFPGA and those arriving on the network interface of the receiver. Further network connections between the systems to remotely start the test cycles and collect the results are not shown.

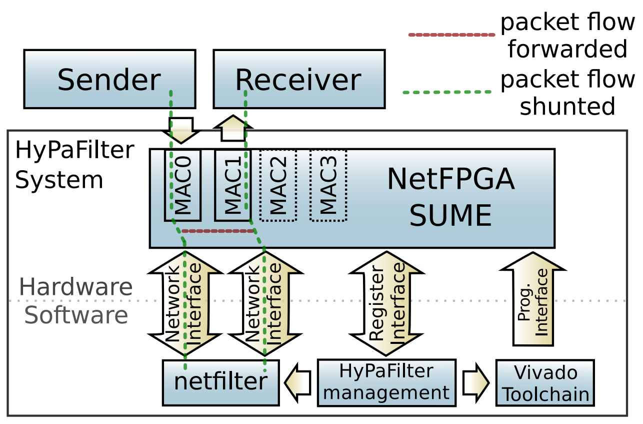


Figure 10: Evaluation setup showing the relevant components. Traffic is generated on the sender and directed through the bridging HyPaFilter firewall.

The HyPaFilter system consists of the following relevant components: Intel Xeon E5-1650 v3 based host, Intel 82599ES dual-port 10 Gbit/s NIC, NetFPGA SUME PCIe card, Ubuntu Linux, `netfilter` framework and `iptables` v1.4.21, Xilinx Vivado 2014.4, as well as the HyPaFilter management tools. The hardware filter core is integrated into a modified data pipeline based on the reference NIC project of the NetFPGA SUME release 1.0.0.

6.1 Test Rule Sets

To evaluate the classification performance under replicable conditions, we generated our test rule sets with ClassBench [33]. The number of rules we could fit onto the FPGA was limited by the timing constraints and resulted in a maximum of 1100 rules. For evaluating our classification algorithm and strategy, we created three different UDP rule sets `acl1k1`, `fw1k1` and `ipc1k1`, with all rules applying the action ACCEPT. This way, the number of dropped packets can be regarded as the packet loss solely due to the architecture.

ClassBench's *trace_generator* was used to generate trace files corresponding to the rule sets. We wrote a C program for generating and transmitting the test packet stream from such a trace file, because no sufficiently fast solution could be found for this task under the given conditions. The rule sets as well as the traces we used are publicly available at [14].

For each test, the rule set was translated by the HyPaFilter management tool, integrated in the NetFPGA SUME pipeline and afterwards synthesized and implemented into an FPGA configuration bitfile. Table 1 shows the resource utilization of the FPGA configuration for a Virtex 7 690T. The relevant parameters are usage of flip-flops (FF), look-up tables (LUT), LUTs used as memory elements (Memory LUT), and block random access memory (BRAM). Differences can be caused by the different rule sets and heuristic algorithms used during the implementation process in Vivado.

To measure the impact of changes or occurrences of complex rules to the the rule set we added in each test new rules to certain positions, starting from the end of the rule set. We used string matching rules for this purpose:

```
-m string --algo bm --string BAD -m statistic\
 --mode random --probability 0.99
```

These complex rules are especially interesting as the use of the full capabilities of `netfilter` is one of the core features of HyPaFilter.

Resource	acl1k	fw1k	ipc1k
FF	9.12%/0.86%	9.12%/0.86%	8.26%/0.86%
LUT	15.07%/1.69%	16.22%/2.85%	13.38%/1.83%
Memory LUT	1.07%/0.01%	1.10%/0.01%	1.10%/0.01%
BRAM	16.73%/2.72%	16.73%/2.72%	14.01%/2.72%

Table 1: FPGA resource utilization overall/HyPaFilter core with different rule sets.

6.2 Architecture Packet Rate

Forwarding packets directly in hardware provides the lowest latency and highest packet rate. Therefore, the first experiment was used to measure the maximum packet rate dependent on the percentage of packets bridged to the software. We will later compare the packet rate of the different strategies against these values.

As the generated packets by the sender will match the rule set at certain positions with a predefined distribution, the hardware filter was used in combination with the validID to shunt parts of the traffic to the software. There were no rules loaded into `netfilter`, all incoming packets are forwarded by the Linux bridge.

We compared the number of ingress packets vs. packets received at the receiver, which is the inverse of packets being dropped in the firewall. Each data point shows the average packet rate of ten 20 second test runs, with distribution of the workload being set by the validID as the variable parameter. The average number of ingress packets arriving at the HyPaFilter network interface in each test run before any classification is 2.3 million in 20 s. Figure 11 shows the percentage of packets arriving at the receiver, the standard deviation was too small to be visible in the plot. The architecture of the host system and the NetFPGA SUME used as a simple NIC is only capable of processing on average 6.4% of the packets which arrive at the NetFPGA input interface. Increasing the validID and therefore reducing the fraction of shunted packets increases the overall amount close to 100% when all packets are directly forwarded by the FPGA NIC.

6.3 Strategy Comparison

To evaluate the performance of the different strategies described in Section 4.2, the setup described in Section 6.2 was now adapted to use firewalling. Starting with a hardware only scenario, we measured the impact of rule insertions without updating the hardware filter definition. This subsequently causes an increasing amount of packets to be shunted to the software firewall.

We conducted our experiments by following a certain test cycle for all three strategies and repeated them for each of the three sam-

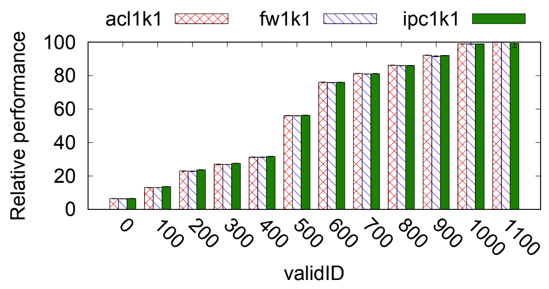


Figure 11: Packet rate of the underlying architecture as a function of the fraction of packets forwarded through hardware. At validID = 0 all packets are shunted to the software, while at validID = 1100 all packets are forwarded.

ple rule sets acl1k1, ipc1k1, and fw1k1. During the test cycle, we measured the average packet throughput rate after iteration n following these steps:

1) implement the sample rule set onto the FPGA and set validID to match everything in hardware

2) for test run $n = 0$ insert a new rule at position $P_0 = 1100$ and set validID $= P_0$

3) for test run $n+1$ insert a new rule at position $P_{n+1} = 1100 - 100(n+1)$ and set validID $= P_{n+1}$

4) repeat last step until $P_n = 0$.

For the first part of this test, we used the full set strategy and loaded the complete rule set in netfilter. As mentioned in Section 4.2, this leads to a high redundancy in the matching. For example, an update at position index $P = 500$ sets validID $= 500$, therefore all packets with matchID ≥ 500 will be shunted. These packets will, however, never match the first 500 rules in netfilter (counting from index zero), making them essentially useless.

For large validIDs, the tests confirmed the assumption that significant performance gains can already be achieved by removing the parts of the netfilter rule set that correspond to matchID $<$ validID (cut set strategy). The interval strategy proved to be a useful additional measure when the amount of packets shunted to software is large or the hardware and software rule sets intersect at a high priority. Figures 12a, 12b, and 12c show the speedup as a factor of the received packet rate compared to using netfilter without any hardware acceleration and string matching rules for insertion. The error bars show the standard deviation.

For the full set strategy, it can be clearly seen that the packet rate behaves non-monotonic and dips near validID $= 800$. This can be explained by the combination of two contrary effects: first, with the validID decreasing, an increasing amount of shunted packets causes the software performance to reach its limit. Second, with the validID increasing, the packets that are shunted will only match a smaller and smaller part at the end of the software rule set. This means that the average number of rules traversed by the packets will also increase, regardless of the constant total number of software rules.

To get a better overview of the performance increase by applying the improved strategies, their packet rate has to be compared against the full set strategy. This relative speedup, again for using complex string matching rules in the insertion process, can be seen in Figures 13a, 13b, and 13c. Large gains of performance of both the cut set and interval strategy can be seen due to the reduction of the long path effect which is causing the equivalent dip for with the full set strategy near validID $= 800$. With an increased amount of complex software rules with high priority (low validID), the advantage of our hardware assisted binary search algorithm used in the interval strategy becomes clear.

6.4 Network Latency

While the packet classification rate is the most interesting parameter to measure for evaluation, the additional latency which is added by security appliances can be a major issue for certain applications, e. g., in data centers. Our network latency measurement splits into two parts: the additional delay of the HyPaFilter hardware core in the NetFPGA SUME pipeline, and the actual delay which can be seen on network packets.

The internal additional delay in the FPGA could be determined in the Vivado Simulator and is fully deterministic at 24 clock cycles. With a clock rate of 180 MHz, the core therefore adds an additional delay of 133 ns compared to the NetFPGA SUME in NIC operation.

In order to check for the overall network latency imposed by the HyPaFilter system, the round-trip time (RTT) was measured with ping, sending 50 packets per test. While a direct connection between sender and receiver (without the NetFPGA SUME) shows a one way latency of 51 µs ($\sigma = 3.2$ µs), with the HyPaFilter system present and forwarded packets only we saw a tolerable increase to 52 µs ($\sigma = 5.4$ µs). For packets shunted through software without any firewall interaction it further increased to 73 µs ($\sigma = 3.5$ µs), the highest average delay of 96 µs ($\sigma = 7$ µs) occurred with shunted packets and an active software rule set of 1100 rules loaded into netfilter.

With the limitation of the uncertainty of the measurement method, the results show that our hardware filtering algorithm is suitable for low latency requirements.

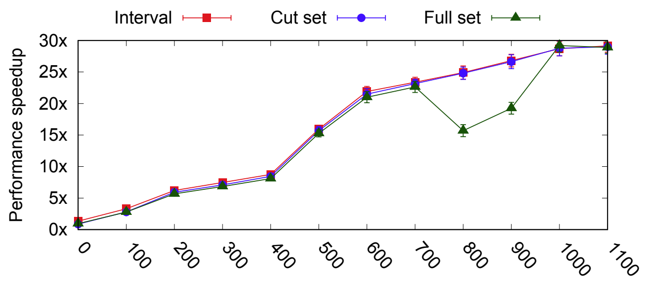
6.5 Rule Set Parameters

The strong influence of the number of rules in a software firewall to its classification performance leads to the question how many rules are loaded into the firewall for the three different strategies after applying the update cycle. These numbers were determined by exporting the rules with iptables-save and counting the correspondent lines. As HyPaFilter uses a binary tree searching algorithm, we also evaluated the worst case path length, i. e., the highest number of potentially traversed rules for incoming packets. Figure 14 gives an overview over the actual number of rules which are active in netfilter for different strategies, as well as the number of rules which have to be evaluated in the worst case.

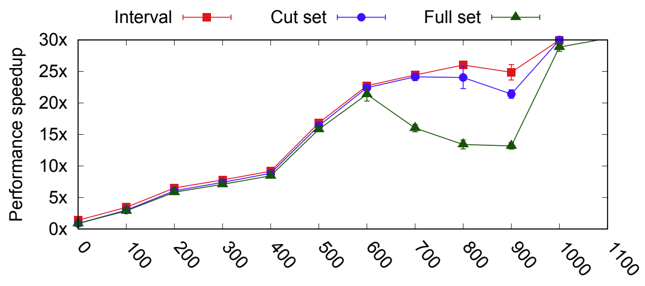
The synthesis and implementation process that is used to generate the new bitfile with one of the test rule sets requires about 45 minutes on the described HyPaFilter evaluation host, using Vivado 2014.4. The Xilinx tool xmd, which is used to configure the FPGA with this bitfile via the programming interface finishes in 17.38 s. During this time, the network is interrupted. In our test cycles, no hardware update was required to reach the stated results.

6.6 Update Delay

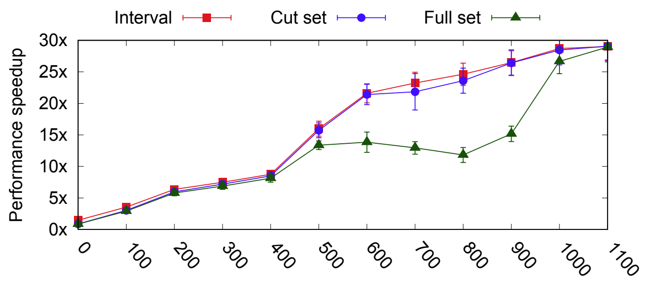
Another interesting parameter is the time required for different types of updates required for different strategies. We therefore measured the time for inserting rules, updating the validID register in the FPGA and uploading a new configuration to the FPGA. According to our test cycle, the delays for the insertion were determined for consecutive insertions of rules at certain positions, i. e., the test at validID = 900 is executed with the assumption of rules preliminary inserted at position 1100 and 1000. The update process involves different operations for each strategy: *a*) for the full set strategy, inserting a single rule with iptables and setting validID *b*) for the cut set strategy, truncating the rule set, inserting and loading this set with iptables-restore, setting validID *c*) for the interval strategy, calculating intervals, inserting the chained rule set with iptables-restore, updating the driver and setting validID. Figure 15 shows the result of this test, as an average of 10 test cycles for each data point. Setting the validID register on the FPGA alone takes 1 µs. The measured time confirms our assumptions about the cost for rule insertions (see Section 4.2). However, even the most demanding updates of the interval strategy could be carried out with a tolerable delay.



(a) `acl1k1` rule set.

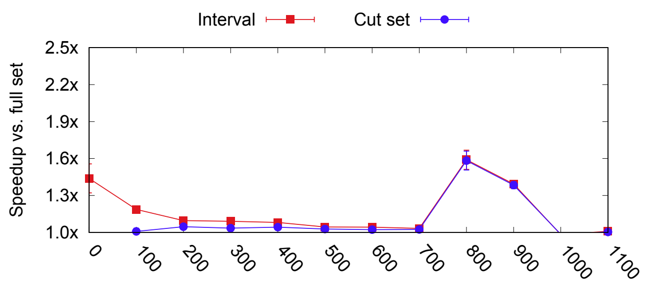


(b) `fw1k1` rule set.

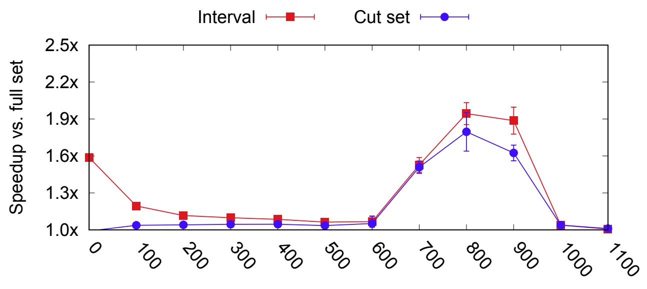


(c) `ipc1k1` rule set.

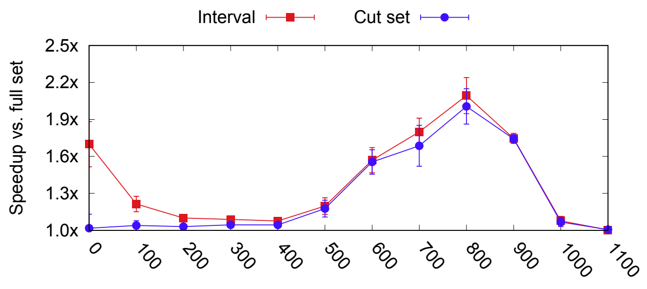
Figure 12: Speedup of HyPaFilter with complex inserted rules and different strategies over a software-only `netfilter` setup.



(a) `acl1k1` rule set.



(b) `fw1k1` rule set.



(c) `ipc1k1` rule set.

Figure 13: Speedup of the enhanced strategies with complex inserted rules, compared to the full set strategy.

6.7 OpenFlow SDN

The logical division into a hardware filtering unit with a software backend is in several aspects similar to the concept of an SDN. In a typical SDN switch setup, a controller would place *flows* dynamically into the hardware, allowing fast transmission of matching packets. For a firewall application, the requirements are more complicated than for a simple switch. Therefore, a controller with firewalling functionality was required.

We replaced the NetFPGA with a Quanta Computer LB8 48-port SDN [27] switch running PicOS. For a fair comparison, we used a publicly available and stable controller, OpenIRIS v2.2.1 [9], which was installed on the HyPaFilter host system. The OpenFlow 1.3 protocol is used for the communication with the switch. OpenIRIS includes a firewall module which can be controlled via the REST API [29]. The number of incoming packets was determined through the web interface of OpenIRIS.

We noticed several issues of the OpenIRIS firewall module during our evaluation:

- Rules could not be added to certain positions. Although it is possible to define a `ruleid`, the parameter seems to be ignored and replaced by a random value which is not related to the actual (logical) position of the rule. New rules are always prepended to the current rule set.
- The source and destination port could not be specified as a range.
- The port fields could not be set to values higher than 32767, obviously due to sign conversion problems.

The update process therefore has to be carried out by first deleting all rules and then adding all rules of our rule set in reverse order. Loading 1100 rules into the module with the REST API takes 3.9 s on average. During our evaluation we found out that the firewall only placed flows into the hardware for ICMP ping packets and established TCP sessions. Our test data (UDP packets), as well as generic TCP packets do not trigger this mechanism, therefore forcing each packet into the slow path to the controller. Although not configurable, this behaviour may be a protection against SYN flooding of the flow table, i. e. purposefully trying to fill the flow table with useless entries. We concluded that due to these effects, a fair comparison against our setup was not possible. Further investigation of these issues is out of the scope of this paper.

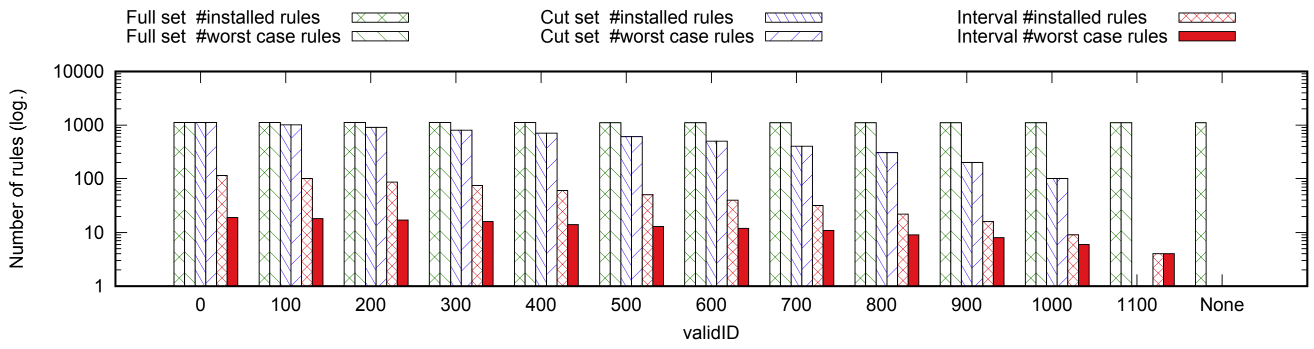


Figure 14: Number of installed/worst case traversed rules for different strategies.

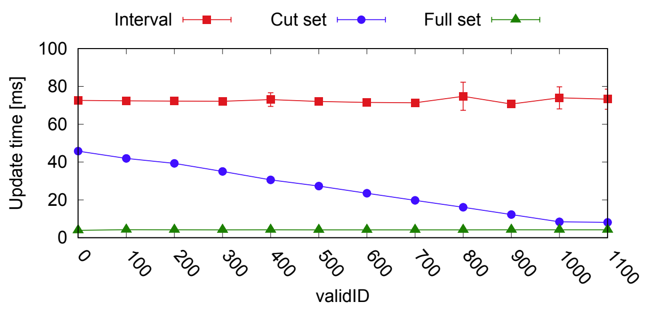


Figure 15: Update latency of different strategies.

7. CONCLUSION

In this work we introduce HyPaFilter, a hybrid packet classification approach which combines the parallel matching capabilities of specialized hardware with the extensive matching semantics of widely used software packet filters. HyPaFilter accomplishes this task by partitioning the implemented packet processing policy into a simple and a complex part, where the simple part can be handled directly in hardware and the complex part is installed in the software filter. Incoming network packets are first processed in hardware and are shunted to the software filter only in the case where complex processing is required. We present a novel strategy how the software-implemented part of the rule set can be organized in order to reuse matching information from the hardware. This strategy can be used on top of `netfilter` and does not require changes of the `netfilter` source code. The actual hardware filter is not limited to our evaluation example, it can be any suitable algorithm which provides the match index. Our evaluation of HyPaFilter based on a combination of a NetFPGA SUME device and a Linux host system demonstrates significant increases in the achievable throughput over a software-only approach, even with rule set constellations where the majority of incoming packets must be processed in software.

Possible future work includes the integration of a fast dynamic state table on the FPGA which can be used to directly handle stateful rules in hardware. This way, stateful rules could be handled entirely in hardware and would not require the bridging of packets. Ideally, this table would still allow access from the host so that the software is able to shunt acknowledged sessions in hardware.

8. ACKNOWLEDGEMENTS

The authors would like to acknowledge the support of the German Federal Ministry for Economic Affairs and Energy, whose grant allowed us to conduct major parts of the fundamental research. Furthermore, we thank the Computer Laboratory of the University of Cambridge for providing the NetFPGA SUME project as a suitable research platform. This work was, in-part, supported by the European Union Horizon 2020 SSICLOPS project (grant agreement 644866).

9. REFERENCES

[1] genugate firewall. www.genua.de/en/solutions/high-resistance-firewall-genugate.html. Last access: Novemver 6, 2015.

[2] IPFW firewall. https://www.freebsd.org/cgi/man.cgi?ipfw. Last access: September 28, 2015.

[3] The netfilter.org project. www.netfilter.org. Last access: September 28, 2015.

[4] OpenBSD packet filter. http://www.openbsd.org/faq/pf/. Last access: September 28, 2015.

[5] VC709 Evaluation Board for the Virtex-7 FPGA. http://www.xilinx.com/support/documentation/boards_and_kits/vc709/ug887-vc709-eval-board-v7-fpga.pdf. Last access: September 29, 2015.

[6] K. Accardi, T. Bock, F. Hady, and J. Krueger. Network processor acceleration for a linux* netfilter firewall. In *ANCS '05: Proceedings of the 2005 Symposium on Architectures for Networking and Communication Systems*, pages 115–123, Oct. 2005.

[7] F. Baboescu and G. Varghese. Scalable packet classification. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 199–210, Aug. 2001.

[8] P. Bosshart et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM '13: Proceedings of the 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 99–110, Aug. 2013.

[9] L. Byungjoon, J. Shin, and S. H. Park. Openflow Controller by ETRI. https://github.com/openiris/IRIS, 2015. Last access: Oct 25, 2015.

[10] M.-S. Chen, M.-Y. Liao, P.-W. Tsai, M.-Y. Luo, C.-S. Yang, and C. E. Yeh. Using netfpga to offload linux netfilter firewall. In *2nd North American NetFPGA Developers Workshop*, 2010.

[11] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang. Parasplit: A scalable architecture on FPGA for terabit packet classification. In *HOTI '12: Proceedings of the 20th*

Symposium on High Performance Interconnects, pages 1–8, Aug. 2012.

[12] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *HOTI '99: Proceedings of the 7th Symposium on High Performance Interconnects*, pages 34–41, Aug. 1999.

[13] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network: The Magazine of Global Internetworking*, 15(2):24–32, Mar. 2001.

[14] S. Hager. Hypaf rule sets. http://hardfire.de/rule-sets.

[15] S. Hager, D. Bendyk, and B. Scheuermann. Partial reconfiguration and specialized circuitry for flexible FPGA-based packet processing. In *ReConFig '15: 2015 International Conference on ReConFigurable Computing and FPGAs*, Dec. 2015. to appear.

[16] S. Hager, S. Selent, and B. Scheuermann. Trees in the list: Accelerating list-based packet classification through controlled rule set expansion. In *CoNEXT '14: Proceedings of the 10th International Conference on Emerging Networking Experiments and Technologies*, pages 101–107, Dec. 2014.

[17] S. Hager, F. Winkler, B. Scheuermann, and K. Reinhardt. MPFC: Massively parallel firewall circuits. In *LCN '14: Proceedings of the 39th Annual IEEE International Conference on Local Computer Networks*, pages 305–313, Sept. 2014.

[18] W. Jiang and V. Prasanna. Large-scale wire-speed packet classification on FPGAs. In *FPGA '09: Proceedings of the ACM/SIGDA 17th International Symposium on Field Programmable Gate Arrays*, pages 219–228, Feb. 2009.

[19] W. Jiang and V. K. Prasanna. Field-split parallel architecture for high performance multi-match packet classification using FPGAs. In *SPAA '09: Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 188–196, Aug. 2009.

[20] W. Jiang and V. K. Prasanna. A FPGA-based parallel architecture for scalable high-speed packet classification. In *ASAP '09: Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 24–31, July 2009.

[21] W. Jiang and V. K. Prasanna. Large-scale wire-speed packet classification on FPGAs. In *FPGA '09: Proceedings of the ACM/SIGDA 17th International Symposium on Field Programmable Gate Arrays*, pages 219–228, Feb. 2009.

[22] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM '98: Proceedings of the 1998 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 203–214, Aug. 1998.

[23] A. Liu, E. Torng, and C. Meiners. Firewall compressor: An algorithm for minimizing firewall policies. In *INFOCOM '08: Proceedings of the 27th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 176–180, Apr. 2008.

[24] D. Liu, B. Hua, X. Hu, and X. Tang. High-performance packet classification algorithm for many-core and multithreaded network processor. In *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded System*, pages 334–344, Oct. 2006.

[25] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA '14: Proceedings of the 41st International Symposium on Computer Architecture*, pages 13–24, June 2014.

[26] Y. Qi, B. Xu, F. He, B. Yang, J. Yu, and J. Li. Towards high-performance flow-level packet processing on multi-core network processors. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architectures for Networking and Communication Systems*, pages 17–26, Dec. 2007.

[27] Quanta Computer Inc. QuantaMesh 5000 Series BMS T5016-LB8D. http://www.qct.io/Product/Networking/Bare-Metal-Switch/QuantaMesh-BMS-T5016-LB8D-p58c77c75c159, 2015. Last access: Oct 25, 2015.

[28] D. Qunfeng, S. Banerjee, J. Wang, and D. Agrawal. Wire speed packet classification without TCAMs: A few more registers (and a bit of logic) are enough. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 253–264, June 2007.

[29] J. Shin. REST API List of OFMFirewall. https://github.com/openiris/IRIS/wiki/REST-API-List-of-OFMFirewall, 2014. Last access: Oct 25, 2015.

[30] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 213–224, Aug. 2003.

[31] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *SIGCOMM '99: Proceedings of the 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 135–146, Aug. 1999.

[32] D. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, Sept. 2005.

[33] D. Taylor and J. Turner. Classbench: a packet classification benchmark. *IEEE/ACM Transactions on Networking*, 15(3):499–511, June 2007.

[34] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. Efficuts: Optimizing packet classification for memory and throughput. In *SIGCOMM '10: Proceedings of the 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 207–218, Aug. 2010.

[35] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. Multi-layer packet classification with graphics processing units. In *CoNEXT '14: Proceedings of the 10th International Conference on Emerging Networking Experiments and Technologies*, pages 109–120, Dec. 2014.

[36] N. Weaver, V. Paxson, and J. Gonzalez. The shunt: an FPGA-based accelerator for network intrusion prevention. In *FPGA '07: Proceedings of the ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, pages 199–206, Feb. 2007.

[37] N. Zilberman, Y. Audzevich, G. Covington, and A. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *Micro, IEEE*, 34(5):32–41, Sept 2014.