Packet monitoring at high speed with FFPF (Technical Report)

Herbert Bos and Georgios Portokalidis
LIACS
Universiteit Leiden
Leiden, The Netherlands
{herbertb,gportoka}@liacs.nl

Abstract

The fairly fast packet filter (FFPF) is an approach to network packet processing that adds many new features to existing filtering solutions like BPF. FFPF is designed for high speed by pushing computationally intensive tasks to the kernel or even network processors and by minimising packet copying. By providing both a richer programming language and explicit extensibility, it is also considerably more flexible than existing approaches. FFPF provides a complete solution for network monitoring that caters to all applications available today. Using its extensibility, the language can even be used as a meta-filter to 'script' together filters from other approaches, such as BPF. If In this paper, we present the FFPF architecture as well as the current implementation.

1 Introduction

Most network monitoring tools in use today were designed for low-speed networks under the assumption that computing speed compares favourably to network speed. In such environments, the costs of copying packets to user space prior to processing them, are acceptable. In today's networks, this assumption is no longer true. The number of cycles that is available to process a packet before the next one arrives (the cycle budget) is minimal. Moreover, the processing requirements are increasing. Consider the following monitoring applications:

1. An intrusion detection system (IDS) that checks the payload of every packet for the occurrence of worm signatures [**?**].
2. A 'Coralreef' application that keeps track of the ten most active flows [**?**].
3. A tool interested in monitoring flows for which the port numbers are not known *a priori*. Such flows are found, for example, in peer-to-peer and H.323 multimedia flows where the control channels use well-known port numbers, while the data transfer takes place on dynamically assigned ports [**?**].

In high-speed networks, none of these applications are catered for in the kernel in a satisfactory manner by existing solutions such as BPF [**?**]. In our view, they require a rethinking of the way packets are handled in the monitoring platform. Specifically, to deal with speed most of the processing should take place at the lowest level in the processing hierarchy, i.e., the kernel or even the hardware [**?**]. Similarly, much more flexibility is required to support such applications.

In this paper, we discuss the implementation of the fairly fast packet filter (FFPF). FFPF provides a complete solution for filtering and classification at high speeds, either in the kernel or on a network processor, while minimising copying. As the name implies, speed is not the only goal for FFPF. Although it needs to be fast and compare favourably to existing solutions, flexibility is at least as important. Fo this reason, FFPF is explicitly extensible with native code and facilitates packet processing at the lowest levels of the processing hierarchy. FFPF is designed as an alternative to kernel packet filters such as CSPF [**?**], BPF [**?**], mmdump [**?**], and xPF [**?**]. All of these approaches rely on copying many packets to userspace for complex processing (such as scanning the packets for intrusion attempts). On the other hand, FFPF is not meant to compete with monitoring suites like Coralreef that operate at a higher level and provide libraries, applications and drivers to analyse data [**?**]. Also, unlike MPF [**?**], Pathfinder [**?**], DPF [**?**] and BPF+ [**?**], the goal of this research is not to optimise filter expres-

sions. Indeed, our filter interpreter is probably inferior to any of these approaches. Instead, the aim of FFPF is to implement a complete, fast, and safe filtering architecture that caters to the needs of most if not all monitoring applications in existence today and to provide extensibility for future applications. In addition, FFPF should be capable of operating at high speeds with a minimum of copying. Moreover, FFPF provides so much flexibility that all of the above approaches can be *used* by making them available as *external functions* (discussed in Section 3.3.1). As the BSD Packet Filter is the most popular among the filtering approaches, most of our comparisons will be against BPF.

Like BPF, FFPF allows users to process network packets in the kernel or even on the network card. The contribution of this paper is that filtering is extended with many new capabilities.

1. FFPF removes all copies from kernel or hardware to userspace. All packets remain in kernel/hardware buffers from where they can be accessed directly by the applications.

2. Instead of simple filters, FFPF also allows full packet processing programs to be loaded in the kernel. Filters, classifiers, and other applications are equally easy to implement.

3. FFPF provides these programs with persistent storage. The storage can be used to keep flow-specific state, such as counters.

4. Users and administrators alike are able to extend the system's API with native functions that can be called from the kernel-based user code. Native functions are able to interact with kernel code and the userspace application, e.g. via the flow's persistent storage. They are used to implement computationally intensive functionality, that would be prohibitively expensive to perform in interpreted programs.

5. FFPF supports multiple 'security groups'. A security group consists of a number of applications that have the same access rights to network packets. Members in a group are allowed to read all packets received by other members of the group, but not those of other groups.

6. The expressions injected by the user in the kernel may modify themselves. This is particularly useful in case the ports that are used by applications are not known *a priori*.

7. Fine-grained admission control is added to prevent, for instance, unauthorised access to security groups, illegal invocation of external functions, or 'stupid filters'.

To our knowledge, few solutions exist that support *any* of these features and none that provide *all* in a single, intuitive, architecture. In this paper, we present the FFPF architecture and implementation[1]. FFPF is implemented in the Linux kernel on top of `netfilter` [**?**] and a prototype exists on the IXP1200 network processor [**?**]. Throughout the text, 'FFPF' refers to the implementation in the Linux kernel, unless explicitly stated otherwise. The remainder of this paper is organised as follows. In Section 2, a high-level overview of the FFPF architecture is presented. In Section 3, the implementation details and the filtering language are discussed. The implementation is evaluated and compared to BPF in Section 4 . Related work is discussed throughout the text and summarised in Section 5. Conclusions are drawn in Section 6.

2 FFPF high-level overview

A high-level overview of the architecture is shown in Figure 1. It shows that most of FFPF is implemented as a single kernel module. The two applications that are currently monitoring the network are each provided with three different buffers which are known as: (1) the main packet buffer ($PktBuf$), (2) a flow-specific index buffer ($Index(f)$), and (3) a flow-specific memory array (M). The main packet buffer is shared between the applications.

The idea behind FFPF is simple. Users load (in the kernel) 'expressions' that process the packets. If a packet is classified as 'interesting', it is pushed in the shared $PktBuf$ and a pointer to the packet is placed in the application's index buffer. An application uses the index buffers to find the packets in which it is interested in the shared $PktBuf$. The third buffer is used for exchanging information between the application and the expression in kernel space or to store persistent state. For instance, the expression may use it to store flow statistics. All buffers are memory mapped, so no copying between kernel and userspace is necessary. The expressions that are loaded in the kernel are able to call extensions in the form of 'external functions' that may have been loaded either by the system administrator or the users themselves. The external functions

[1] Available from `www.liacs.nl/~herbertb/projects/ffpf/`

contain highly optimised native implementations of operations that would be too expensive to execute in a bytecode interpreter (e.g., pattern matching, MD5 message digest). While this may appear as a potential safety problem, it will be shown that both speed and safety are explicitly provided.

2.1 FFPF: instantiating flows

In FFPF, a *flow* is any stream of packets that is of interest to an application. Examples include: 'all packets containing specific IP addresses and TCP ports', 'all UDP packets', 'all UDP packets containing a worm signature plus all TCP SYN packets', etc. An application may open multiple flows at the same time. Flows are opened in 4 steps. Firstly, a flow is *created*. Creating a flow sets up a user-space data structure which is returned as a flow identifier, but does not result in any packets being captured. Secondly, the data structure is *populated* by specifying for instance a filter, or a callback function for the flow. Thirdly, the flow must be *connected*. Only at connect time the flow is instantiated in the kernel, provided it passes the *admission control* check. Fourthly, an instantiated flow by itself still does not capture any packets; it first needs to be *activated*. Conversely, an activated flow can be *paused* (and subsequently re-activated). Finally, a flow can be *closed*. When a flow is closed all corresponding state is destroyed.

2.2 FFPF: security groups

A concept not found in other packet filters is that of *security groups*. A security group is a set of applications with the same *access rights* to packets, i.e., if one application in the group is allowed to read a packet, all other applications in the group are also allowed to access it. Security groups are used to minimise packet copying. Applictions in the same group *share* a common packet buffer ($PktBuf$). $PktBuf$ contains all packets for which one or more applications in the group have expressed interest. If more than one group expresses interest in the packet, it is copied only *once* per group, unlike BPF which copies the packet to each application separately. This makes FFPF cheaper than BPF when supporting multiple applications. A special group, G_0, gets zero-copy access (provided this is supported by the hardware). G_0 is statically assigned to all applications owned by `root`. As a result, in a common environment in which only the `root` runs monitoring applications, all flows access the same zero-copy packet buffer.

2.3 FFPF: processing

In FFPF almost all processing takes place at the lowest possible level, e.g. in the kernel or network processor. Beware that in general the 'lowest possible level' only means the level that has the fewest intermediate layers beneath it in packet processing. Accordingly, if specialised hardware is able to deliver packets directly to the address space of an application (without intervention of the kernel), kernel and userspace are at the same level. This is the case, for instance, for DAG cards [?]. The principle, however, remains valid: as much as possible of the processing should happen at the lowest possible level (e.g. the DAG card). This is also exemplified by our implementation of FFPF on IXP1200 network processors, where packet processing and buffer management is handled entirely by the network card.

3 Implementation

3.1 The Buffers

Both $PktBuf$ and all index buffers are circular buffers of N fixed size slots, with N constant for all circular buffers. $PktBuf$ slots are large enough to hold the maximum packet length, while the slots in the index buffers can hold no more than two 32 bit values: an index into $PktBuf$ and the packet's classification result. Whenever an 'interesting' packet is received on flow f, it is pushed to $PktBuf$ (unless a copy already exists) and a reference to the packet as well as the result of the classification is placed in $Index(f)$. A packet is considered 'interesting', if the flow expression '$Expr(f)$' processing the packet returns a non-zero result. Each valid slot in $Index(f)$ contains a result *pair*, consisting of (1) the index of the corresponding packet in $PktBuf$, and (2) the classification result returned by $Expr(f)$.

Applications read the packets that are received for a specific flow f by indexing $PktBuf$ with the values in $Index(f)$. A configuration parameter determines whether the link layer header should also be captured (the default is to capture from IP header onwards). As applications access the index buffers, the results of classification are immediately available. Although the indices only point to the packets in which they are interested, applications are also able to see what other applications in the same security group have received (but not what is received by other groups). The buffers in security group G_0 need not be copied at all. For instance, the FFPF implementation on the IXP1200 network

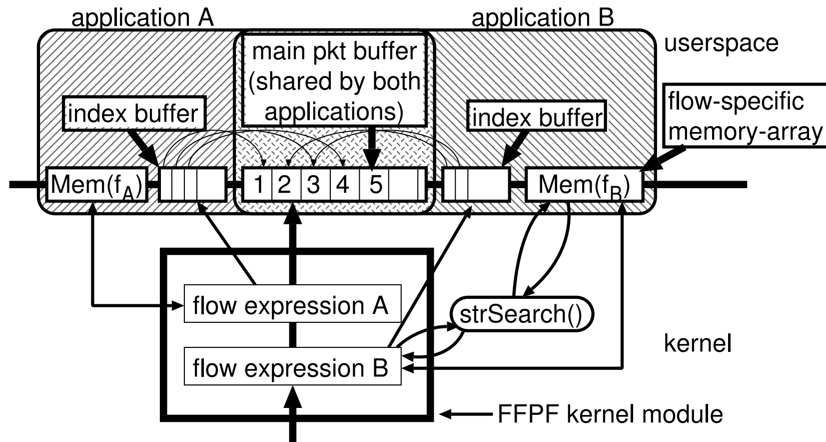

Figure 1: The FFPF architecture

processor receives all packets in G_0's $PktBuf$ *directly* and this buffer is memory mapped in its entirety to user space. Other groups do incur a single copy per packet in which they are interested (albeit not from kernel to userspace). $PktBuf$ and $Index(f)$ are `mmap`-ped 'read only'. It will be shown shortly that the flow-specific memory array is mapped read/write for the application.

3.1.1 Circular packet buffers

Circular buffers in FFPF have two indices to indicate the current read and write positions. These are known as R and W. Whenever W catches up with R the buffer is full and the packet is dropped. Both R and W are mapped read-only to an application's address space and updated in the kernel (or in the case of FFPF on IXP1200s: the network card). The packet receiving code in the kernel (or network card) writes data in a group's $PktBuf$ and updates W until the buffer is full. 'Full' means that W has caught up with the *slowest reader* in the group. Thus, the slowest reader in a group may block all other flows in that group (but not those in other groups). The R value of the slowest reader will be denoted by $\hat{R}$. The values of R are updated explicitly by the applications. One of the keys to speed is that R need not be incremented by 1 for every packet of a flow that is processed. Instead, it is possible that an application processes 1000 packets of a flow and then increments the value of R by 1000 in one go. Doing so saves 999 kernel boundary crossings. A similar mechanism is used for DAG cards as described in [**?**].

As an example, in the implementation of FFPF on IXP1200 network processors, the implementation of packet processing and buffer management is handled entirely by the IXP. The IXP receives a packet, places it in G_0's $PktBuf$, updates W, receives the next packet, and so on. Meanwhile, the flow expressions are executed in independently running processing engines on the network processor and determine whether a reference to the packet should be placed in the flows' index buffers. Applications access packets immediately, as the buffers are mapped through to userspace. While the applications are processing the packets, the kernel is not used at all. Only after an application has processed n packets of a flow and decides to advance its R explicitly, the kernel is activated. On the reception of an `advance_read_pointer()` call, the kernel will calculate the new value of $\hat{R}$ and store it on the IXP so that it can be used by the packet receiving code. In the extreme case, where only a single flow is active, the IXP code and the application work fully independently and the number of interrupts and context switches can be kept to a minimum (e.g., context switch only on the occasional `advance_read_index()` call and on timer interrupts).

3.1.2 Flow-specific memory array

The third buffer in Figure 1 is the flow's memory array M . It is used by both the flow expression ($Expr(f)$) in the kernel and the userspace application. User applications have read and write access to the memory arrays of their flows, so the arrays

can be used to exchange data between the application and a flow expression. The flow-specific memory area is *persistent*, i.e., its contents remain valid across multiple invocations of $Expr(f)$. It is argued in [?] that the absence of persistent state is one of the major drawbacks of BPF. While [?] describes how BPF can be extended to also allow for persistent memory (and explicit switching between persistent and non-persistent memory is needed), this paper describes an approach in which it is part of the design from the outset.

A simple use case is a flow f which treats the entire memory array as a hash table that is used to count the number of packets received on all TCP/IP flows. The corresponding $Expr(f)$ first checks whether a packet is TCP/IP. If so, it calculates a hash of the `<ipsrc,ipdest,srcport,dstport>` tuple and increments the counter stored at that location in the memory array. The result is that without intervention by the user application, the memory array contains the packet counts of all TCP/IP flows seen by the system (assuming the hash table is large enough). The implementation of this example consist of one line of FFPF code which is shown in Figure 3 and discussed in Section 3.3.1.

3.2 The Flows

After a flow is created, and prior to instantiation, the flow can be populated with a *flow specification*. Currently, the following features are supported:

1. *device* - limits the flow to packets received on a specific interface;
2. *callbacks* - call back the application after n packets of a flow have been received;
3. *'wait_for_n'* - if part of the flow specification, it means that each call to *activate* the flow will block until at least n packets have been received;
4. *flow expressions* - programs loaded in the kernel responsible for filtering, classifying, and other operations that need to be performed on the packets in the flow.

If *device* is not specified, the default is taken. The default is specific to an FFPF implementation and may well correspond to more than one interface. In the case of an FFPF implementation on IXP1200s, for instance, the default will be 'all ports of the network processor board', while in the case of FFPF on top of a Linux netfilter hook [?], this means 'all devices of which packets are received at the hook'.

After instantiation, each flow has read access to its R and W values and a userspace function `cbuf_readspace()` returns the number of unread packets currently in the buffers. The simplest way to print the packets in the circular buffers for a flow f is therefore:

```
for (;;) {
  n = cbuf_read_space (f);
  for (i=0; i<n i++)
    print_pkt (f->buffer [f->R+i]);
  advance_read_pointer (f, n);
  sleep (10);
}
```

While this code works reasonably efficient (especially compared to solutions where every packet results in a kernel-userspace interaction), callbacks and *'wait_for_n'* give applications more control over when to process the packets in a flow.

3.3 The Language

Flow expressions are among the most complex features of FFPF. The language in FFPF is more expressive than, for instance, BPF. In addition, it is explicitly extensible, so that functionality that is hard to implement in FPL-1 (the FFPF programming language) can still be accessed through a 'service interface', not unlike that of SNAP [?].

3.3.1 FPL-1

FPL-1 is a low-level stack language with support for most simple types and all common binary and logical operators. In addition, FPL-1 provides restricted looping and explicit memory operations to access the flow's persistent memory array. Flow expressions in FPL-1 are compiled to byte code and inserted in the FFPF kernel module by means of a special purpose code loader (discussed later).

Each time a packet arrives on one of the monitored interfaces, the FPL-1 expression of a flow f is called to see whether or not f is interested in this packet, i.e. if $Expr(f)$ returns a result not equal to zero (see also Figure 1). If so, the packet is copied to $PktBuf$ and both a reference to the packet and the result of $Expr(f)$ are placed in $Index(f)$. FPL-1 is an extremely simple language with few keywords. The entire language is summarised in Figure 2. Comma's and brackets have no use other than grouping and making the code more readable. It is realised that the terse assembly-like language is a little intimidating to users accustomed to tools like snort [?] and pcap [?]. For this reason, we have implemented most of the monitoring API (MAPI) proposed by the SCAMPI project [?]

on top of FFPF. The MAPI is a standardised monitoring API that is more expressive than existing approaches. While simple enough to support existing approaches (e.g. there exists an implementation of pcap on top it), MAPI is expressive enough to permit advanced queries like 'scan all packets for the occurrence of a string and return only the packets that match'. In the future, we intend to develop a compiler that takes rules from pcap and snort and translates them to FPL-1.

Although it is beyond the scope of this paper to discuss FPL-1 in detail, in the remainder of this section the most interesting features of the language are briefly explained. FPL-1 employs low-level, assembly-like instructions with post-fix notation to access the packets. A trivial example of an FPL-1 expression is the following program which returns 1 if the IP packet that it processes is a UDP packet (i.e., byte 9 equals 17) *and* it is sent to port 54321 (i.e., short 11 equals 54321):

```
(c9,N17,=)(s11,N54321,=)&&
```

The constructs `c9` and `s11` are interpreted as *offsets* from the start of the packet. The logical AND operation (`&&`) can be used as a conditional statement and is evaluated lazily. In other words, if `(c9,N17,=)` is false, it will not evaluate the second part. The value returned by the expression is either the value on top of the stack when the '`RET`' is called, or the value that is left on the stack when the evaluation completes.

Subexpressions and offsets Often in packet monitoring, it is necessary to access data at an offset that depends, for instance, on the length of the header (or the length of the packet). For this purpose, FPL-1 uses the notion of a *subexpression*, indicated by square brackets. As a simple example, given that `tot_len`, the total packet length in IP is determined by the 3rd and 4th bytes of the IP header (i.e., the short integer at offset 1), the following expression tests whether the byte at offset '`tot_len` − 8' is equal to 0:

```
(c[(s1,N8,-)],N0,=)
```

The result of the subexpression (the expression between the square brackets), is used as if this value appeared at this place in $Expr(f)$. For instance, '`c[N[N0]]`' is equivalent to '`c[N0]`' which in turn is equivalent to `c0`, i.e., the first byte of the packet. It is possible to return from a subexpression explicitly using the '`RET`' statement. Also, subexpressions may be nested and may contain arbitrary expressions themselves. They provide a flexible way to access fields from the packet at variable offsets.

Restricted 'For' loops For resource safety, the `For` loop construct is limited to loops with a predetermined number of iterations. Users specify both start and end values of the iteration variable, as well as the amount by which the loop variable should be incremented or decremented after each iteration. The `ForBreak` instruction, allows one to exit the loop 'early'. In this case (and also when the loop finishes), execution continues at the instruction following the `ForEnd` construct. `For` loops can be used to test a small range of bytes in the packet or even to scan the entire packet payload for the occurrence of a pattern. However, as string searching is expensive, this operation is better implemented using external functions (as shown in Figure 1 and as discussed in Section 4).

Persistent state Stacks are useful for calculations, but not for persistent state. For example, FPL-1 expressions start with a fresh stack each time a packet is received, so that state across invocations of the flow expression (e.g. packet counters) cannot be kept on the stack. Instead, FPL-1 has explicit commands to access the flow's persistent memory array. The operations `LD` and `ST` specify a memory address in the array to read from or write to. All accesses to M are checked for bounds violations.

As an example of how to use the memory, and also as an example of the compactness of FPL-1 code, consider Figure 3. The code implements the flow f that was mentioned in Section 3.1.2 an that keeps track of how many packets were received on each TCP connection. For this, the code uses a hash table that is stored in $M[1] ... M[n+1]$ (where n is the size of the hash table. $M[0]$ is used to store a temporary variable. All operations in Figure 3 (i.e., '`=,Hash,+,ST`'and '`&&`') pop two values off the stack and push the result back on. For `ST` the first pop provides the location in M where the value should be stored and the second pop provides the value to be stored (this is also pushed back on the stack again). A `LD` works in an analogous manner except that just a single value is popped off the stack (the position in M from which to read) and the value that is found at the corresponding memory location is pushed on the stack. The hash function '`Hash`' used in the expression is the default FPL-1 hash function that calculates a hash over a byte sequence of arbitrary length. The return value is less than the size of the flow's memory array divided by two, so that it can easily be used as an index in the array.

The code works as follows. If the packet is a TCP segment (byte 9 of the packet

types	simple operations	other operations
'b': bit	'!,=': (not) equal	'[expr]': sub expression
'c': int8	'+,-,*,/,%,&,|': arithmetic	'For..ForBreak..End': looping
's': int16	'<,<=,>,>=': other relational	'Extern'foo'': external function
'i': int32	'&&,||': logical and/or	'LD,ST': load from, store to memory
'N': int32 constant	'Hash': simple hash function	RET: return top of stack

Figure 2: FPL-1 language overview

```
# if this is tcp, hash the tcp connection fields and count the pkts
((c9,N6,=)((((((N14,N12,Hash),N1,+),N0,ST),LD),N1,+),(N0,LD),ST),&&),0,&&
```

Figure 3: Realistic example of FPL-1 code: count TCP flow activity

equals 6), a hash is calculated over bytes 14-23 of the packet (in other words, over the <ipsrc,ipdest,srcport,dstport> tuple). Next, this value is incremented by 1 (to make sure it is always $\geq$ 1) and the result is stored in M [0] using the ST instruction (which also pushes the value on the stack). Let's call this value h. The script then loads the value of M [h], increments it by 1 and pushes it on the stack. Let's call this value C (for count). Next, h (saved earlier in M [0]) is loaded again and C is stored at M [h]. The counting is now done. At that point C is also the top of the stack. This value is logically 'AND'ed with the result of (c9,N6,=) and the outcome of that is logically 'AND'ed with '0', to ensure the value that is returned by this function is *always* zero, so that *no packets will be stored in* $Index(f)$. Assuming the hash table is big enough, $Expr(f)$ tracks the activity in all TCP flows, without requiring a single computation in userspace.

External functions An essential feature of FFPF is its extensibility and the concept of an 'external function' is another key to speed. It is possible for both users and administrators to register FFPF kernel functions (fully optimised native code) that can be called from within the filter expression. The only difference between code registered by users and by administrators is in what this code is allowed to do. This will be discussed in more detail in Section 3.5. In FPL-1, an external function is called using the Extern construct. For instance, Extern'foo' will call external function foo. When foo returns, its return value is placed on the stack. In Figure 1, an external function called 'strsearch' is called by the flow expression of flow B. External functions allow users to call efficient C implementations of computationally expensive functions, such as checksum calculation, or pattern matching.

An external function takes as parameters pointers to the packet and to the flow's persistent memory array. If the function was registered by the system administrator it is also provided with a pointer to the flow expression itself, so that even FPL-1 code modification can be easily implemented (the usefulness of this is discussed below). External functions can process the packet just like normal flow expressions and are able to place results either in the flows' memory arrays as well as in their return values.

External functions are identified by users by their names. Internally, however, the string is mapped on a unique integer identifier. The mapping from string to integer is done (1) when the function is registered, and (2) when the user code is compiled. The advantage is that calling the external function, which happens on the fast path, does not involve a string search for the appropriate function name. Instead, indexing a table of function pointers is all that is required.

A small library of external functions has been implemented. One of the more useful functions in the library is for instance an implementation of the Aho-Corasick algorithm for efficient pattern searching with a large number of patterns [**?**]. The implementation will be evaluated in Section 4. Another interesting example of an external function is one that calls a BPF filter. In other words, FFPF programs can be used as a *meta-expression*. Provided the corresponding external function is implemented, FFPF is able to provide to its users practically all filtering techniques available in the literature (including BPF+ [**?**], DPF [**?**], PathFinder [**?**], etc.) and may even 'script together' filters from different approaches (much like a shell script on UNIX).

Self-modification External functions that are loaded by the system administrator have one additional feature: they allow for automatic modification of the flow expression from which the function was called. This is useful for applications that employ dynamic port allocation. Such applications use con-

trol channels with well-known port numbers, while data transfer takes place over ports that are negotiated dynamically and hence not known *a priori*. Examples can be found in peer-to-peer networks and multimedia streams that employ control protocols like RTSP, SIP and H.323 [?, ?] to negotiate port numbers for data transfer protocols such as RTP [?].

These data flows are complex to monitor and the problem was considered important enough to develop a special-purpose tool known as `mmdump` (based on `tcpdump`) to handle it [?]. Like xPF [?], `mmdump` adds statefulness to the pcap/BPF architecture and in addition allows filters to be self-modifying. A filter may capture and inspect all control packets and if they contain the port number to be used for data, modify itself to also capture these packets.

This behaviour is supported in FFPF by allowing an external function to modify the FPL-1 expression from which it was called. For instance, given that RTSP packets are sent on port 554, the filter in Figure (4.a) captures all RTSP/UDP packets and calls an external function called `handle_RTSP` to process them further. The return value of the function `handle_RTSP` is 0, so that RTSP packets are not delivered to the user. When called, the function scans all RTSP session packets for the occurrence of '`Transport`', '`client_port`' and '`server_port`' to find the transport protocol and the port numbers that will be used for data transfer (e.g., audio and video). It will add this information to the original filter by appending an expression for each of the port numbers. For instance, if one of these port numbers is 8000, the expression in Figure (4.b) will do the same as the expression in Figure (4.a), while also delivering the corresponding RTP packets to the user.

3.3.2 FPL-2

FPL-1 is a straightforward interpreted stack language. Although the bytecode is fairly efficient, running it in an interpreter hurts performance. Moreover, as observed by McCanne and Van Jacobson: for modern processor architectures, stack-based languages are less efficient than register-based approaches [?]. We have not investigated to what extent this is still true more than a decade later. Indeed, many stack languages are still developed today and some even enjoy huge popularity (e.g., Java, PostScript, and .Net's Common Language Runtime which runs for instance programs written in C#). Register-based approaches also have issues with portability and code size. Even so, it is certainly *easier* to map register-based languages on a matching load-store architecture efficiently. For this reason, the language issue has recently been reconsidered and we are now adding support for a new language that (1) compiles to fully optimised object code, and (2) is based on registers and memory, and (3) contains all of BPF's instructions in addition to a whole set of new instructions. A configuration switch determines whether to use 'old-style' FPL-1, or 'new-style' FPL-2.

Although in the current implementation it is actually possible to load, run and remove FPL-2 filters (as native code), the compiler itself is still under development. For this reason only the principles are discussed in this paper. FPL-1 and FPL-2 compilers are able to generate 'resource safe' code, i.e., it is possible to check at compile time how many resources can be consumed by an expression, which external functions are called, how many loop iterations may be incurred, etc. Neither language supports pointers and interaction with the rest of the kernel is limited to the explicitly registered external functions. As a result, a simple admission check rejects flow expressions that do not agree with the local safety policy and no runtime checks for resource consumption are necessary. At runtime FFPF only checks for array bound violations, divide by zero, etc.

In an approach modelled after the OKE (see Section 3.5), the FPL-2 compiler takes the flow expression, checks whether it is safe and if so, compiles it to a linux kernel module which is subsequently compiled by `gcc`. It also generates a *compilation record*, which proves that this module was generated by the local (trusted) FPL-2 compiler. The proof contains the MD5 of the object code and is signed by the compiler (Figure 5).

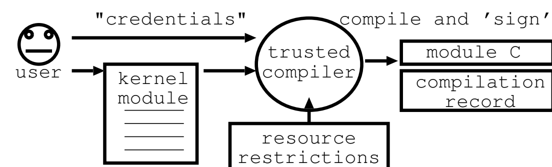

Figure 5: User compiles kernel module

To load an FPL-2 flow expression, users provide the code loader with the object code, as well as the code's compilation record. The code loader checks whether the code is indeed FPL-2 code generated by the local compiler and if this is the case, loads it in the kernel. The user has now loaded a fully optimised register-based expression in the kernel (see Figure 6).

(a) `(c9,N17,=)(s11,N554,=),Extern'handle_RTSP',&&,&&`
(b) `(c9,N17,=)(s11,N554,=),Extern'handle_RTSP'&&,(s11,N8000,=)||,&&`

Figure 4: RTSP example: (a) captures RTSP packets, (b) same, but also sends RTP packets to user

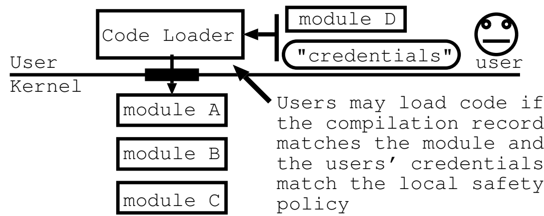

Figure 6: User loads module in the kernel

3.4 Admission Control

In reality, the code loader's admission check is more complex, as it is also used to check additional credentials provided by the user. These credentials determine, for instance, whether or not the user is allowed to load an expression in this security group (or at all). Both for FPL-1 and FPL-2, admission control is implemented as a stand-alone daemon that is called at flow connect time. Recall that a flow is connected (or instantiated) *after* it has been created and populated, where populating a flow means adding, for instance, a flow expression, requests for specific callbacks, etc.

At instantiation time, the complete flow specification is combined with authentication/authorisation information and sent as an *instantiation request* to the FFPF kernel module. The kernel module pushes the instantiation request on a queue that is read by the admission control daemon. The daemon reads the request, compares the flow specification both with the user's credentials and with the host's local security policy and returns a verdict ('accept' or 'reject'). Credentials and trust management in FFPF are implemented in KeyNote [**?**]

In addition to ensuring that users do not attempt to create flows in security groups to which they have no access, the admission control daemon also provides much finer-grained control. For instance, it possible to specify for a specific user that a flow f will be accepted only if in $Expr(f)$ the first occurrence of external function `foo` is preceded by the first occurrence of function `bar`, or that the number of calls to `bar` is always less than the number of calls to `foo`, etc.

Observe that flow creation and instantiation are deliberately decoupled. The alternative is to create a flow which immediately captures packets and to add filters and functions to it incrementally. Not only may the latter approach result in capturing and processing 'unwanted' packets for a brief period of time, it also prevents fine-grained admission control. Consider for example, the following safety policies (1) a call to an external function `strsearch` is permitted for packets arriving on NIC-1 (but not for other NICs) *only* if it is preceded by a sampling function, (2) all calls to a function`produce_end_result` *must* be followed by a return statement, (3) if no call is made to an external sampling function, the callback that is requested should wait for at least 1000 packets (e.g., to limit the number of callbacks). These policies can only be checked if the entire flow specification is available. The examples show that admission control guard against 'unsafe' flows, but can also be used to guard against 'stupid errors'. A slightly modified version of the FFPF admission control daemon is also used in the SCAMPI network monitoring project [**?**].

3.5 Third-party external functions

Normally, external functions that are coded in C and compiled as kernel modules can only be loaded by the system administrator. However, in previous work we have shown how third-party users can load fully optimised native code in the Linux kernel, without compromising safety in any way using the Open Kernel Environment (OKE[2]). OKE support was added to FFPF, so that even non-root users are allowed to load fast native functions in the kernel and register them with FFPF. Subsequently, these functions can be called by flow expressions just like ordinary external functions. The details of the OKE safety mechanisms are published elsewhere (e.g. [**?**, **?**]), so in this section only the results are summarised.

The FPL-2 way of injecting code was directly

[2]Available from `www.liacs.nl/~herbertb/projects/oke/`

modelled after the OKE, so here the same compile and load steps are followed as sketched in Figures 5 and 6, except that qthe language used in the OKE is *Cyclone* , a 'crash-free' version of C [**?**]. Unlike FPL-2, this is a language that supports pointer memory allocation and full interaction with the kernel. Accordingly, to be able to generate 'resource safe' code, the compiler must check and instrument the user code much more strictly.

Depending on the credentials provided by the user the OKE compiler will restrict the user code in terms of access to resource, e.g. in terms of CPU, heap, and stack usage, access to APIs, access to sensitive fields in packets at zero runtime overhead (e.g., access IP addresses), accesses to kernel heap, etc. Many of the safety mechanisms are enforced by 'environment-setup-code' that is automatically prepended to the code. For instance, in the case of FFPF, the environment setup code determines that the user registers an external function that matches the prescribed format, and explicitly provides the API to the rest of the kernel.

The amount of restrictions may vary from user to user. Highly-privileged users may have unrestricted heap space, while others may get only a small amount (and yet others may get no heap whatsoever). The result is that the OKE compiler is able to generate fully optimised, native code that is guaranteed to be crash free and resource safe with respect to a local safety policy. These policies, the trust management and the credentials that used throughout FFPF are implemented in KeyNote [**?**]. Given the appropriate credentials, user are permitted to load code of a specific type, i.e., with a specific set of resource restrictions. The credentials are checked by admission control.

FFPF users are allowed to load external OKE functions in the kernel and register them for use in FPL-1 nd FPL-2, provided the resource restrictions that were applied to the user's code agree with the local safety policy and the user is in possession of the appropriate credentials. Once loaded, the code runs natively at full speed, although any resource constraint that cannot be checked statically incurs a runtime check. The advantage of external OKE functions is that users do not depend on the root user to load the desired functionality as an external function. The disadvantage is the overhead incurred by the OKE runtime checks. In practice, the cost of full resource control in the OKE is less than 10%.

4 Experimental analysis

FFPF is evaluated by comparing its performance to BPF. While the main benefit of FFPF may well be the extended functionality, FFPF is shown to outperform pcap/BPF on all accounts, despite the slow, stack-based implementation of FPL-1. First, experiments are evaluated that do not exploit any of the advanced features of FFPF. Next, experiments with multiple applications and external functions are discussed. All measurements were taken on a 1.8 GHz P4 running a 2.4.18 kernel connected via a 64/66 PCI bus to a gigabit ethernet interface. The system is capable of handling a maximum rate of roughly 400 Mbps (using UDP packets of 1470 bytes) without significant packet loss.

FFPF was implemented in the Linux kernel on top of netfilter. There is also a prototype implementation on IXP1200 network processors (currently without external functions). Only the latter provides true zero-copy functionality. To be fair to BPF and evaluate equivalent systems, only the former implementation is used in the comparison. It captures packets on the netfilter `PRE_ROUTING` hook, i.e. just before the protocol stack determines where this packet should go. Unfortunately, this implementation is far from optimal, as a packet incurs a significant amount of processing in the Linux kernel even before it arrives at the netfilter hook. For instance, the packet is queued in a backlog buffer, a soft IRQ is scheduled and handled, and if needed, defragmentation is performed. Only after all this has completed the packet arrives at the hook. As a result, FFPF incurs significant overhead that has little to do with FFPF itself. BPF does not suffer from this to the same extent, as it receives the packets much earlier in the processing path. To decrease the overhead, we are currently porting a driver that captures packets in the FFPF circular buffer *directly*[3] (without any kernel processing). It is expected that this will improve performance dramatically.

Also, the current implementation of FPL-1 is not very fast. For instance, a filter that (a) tests whether a packet is UDP/IP with a specific port number and (b) copies the packet to a group's $PktBuf$ costs 2700 cycles in FPL-1. The same filter in FPL-2 (with manual translation from FPL-2 to C) takes 480 cycles. In this section, however, we only evluate FFPF with FPL-1. The BPF version used is the vanilla implementation in the Linux 2.4.18 kernel.

Figure 7 shows the per-packet overhead for BPF and FFPF when executing a simple filter at different rates. The filter tests whether the packet is UDP/IP

[3] We are indebted to Luca Deri for sending us the code.

sent to destination port 4333. The pcap implementation is a `pcap_loop` that blocks until it has received 100 packets. The equivalent for FFPF is a 'wait_for(100)' operation that does the same. What is shown as per-packet overhead in all figures in this section, is the time it takes to receive 100 packets in the application, divided by 100. This is not the overhead due only to the filters or the copies; it also includes sleeping while waiting for packets, etc. While micro-measurements, such as the cycle counts mentioned above, more accurately reflect what overhead is caused by what component of the architecture, we take the view that user experience is what matters. It means that much of the time in the measurements is spent 'waiting' for packets. The figures show that the difference between FFPF and pcap is minimal and that FFPF is marginally cheaper, despite its inferior interpreter. The lines drop for increasing rate because the time that is spent 'waiting for packets' which dominates the overhead at low speeds decreases at high speeds.

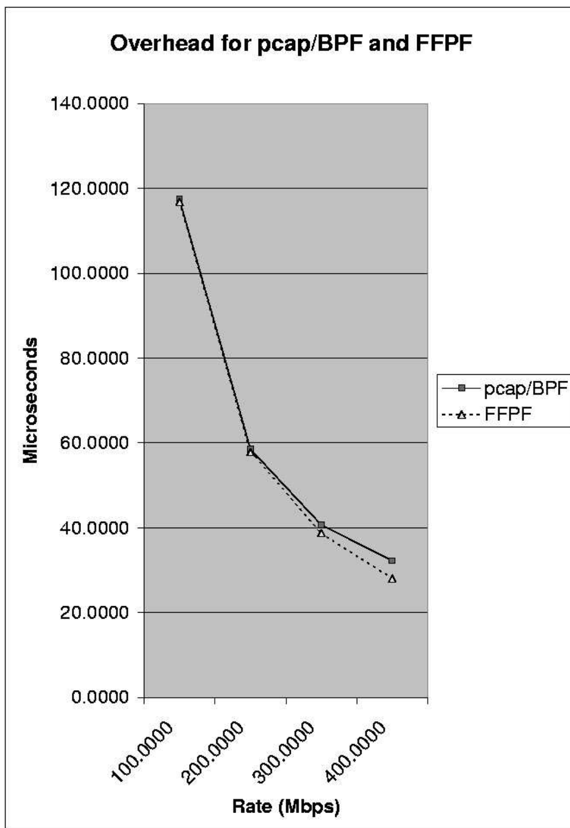

Figure 7: BPF and FFPF overhead for simple packet filtering

In Figure 8, we also plot the drop rates for pcap and FFPF at different speeds. FFPF starts to lose packets at 500 Mbps, when it drops 0.0003% (or 51 packets). However, this is without taking into consideration the performance of netfilter. For rates below 500 Mbps, netfilter does not drop packets, but at 500 Mbps, close to 20% of the packets is lost by netfilter. Even so, at that rate, pcap loses 68% of the packets.

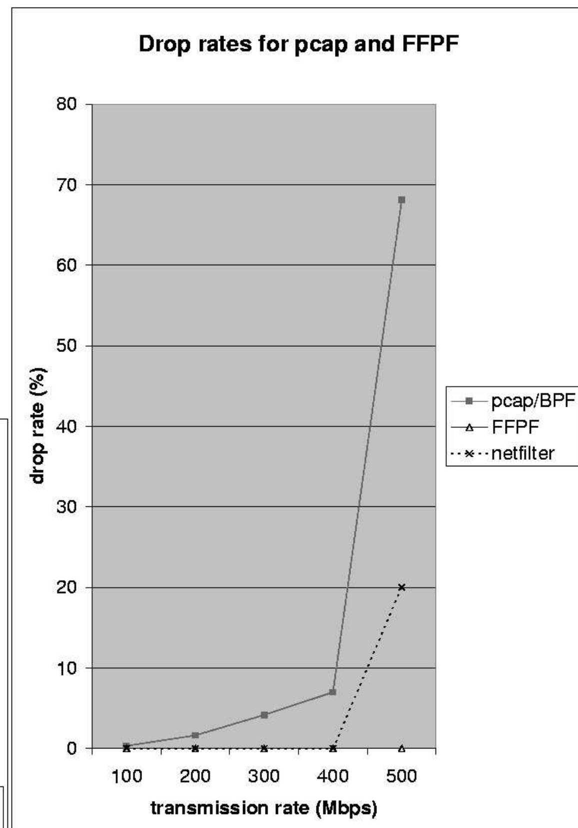

Figure 8: BPF and FFPF drop rates at various speeds

Next, we modify our experiment to TCP across a single link and measure the throughput that can be achieved on the connection while monitoring is active. Besides changing from UDP to TCP, the number of packets for which the applications wait is increased to 1000. For comparison, an implementation that blocks waiting for a single packet is also shown. The amount of time spent sleeping in this case is less, but as expected the maximum throughput drops. There is still only a single connection and a single application. The results are shown in Table 1. FFPF outperforms pcap both in overhead per packet and packet loess (pcap takes 1.5 times as long and loses roughly 10% of the bandwidth).

After these basic measurements, in the remainder

Experiment	per-pkt overhead	max rate (Mbps)
pcap_loop_1000	43.3 sec	382
ffpf_wait_1000	28.5 sec	408
ffpf_wait_1	31.8 sec	357

Table 1: Maximum TCP throughput while monitoring

Experiment	per-pkt overhead
pcap_loop_100 4 applications	78 sec
ffpf_wait_100 4 applications	29 sec

Table 2: Overhead with 4 applications

of this section, more complex experiments are considered. Table 2 shows what happens to the overhead per packet at 400 Mbps if the number of applications interested in the packet is greater than one. The table shows the result for 4 identical applications. For pcap, the overhead per packet increases significantly (presumably due to additional copying), while the overhead for FFPF actually decreases somewhat (as the number of copies per packet remains 1). The results confirm that FFPF is less expensive when multiple overlapping flows are active simultaneously.

In the final experiment, the usefulness of external functions is demonstrated. The application uses an advanced string matching algorithm known as Aho-Corasick to filter packets based on keywords [**?**]. One of the advantages of the Aho-Corasick algorithm is that the number of strings to look for hardly influences the time it takes to process a single packet. It is therefore ideally suited for scanning network traffic in domains like intrusion detection. Indeed, the latest version of the intrusion detection system known as Snort also employs the algorithm [**?**].

The application in this experiment is interested in capturing only those packets that contain the strings `www`, `WWW`, `http`, or `HTTP`. The experiments were conducted twice. The first time ('min work'), the matching packets all have the string in the first few bytes of the 1470 byte long payload. The second time ('max work'), the string sits in the last few bytes. This represents maximum and minimum overhead created by Aho-Corasick. In the FFPF implementation, the algorithm is executed in the kernel as an external function and only the matching packets are transferred to userspace. As BPF is not able to perform Aho-Corasick in the kernel, the pcap version applies the code to all packets in userspace. The percentage of packets that match is set to 10%, 50% and 90%, respectively. Again, we measure in userspace how long it takes to receive 1000 packets and divide this number by 1000. The algorithm alone takes 412 clock cycles for 'min work' and 12016 for 'max work', which translates to an overhead that ranges between 0.2 μs and 6.7 μs. The results are shown in Figure 9 for a rate of 400 Mbps. Especially when the percentage is low (as would normally be the case), pcap/BPF wastes a lot of effort copying packets to user space that are subsequently discarded. FFPF performs better (approximately twice as well), as it only needs to push up the *right* packets. In addition, Figure 10 shows that the number of dropped packets is much worse for pcap than for FFPF. Actually, again no packets are dropped by FFPF itself, and all packet loss can be attributed to netfilter. As the percentage that matches decreases below 10%, the dropped packets ratio for pcap is greater than 50%.

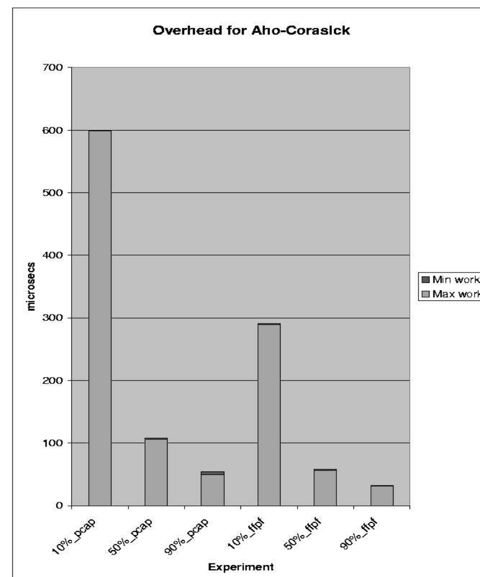

Figure 9: Overhead for Aho-Corasick algorithm

External functions provide an advantage over pcap mainly in cases where (1) the processing cannot realistically be done in BPF, (2) the copying overhead compared to the processing overhead is significant, (3) many packets are not interesting to the user application anyway. However, even for a fairly expensive operation like string search, FFPF performs significantly better than pcap.

5 Related work

Many attempts were made to extend and improve BPF. For example, both xPF and mmdump add per-

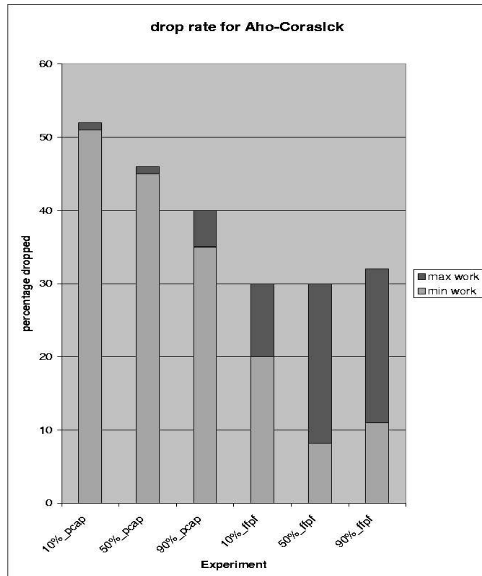

Figure 10: Drop rate for Aho-Corasick algorithm

sistent state to BPF. The tool mmdump was tailor-made for multimedia applications with dynamically allocated ports and for this purpose allows for filter modifications[**?**, **?**]. This feature has been generalised in FFPF by the flow-specific memory arrays.

MPF enhances the BPF virtual machine with new instructions for demultiplexing to multiple applications and merges filters that have the same prefix [**?**]. This approach is generalised by PathFinder which represents different filters as predicates of which common prefixes are removed [**?**]. PathFinder is interesting in that it is amenable to implementation in hardware. DPF extends the PathFinder model by introducing dynamic code generation [**?**]. BPF+ [**?**] shows how an intermediate static single assignment representation of BPF can be optimised, and how just-in-time-compilation can be used to produce efficient native filtering code. All of these approaches target filter optimisation especially in the presence of many filters, and as a result are not supported directly in FFPF (although it is simple to add any of these approaches as an external function). With FPL-2, FFPF relies on `gcc`'s optimisation techniques to combine filters from multiple flows and on external functions for expensive operations.

Operating systems like Exokernel, and the original Nemesis [**?**, **?**] allow users to add code to the oprating system and implement single address spaces to minimise copying. While FFPF no doubt can be efficiently implemented on either of these systems, one of the most important features is that it minimises copying on a very popular OS that does not have a single address space.

Most closely related to FFPF is the SCAMPI architecture [**?**]. SCAMPI borrows heavily from the way packets are handled by DAG cards [**?**]. It is assumes the hardware is able to write the packets immediately in the applications' address spaces and implements access to the packet buffers through a userspace daemon. Traditional network cards are supported but handled much like pcap handles packet processing: packets first have to be pushed to userspace. Moreover, SCAMPI does not support user-provided external functions and relies on traditional filtering languages (BPF). Unlike FFPF, finally, SCAMPI allows only a non-branching (linear) list of functions to be applied to a stream.

6 Conclusions

In this paper, the architecture and first implementation of the fairly fast packet filter is discussed. FFPF provides a complete monitoring platform that caters to a variety of applications. It was shown to be both more flexible and more efficient than an approach based on the BSD packet filter. If necessary, existing filtering approaches can be added as extensions. Speed is gained by minimising packet copying and by allowing users to execute computationally expensive functions as native code in the kernel. While it is clear that some components of FFPF need to be optimised (e.g. the expression language and the combination of multiple filters), it is our belief that the FFPF architecture q represents a significant step forward from common approaches like BPF.

Acknowledgements

This work was supported by the EU SCAMPI project IST-2001-32404.We would like to thank Mihai Cristea and Trung Nguyen, both from the Universiteit van Leiden, for their work on the FPL-2 compiler and the implementation of FFPF on IXP1200s, respectively. Many thanks to Kostas Anagnostakis of the University of Pennsylvania for commenting on earlier versions of this paper.

References