A compiler for Packet Filters

Mihai-Lucian Cristea and Herbert Bos

Universiteit Leiden
Niels Bohrweg
2333 CA Leiden
{cristea,herbertb}@liacs.nl

Keywords: embedded systems design, packet filtering, network processors

Abstract

FFPF, the Fairly Fast Packet Filter allows users to stick a small amount of code into the kernel. The code currently uses a low-level stack language for describing the flow expressions. In modern processor architectures, the processing speed of a stack-based language is slower than a register-based language. Therefore, our architecture proposes to boost up the filter processing by using the well-known gcc compiler. In this paper, we describe the compiler architecture as well as a specific implementation using FFPF.

1 Introduction

In today's high-speed networks, monitoring tools are required to handle very large amounts of network traffic per time unit. Moreover, the users' requirements become more and more complex (e.g. scanning every packet for the occurrence of a worm [10]). For these reasons, more sophisticated tools are needed than the ones that are available today. Furthermore, these tools should be geared for high traffic rates.

A monitoring tool consists mainly of three parts: at the highest level is the interface to the user and at the lowest level is the kernel device driver. The third part, no less important than the other two, is the 'packet filter'. In this paper, a packet filter refers to a small amount of code that processes packets at an early stage, e.g. to classify it as interesting, or to gather statistical information about traffic. In essence, it is an expression of the user monitoring requirements at the lowest levels, and it should satisfy the following two conditions. First, it should be easy for users to specify them and the packet filter language should provide flexibility. Second, the filter should be processed as fast as possible. Therefore, the packet filters should be suitable for running at the lowest possible level (e.g. the OS kernel or even specialised hardware such as Network Processors (NP) [6]. Unfortunately, current monitoring tools do not accomplish these goals well.

The fairly fast packet filter (FFPF) [3] is an approach to network packet processing that adds many new features to existing filtering solutions like BPF [8]. FFPF is designed for high speed by pushing computationally intensive tasks to the kernel (or even network processors, if present) and by minimising packet copying. FPL-1, the current filter language, is a straightforward interpreted stack language. Although the byte code is fairly efficient, running it in an interpreter hurts performance. 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. The new language, FPL-2, and its compiler are the topic of this paper.

The FFPF architecture proposes to achieve the goal of fast packet processing by using a custom front-end for FPL-2 filter expressions, and as back-end the well-known `gcc` compiler. As a result, the object code will be heavily optimised even though we did not write an optimiser ourselves. This object code will be invoked by FFPF for each incoming packet. During packet handling, the filter object code updates the processing results in a local memory area shared with FFPF and from here it is `mmapp`-ed to the applications. Moreover, by using a safe language for the packet filters, the resulting system provides safety, while packets are processed at the speed of native code, fully optimised for the latest hardware. A trusted FPL-2 compiler and a custom code loader guarantee that only programs written in FPL-2 can be loaded in the FFPF framework.

The remainder of this paper is organised as follows. In Section 2, an overview of the FFPF monitoring tool is presented. The direct benefits of using a filter compiler in FFPF are shortly described in Section 3. In Section 4, the compiler design and architecture are discussed. The implementation details are presented in Section 5. The evaluation of the proposed architecture is highlighted on a concrete example in Section 6. Related work is highlighted in Section 7 and conclusions drawn in Section 7.

2 The FFPF tool

FFPF provides a complete solution for filtering and classification at high speeds, either in the kernel or on a network processor, while minimising copying. FFPF was implemented in the Linux kernel on top of netfilter [11]. There is also a prototype implementation on IXP1200 network processors (currently it does not include all features). While both versions significantly reduce copying, only the latter provides true zero-copy functionality.

A high-level overview of the architecture is shown in Figure 1. It shows that the central component of FFPF is the Buffer Management System (BMS). It is beyond the scope of this paper to describe the functionality of BMS in detail. BMS consists of a main buffer shared by all applications that might access it. BMS also includes several secondary buffers and pointer lists needed to assure a good system management. The main purpose of BMS is to capture all packets that are considered ‘interesting’ and hand a reference to these packets to the appropriate applications. The idea behind FFPF is simple. Users load ‘expressions’ that process the packets. If a packet is classified as ‘interesting’, it is pushed in the shared buffer and a pointer to the packet is placed in the application’s index buffer. An application uses the index buffer to find the packets in which it is interested. In addition, an expression has its own chunk of memory (known as `’MEM’`) that is shared with the application and which it may use to store results to be read by the application or temporary values that do not disappear between invocations (persistent state). All buffers are memory mapped, so no copying between kernel and user-space is necessary.

FFPF allows one to insert fairly complex expressions in the kernel of an OS. In fact, the expressions are more like simple programs and may contain loops, hashing, functions, etc. The filter evaluation is ‘fairly’ efficient, but not as fast as it could be, as it uses a stack-based interpreter. In the ’93 BPF paper, Steven McCanne and Van Jacobson have argued that stacks are slower than register/memory/accumulator interpreters.

3 Improving the packet filter processing

Flow expressions are among the most complex features of FFPF. The language used in FFPF so far for describing a flow expression, is called FPL-1 (the FFPF programming language 1).

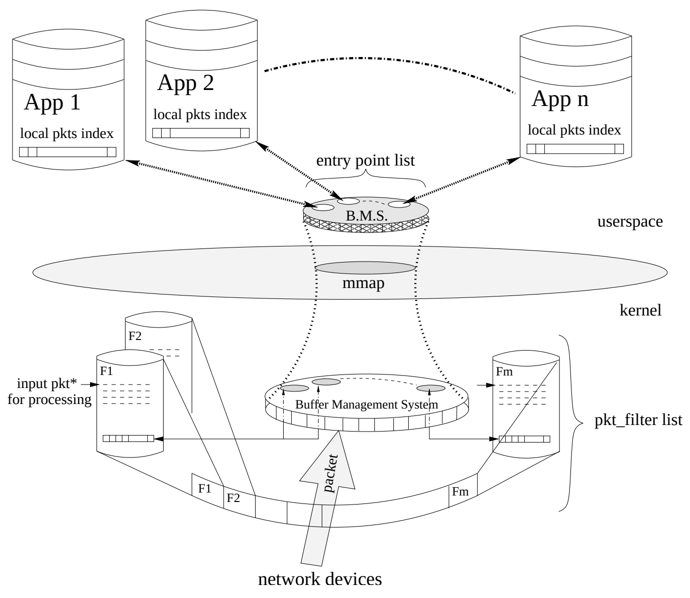


Figure 1: FFPF overview

3.1 From interpreted to compiled code

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 a 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. 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. By calling an FPL-1 expression, a long process is involved as the filter’s byte-code runs in an interpreter. As it is difficult to compete both in writing efficient code optimisers and in providing an efficient packet processing environment, we have chosen to exploit the existing optimiser of `gcc` for the former task. In essence, the FPL-2 compiler generates C code that is subsequently completed by `gcc` to a Linux kernel module that interfaces with FFPF. In this paper, we describe the support that was added for FPL-2, a new language that ① compiles to fully optimised object code, and ② is based on registers and memory. A configuration switch determines whether to use ‘old-style’ FPL-1, or ‘new-style’ FPL-2.

3.2 The FFPF-compiler tool

Assuming that the user has a filter expression written in FPL-2, there are three steps to follow as illustrated in Figure 2. The first step consists of translating the filter into an FFPF-filter module. This module is a direct translation of FPL-2 code into C code and forms the core of the future kernel module that FFPF needs to invoke on every packet. As a second step, the

FFPF-compiler passes the C code files to `gcc`. The result is the object code of an FFPF kernel module. The last step is loading the filter into FFPF by calling the FFPF helper function 'Insert filter'.

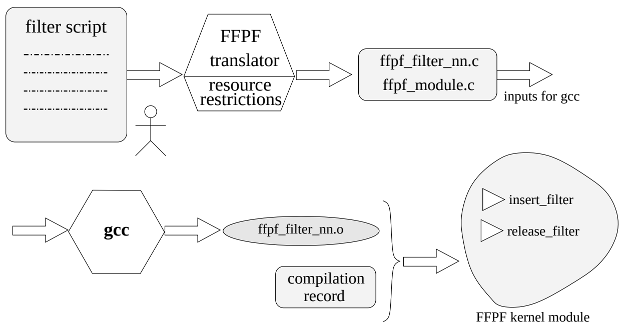


Figure 2: User compiles FPL-2 filter expression.

The FPL-2 compiler, in addition to the 'C translation', also makes some specific checks for security reasons. Because our framework allows users to run code in very low hardware levels, we must provide proofs of authorisation of the compiled code before injecting it into kernel or hardware. The FPL-2 compiler accomplishes this by using KeyNote to generate a *compilation record*, which proves that this module was generated by this (trusted) FPL-2 compiler [1]. The proof contains the MD5 of the object code and is signed by the compiler. When loading an FPL-2 filter expression, users provide 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's FFPF target. In fact, these are additional security checks, but these are beyond the scope of this paper. Interested readers are referred to [2]. The user has now loaded a fully optimised register-based expression in the kernel (see Figure 2).

4 The FPL-2 language

In this section, the compiler architecture and its language elements are described. The FPL-2 compiler uses the traditional compiler construction tools such as lex and yacc for lexical analysis and parsing phases of compilation. It also uses `TreeCC` [13] as abstract syntax tree (AST) tool to facilitate working with bison/yacc.

In the followings subsections, the FPL-2 elements are described.

4.1 Operators, expressions and statements in FPL-2

Operators act upon operands and specify what is to be done to them. Expressions combine variables and constants to create new values. Statements are expressions, assignments, external function calls, or control flow statements which make up filter programs. As illustrated in Figure 3, most of the operators/expressions are designed in a way that resembles well-known imperative languages such C or Pascal. Therefore, the users should find it fairly easy to use this filter language.

operator-type	**operator**
Arithmetic	`+, -, /, *, %, --, ++`
Assignment	`=,*=, /=, %=, +=, -=` `<<=, >>=, &=, ^=, |=`
Logical/Relational	`==, !=, >, <, >=, <=,` `&&, ||, !`
Bitwise	`&, |, ^, <<, >>`

statement-type	**operator**
if/then	IF (expr) THEN statement FI
if/then/else	IF (expr) THEN stmt1 ELSE stmt2 FI
for()	FOR (initialise; test; update) stmts; BREAK; stmts; ROF
external function	EXTERN(fnct_name, sharedMem_IndexRead, sharedMem_IndexWrite)
hash()	HASH(start_byte, size)

Figure 3: Operators, expressions and statements

4.2 Restricted FOR loops

For resource safety, the `FOR` loop construct is limited to loops with a pre-determined 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 `BREAK` instruction, allows one to exit the loop 'early'. In this case (and also when the loop finishes), execution continues at the instruction following the `ROF` 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.

4.3 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 (see also [3]). While we do not intend to discuss the way users may load native code in the kernel directly, we emphasise that it is done in a safe manner. The method that was used for this purpose is known as the Open Kernel Environment, which is described in more detail in [4]. In FPL-2, an external function is called using the `EXTERN` construct. For instance, `EXTERN(foo)` will call external function `foo`. 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 a pointer to the packet, and also pointers to memory blocks to be used by the function. Every external function that is registered also declares information about the memory needed by the function. This takes the form of two parameters: ① start index of 'read-only' shared memory with external functions, ② start index of 'write' shared memory (e.g. some functions may return results and this is how such elasticity is reflected). An external function can process the packet just like normal flow expressions and is able to place results either in its shared memory `MEM` or in its return values. We will discuss the memory allocation in a little more detail in Section 4.4.

4.4 Data addressing modes

The addressing modes of packet data are important for ease of use. Therefore, we support several ways of addressing in order to provide an intuitive way of handling the data.

Packet data addressing Index addressing mode combined with *variable offset* index addressing mode can give any bit of data within the packet data, as shown in Figure 4. For improving the readability of programs, we used the lexical conventions according to the industrial standard IEC 1131-3 [7].

types	operator	point to
Byte	PKT.B[num]	the whole byte 'num'
	PKT.B[num].U4[0-1]	the lowest/highest 4 bits of byte 'num' cast to byte
	PKT.B[num].LO or PKT.B[num].HI	the lowest/highest 4 bits of byte 'num' cast to byte
	PKT.B[num].U1[0-7]	the bit of byte 'num' cast to byte
Word	PKT.W[num]	the word 'num'
	PKT.W[num].U8[0-1] PKT.W[num].HI or PKT.W[num].LO	the lowest/highest part of word 'num' cast to byte
	PKT.W[num].U4[0-3]	a byte of word 'num'
	PKT.W[num].U1[0-15]	a bit of word 'num'
DWord	PKT.DW[num]	the double-word 'num'
	PKT.DW[num].U16[0-1] PKT.DW[num].HI or PKT.DW[num].LO	the lowest/highest part of dword 'num' cast to word
	PKT.DW[num].U8[0-3]	a byte of dword 'num'
	PKT.DW[num].U4[0-15]	a half-byte of dword 'num' cast to byte
	PKT.DW[num].U1[0-31]	a bit of dword 'num' cast to byte

Figure 4: Packet addressing modes

Some examples of packet addressing are drawn in Figure 5. These examples show how easy and intuitive a specific IP field is reached within the packet. The most used fields may also be defined as *macros*, allowing users to customise the way they express themselves. Moreover, using a register or memory variable as index for packet reference the language increases considerably the applications area. In the (trivial) example shown below, the sum of the first 20 bytes in the packet is computed.

```
FOR (R[0]=0;R[0]<20;R[0]++)
  M[0]+= PKT.B[R[0]];
ROF
```

IP field	operator	result
VERS	PKT.B[0].HI	U8 (the value of high four bits of the first byte)
HDR-LEN	PKT.B[0].LO	U8 (the value of low four bits of the first byte)
Precedence+TOS	PKT.B[1]	U8 (the value of whole byte)
TOTAL-IP-LEN	PKT.W[1]	U16 (the value of second word)
DATAGRAM-ID	PKT.W[2]	U16 (the value of third word)
FRAGM-AREA	PKT.W[3]	U16 (the value of fourth word)
TIME-TO-LIVE	PKT.B[8] or PKT.W[4].HI	U8
PROT-CARRIER	PKT.B[9] or PKT.W[4].LO	U8
HEADER-CHKS	PKT.W[5]	U16
SRC-ADDR	PKT.DW[3]	U32
DEST-ADDR	PKT.DW[4]	U32
UDP-SRC-PORT	PKT.DW[5].HI or PKT.W[10]	U16
UDP-DEST-PORT	PKT.DW[5].LO or PKT.W[11]	U16

Figure 5: IP packet fields

Memory data addressing Another important addressing mode is used for accessing the memory locations. We support two types of memory: a shared memory array `MEM` and fast local registers. The shared memory `MEM` is provided by the FFPF core. Therefore, the filter module does not perform dynamic memory allocation/deallocation. The shared memory is used for data exchange between software modules. In the FFPF architecture, there are many interfaces related to shared memory, but only two of them are relevant for the packet filtering language and thus they are described presently. The first interface involves the data exchange between the FFPF core and each loaded filter module. As an example, the results of a particular filter module can be periodically read by a user application. The second interface is used for data exchange between 'external' functions. Assume that there are two functions `'Foo'` and `'Bar'` already registered within the FFPF core, and one needs to access the processing results of the other.

Generally, using data stored in registers increases the processing speed in case of very often used variables. The maximum number of local registers is defined in the 'resource restrictions' and depends on the hardware (as indicated by Figure 2).

As shown below, a memory location is easily accessed (retrieved from / stored in) by using the assignment operator.

```
M[12]=123+45*6/7-M[8]; // memory access
R[0]=M[8]%R[1];        // register access
R[1]=123+R[1];         // compute the relative index
M[R[1]] += IP_LEN      // store total packet size
                       // PKT.W[1] (from vocabulary)
```

Using the language elements already described up to here, an example of filter code is drawn below.

```
IF ((R[0]<100) && (PKT.W[1]>150))
  // or IP_LEN > 150 (big packets)
THEN
  M[0]+= IP_LEN; // total no. of bytes
  R[0]++;        // count no. of big packets
FI
```

This trivial example collects the data that is needed to calculate the average size of the first 100 packets larger than 150 bytes. In fact, it counts the total amount of bytes and it keeps track of the number of incoming packets. The purpose of the example is to show how data fields are referenced within packets by using operator `'PKT.'` and also to show how variables are handled, by using memory `'M[]'` and registers `'R[]'`.

5 Implementation

Our architecture implements a source-to-source compiler, a kernel module project and the interface to the existing FFPF monitoring tool.

5.1 FFPF translator

The FFPF translator takes a filter expression, written in FPL-2 from a file (e.g. '`ffpf_filter_01.kef`'). The translates gives the translated C code as result. This file (e.g. '`ffpf_filter_01.c`') is part of the filter kernel module. Next, this kernel module is compiled by `gcc` with full optimisation on and will be inserted by the authorised user. As a result, FFPF will register and start handling it.

5.2 Interfacing FFPF with compiled filter object

There are two interfaces. The first interface describes inserting/registering and unregistering/removing kernel modules. The second interface takes care of run-time handling of the filter within FFPF.

Assuming that a filter has been compiled in the filter kernel module, it must be inserted and registered by FFPF. The user can simply inject the filter module into the kernel by using the FFPF loader (e.g. '`./ffpf_loader ffpf_filter_nn.o`'). While doing so, FFPF checks also the module authorisation and decides whether the module is going to be accepted or not. See Callout 1 in Figure 6 and `init_module()` from the code-lines described in Figure 7.

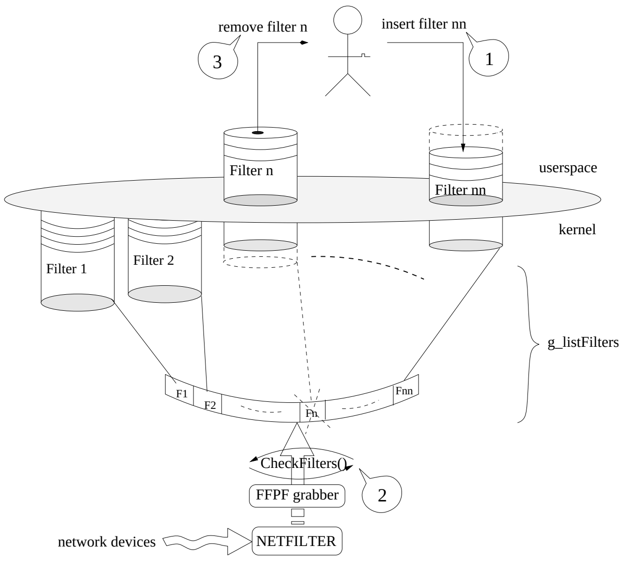


Figure 6: FFPF filters interfaces

The FFPF "packet grabber" employs for instance a hook registered within Linux `Netfilter`. The hook behaves almost as an interrupt handler, and is invoked at each incoming packet. The hook function gets as parameter a pointer to current packet. The packet pointer is passed to a function `CheckPkt` from each filter module registered in the filters list `'g_listFilters'`. The filter list is maintained by the FFPF core in such way that any filter registering/unregistering operation goes safely through only one point and thus mutual exclusion is ensured. See Callout 2 in Figure 6 and code-lines in Figure 8.

The user might release a packet filter from the filters list by invoking the FFPF loader with the `-u` option on the specific filter (e.g. '`./ffpf_loader -u ffpf_filter_n.o`') as illustrated in Callout 3 in Figure 6. When removing the module, FFPF automatically checks whether the specified filter is registered or not and if so, releases the pointer from '`g_listFilters`'.

```
int init_module(void)
{// registering the filter module using FFPF's helper
  struct module* pOwner = THIS_MODULE;
  printk (PFX "++ entering %s\n", pOwner->name);
  FFPF_Filter_Register(pOwner->name, &CheckPkt);
  return 0;
}
int CheckPkt(char* pPkt, int iSize, void* pMemory)
{
  MEM* pSharedMem = (MEM*)pMemory;
  /* TRANSLATED CODE WILL BE ADDED HERE @*/
  return 0;
}
```

Figure 7: FFPF filter module template

```
int CheckFilters(struct iphdr *pkt)
{
  struct list_head* pPos;
  int iFiltersFound = 0;
  FILTER* pFilter=NULL;
  FOR_EACH(pPos, &g_listFilters)
  {
    pFilter=list_entry(pPos, FILTER, g_listFilters);
    pFilter->pCheckFunction((char*)pkt,
          sizeof(struct iphdr), &pFilter->Memory);
    iFiltersFound++;
  }
  return iFiltersFound;
}
```

Figure 8: FFPF hands packets to registered filter modules

6 A filter example

The proposed architecture is evaluated by ① taking a practical filter expression, written in FPL-2, ② compile it using the FFPF-compiler, ③ inject it into the Linux kernel with the help of FFPF, and ④ measuring its overhead during filter processing. The results are compared to an equivalent implementation in FPL-1. As the compiler is still under construction, only a few simple examples are evaluated. The first filter example consists of counting all TCP flows, where a 'flow' is defined as all incoming packets from a certain source and are going to a certain destination, it is shown below.

```
IF (PKT.B[9] == 6)  // is this TCP?
THEN
  R[0]=HASH(12, 8); // calc. hash over IP addresses
  M[R[0]]++;        // increment the counter
FI
```

This filter expression checks whether the IP packet is TCP (tenth's byte of the packet is equal to 6) and if so it makes a hash of the packet fields 'SRC_ADR' and 'DEST_ADR' together (8 bytes of data, starting at byte number 12). The hashing result, that is unique with high probability, forms an index (flow identifier) for storing the counter value.

6.1 Compiling a filter

The FPL-2 compiler consists of a script that first generates all the source files needed for the kernel module, and then calls gcc to compile them into a loadable module. Next, it uses KeyNote to generate a compilation record for the resulting object code.

6.2 Processing a filter

Assuming that the filter example has been successfully injected into the FFPF core, each incoming packet will be processed as follows (see Figure 6). Once the packet is received by the Linux `Netfilter` framework, its pointer is passed to the FFPF "packet grabber" and thereafter to the `CheckFilters()` function. As observed in Figure 8, the `CheckFilters()` function runs one by one every `pCheckFunction` found registered in the filters list '`g_listFilters`'. By calling `pCheckFunction`, the compiled filter expression is executed. This `pCheckFunction` has the following parameters: the pointer of the current packet as input for data processing, and the address of the local shared memory (provided by FFPF core) as input/output for processing results.

In this way, the goal - processing data 'as fast as possible' is achieved. No packets and no memory blocks are copied. The application user gets access directly to the processing results by the 'memory mapping' mechanism already implemented in the FFPF core.

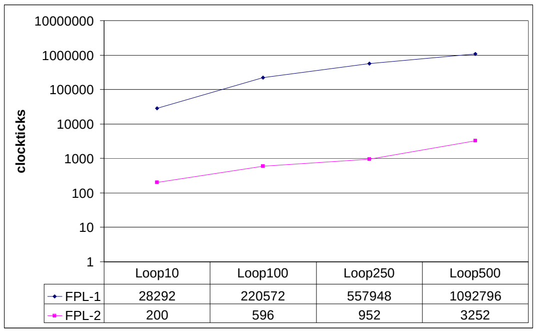
6.3 Results

The benchmark consists of running the same filter expression for a certain number (e.g. 15) of succesive times, measuring the overhead, in clock-ticks, introduced by the filter check function for each time. The result is the median value of these 15 measurements and it is shown in Figure 9 among other filter expression processing results.

```
M[2]=10;  //100; 250; 500 - maximum iterations number
M[0]=0;
FOR (M[1]=0; M[1]<M[2];M[1]++)
IF (PKT.B[M[1]] == 0x65)  // is this character 'A'?
THEN
  M[0]++;                 // increment the counter
FI
ROF
```

This filter, in all its three versions (maximum iterations number differs), perform an extensive computation - searching of a specific character (e.g. 'A') in the packet data. If such a character is found, it is counted.

It is clear that FPL-2 easily outperforms FPL-1 (note that the scales are logarithmic). This is no surprise, as FPL-1 uses a handwritten interpreter, while FPL-2 is fully optimised C code. Especially for more complex processing, such as looking at all bytes in the payload, this difference in performance is very big.



	Loop10	Loop100	Loop250	Loop500
FPL-1	28292	220572	557948	1092796
FPL-2	200	596	952	3252

Figure 9: FPL-1 versus FPL-2.

	Assign	**Hash**
FPL-1	1020	1684
FPL-2	172	392

Figure 10: FPL-1 versus FPL-2.

In Figure 10 we also show a comparison of FPL-1 and FPL-2 of the performance of applying a single assignment, and of a hash function. Again, it is clear that FPL-2 is much more efficient. As a result, we believe that FPL-2 is an important contribution to the FFPF framework.

7 Related work

FFPF is strongly related to previous approaches to packet filtering such as MPF and BPF [9, 8]. The FPL-1 language is similar to MPF in that it uses an interpreted stack language, while FPL-2 is similar to BPF in that is designed for speed, and based on a registers/memory/ALU model. Unlike BPF, however, FPL-2 is not interpreted but compiled to fully optimised native code. FFPF differs from most existing approach by minimising copying.

The loading model for fully optimised native code was based on similar code loading approaches in the Open Kernel Environment [4]. The OKE in turn uses ideas originally found in SPIN [12], but employs a trusted compiler and custom code loader, as it cannot rely on safety features of its programming language. The same is true for FPL-2.

FFPF is related to the SCAMPI project [5]. While both projects share the same philosophy of processing as much of the data as possible at the lowest levels of the processing hierarchy, FFPF is significantly less complex.

Not surprisingly, most closely related to FPL-2 is FPL-1. The languages are roughly equivalent in expressive power, but FPL-2 is much more readable. The main differences as far as language design is concerned are that ① operands in FPL-2 are always explicit (instead of implicitly using the stack), ② the addressing modes are better structured, ③ the partitioning of `MEM` when calling an external function is explicitly addressed, and ④ registers were added.

8 Conclusions

In this paper we have described how FFPF, the fairly fast packet filter, was enhanced with a new language, known as FPL-2. The language has several advantages over its predecessor FPL-1. Firstly, it is compiled to fully optimised native code, rather than byte-code that is executed in an interpreter. Secondly, it is based on a modern memory/registers/ALU model, rather than on the (slower) stack-based architecture used by FPL-1. Thirdly, its similarity to traditional imperative programming languages makes FPL-2 much more readable than its predecessor. At the same time safety is guaranteed by the trusted compiler and the custom code loader which ensure that only programs known to be safe can be loaded in the kernel or network processor. The FPL-2 approach was evaluated experimentally by implementing a set of programs in both FPL-1 and FPL-2 and comparing their execution times. As expected, FPL-2 significantly outperforms FPL-1 in every respect.

Acknowledgements

This work was supported by the EU SCAMPI project (IST-2001-32404).

References

[1] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, chapter The Role of Trust Management in Distributed Systems Security, pages 185–210. Springer-Verlag Lecture Notes in Computer Science, Berlin, 1999.

[2] H. Bos and G. Portokalidis. Ffpf: Fairly fast packet filters. `http://www.liacs.nl/~herbertb/projects/ffpf/`.

[3] H. Bos and G. Portokalidis. Packet Monitoring at High Speed with FFPF. Technical Report TR-2004-01, LIACS, Leiden University, Niels Bohrweg 1, 233 CA Leiden, January 2004.

[4] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *Proceedings of IEEE OPENARCH'02*, New York, USA, June 2002.

[5] J. Coppens, S. V. den Berghe, H. Bos, E. Markatos, F. D. Turck, A. Oslebo, and

S. Ubik. SCAMPI: A scalable and programmable architecture for monitoring gigabit networks. In *Proceedings of E2EMON Workshop*, Belfast, UK, September 2003.

[6] T. Engbersen. Network processors. *Computer Networks*, 41(5):545–547, April 2003.

[7] P. S. in Industrial Control Programming. Iec-61131, 2003. `http://www.plcopen.org/`.

[8] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX conference*, San Diego, Ca., Jan. 1993.

[9] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th Symposium on Operating System Principles*, pages 39–51, Austin, Tx., Nov. 1987. ACM.

[10] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Adminstration Conference*, 1999. Available from `http://www.snort.org/`.

[11] P. Russel. Writing a module for netfilter. *Linux Magazine*, June 2000.

[12] S. Savage and B. Bershad. Issues in the design of an extensible operating system. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, Nov. 1994.

[13] P. L. Southern Storm Software and I. Free Software Foundation. Tree compiler-compiler, 2003. `http://www.southern-storm.com.au/treecc.html`.