eBPF on the Mainframe

Packet filtering and more

Michael Holzheu <michael.holzheu@de.ibm.com>
LinuxCon Dublin 2015/10/05

Trademarks & Disclaimer

The following are trademarks of the International Business Machines Corporation in the United States and/or other countries.

AIX*	IBM*	PowerVM	System z10	z/OS*
BladeCenter*	IBM eServer	PR/SM	WebSphere*	zSeries*
DataPower*	IBM (logo)*	Smarter Planet	z9*	z/VM*
DB2*	InfiniBand*	System x*	z10 BC	z/VSE
FICON*	Parallel Sysplex*	System z*	z10 EC	
GDPS*	POWER*	System z9*	zEnterprise	
HiperSockets	POWER7*		zEC12	

* Registered trademarks of IBM Corporation

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

The following are trademarks or registered trademarks of other companies.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.
IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.
Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
Windows Server and the Windows logo are trademarks of the Microsoft group of countries.
ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Java and all Java based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.
Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.
Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

* Other product and service names might be trademarks of IBM or other companies.

Notes:

Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.

IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.

This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Prices subject to change without notice. Contact your IBM representative or Business Partner for the most current pricing in your geography.

Notice Regarding Specialty Engines (e.g., zIIPs, zAAPs, and IFLs)

Any information contained in this document regarding Specialty Engines ("SEs") and SE eligible workloads provides only general descriptions of the types and portions of workloads that are eligible for execution on Specialty Engines (e.g., zIIPs, zAAPs, and IFLs). IBM authorizes customers to use IBM SE only to execute the processing of Eligible Workloads of specific Programs expressly authorized by IBM as specified in the “Authorized Use Table for IBM Machines” provided at

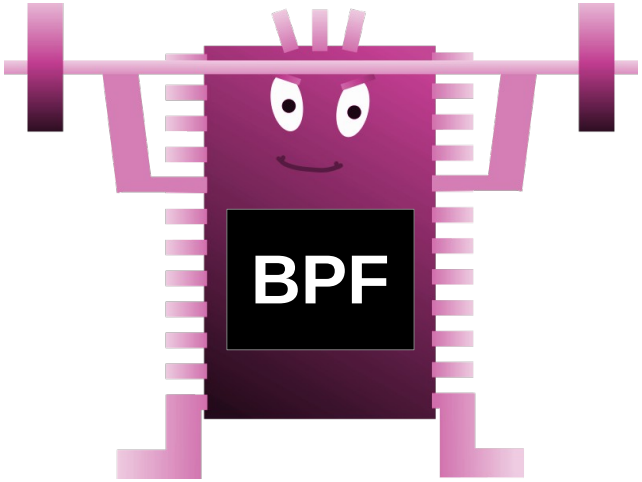
www.ibm.com/systems/support/machine_warranties/machine_code/aut.html (“AUT”).

No other workload processing is authorized for execution on an SE.

IBM offers SEs at a lower price than General Processors/Central Processors because customers are authorized to use SEs only to process certain types and/or amounts of workloads as specified by IBM in the AUT.

Agenda

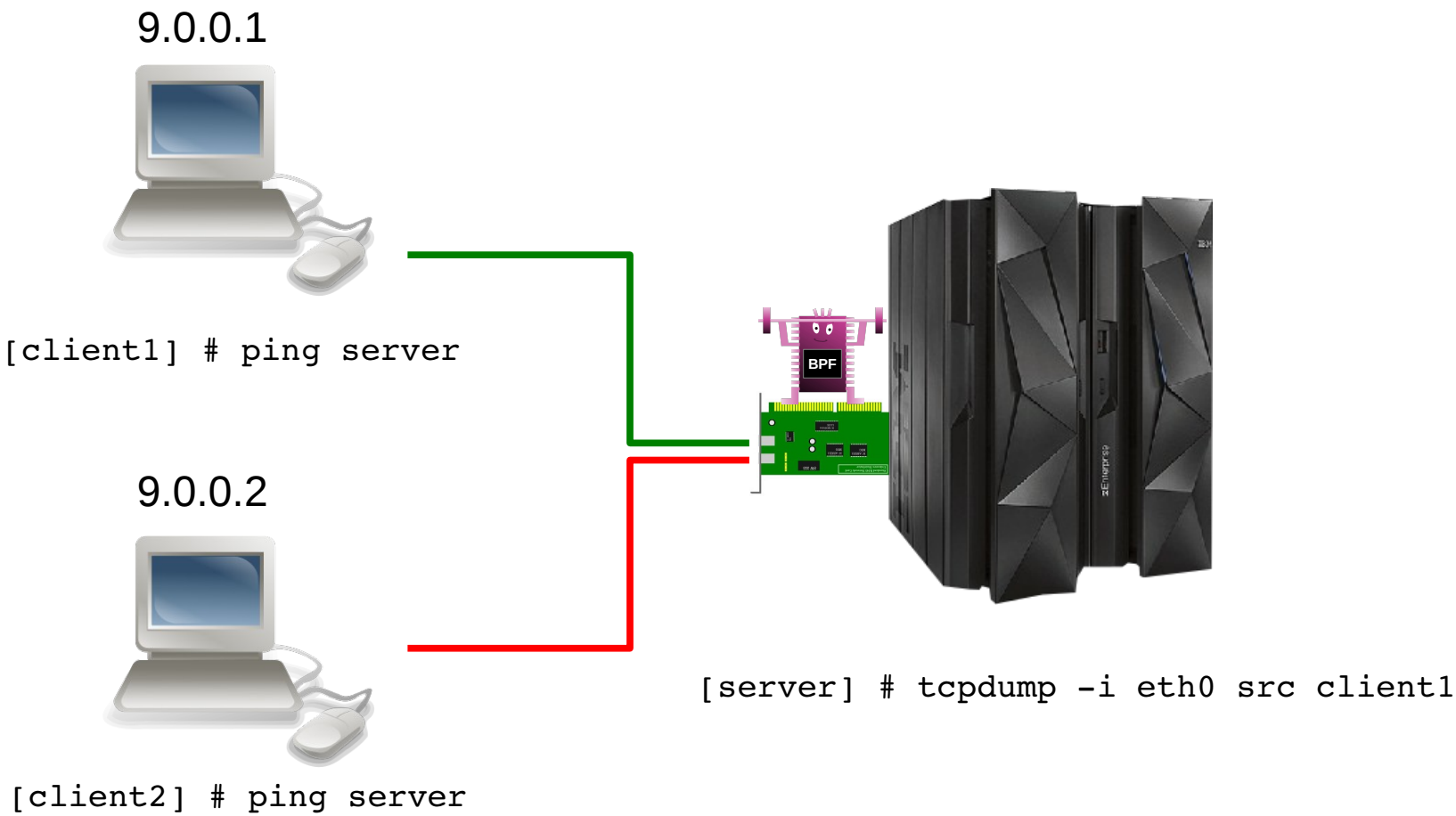
- Berkley Packet Filter (BPF)
- extended Berkley Packet Filter (eBPF)
- eBPF system call
- eBPF and LLVM


BPF

BPF - Berkley Packet Filter

- First described in paper 1992 (Steven McCanne and Van Jacobson)
- Generic assembler language and byte code (ISA)
- Used for kernel packet filtering on Free-BSD and Linux
- Interpreter and BPF kernel JIT compiler backends
- Filter program is attached to socket
- Efficient: Packets are dropped already in kernel

BPF - Scenario tcpdump

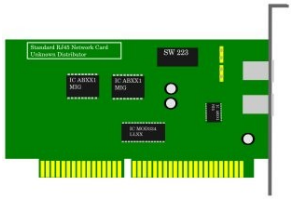


BPF - Scenario tcpdump

```
[server] # tcpdump -i eth0 src client1
```

kernel

[client1]



[client2]

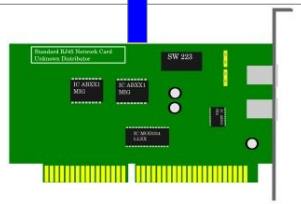
BPF - Scenario tcpdump

```
[server] # tcpdump -i eth0 src client1
```

```
sockfd = socket()
```

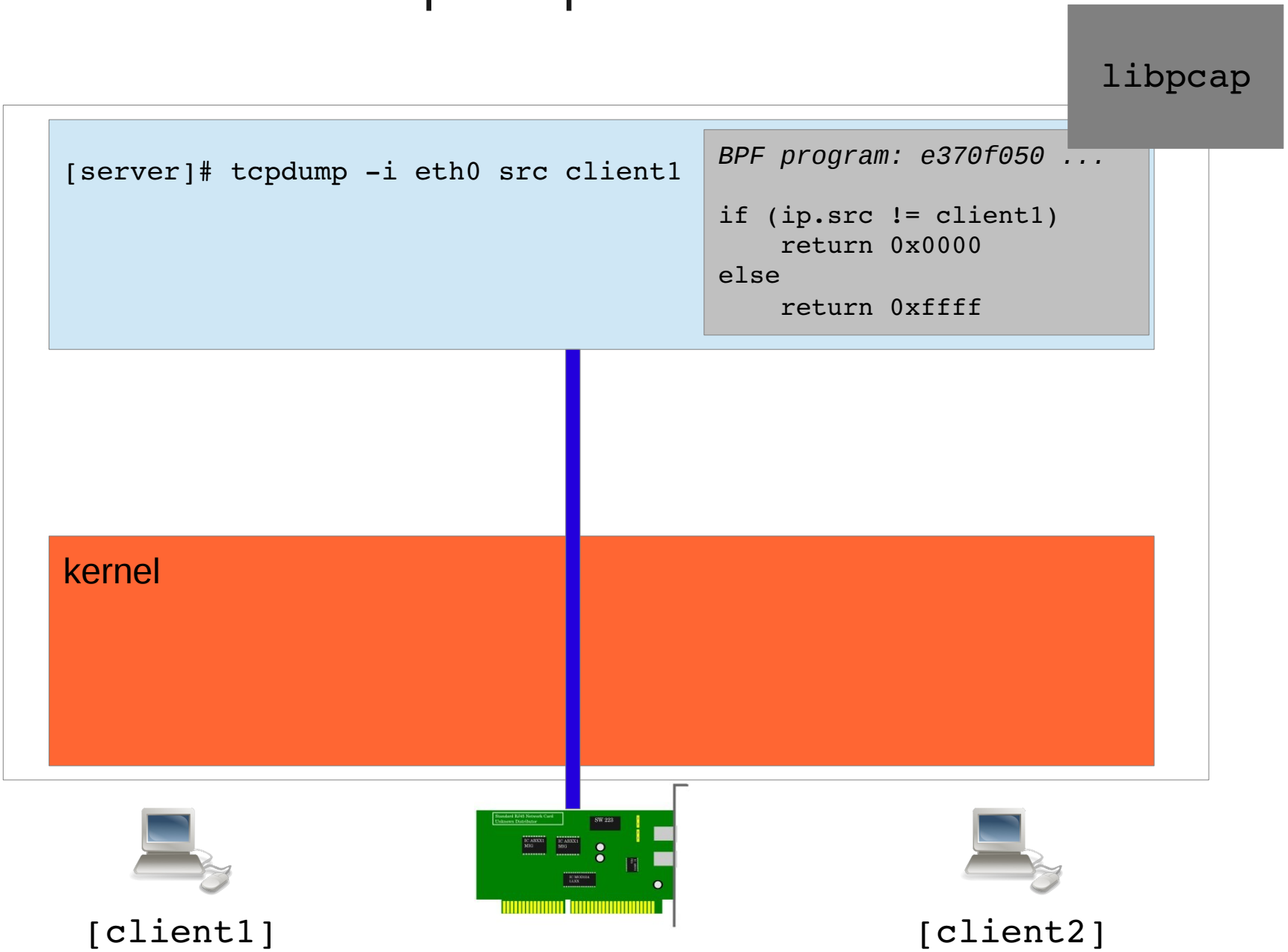
kernel

[client1]



[client2]

BPF - Scenario tcpdump



BPF - Scenario tcpdump

```
[server] # tcpdump -i eth0 src client1
```

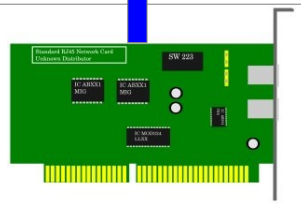
```
setsockopt(sockfd,
  SOL_SOCKET,
  SO_ATTACH_FILTER, &bpf,
  sizeof(bpf));
```

kernel

```
BPF program: e370f050 ...

if (ip.src != client1)
    return 0x0000
else
    return 0xffff
```

[client1]



[client2]

BPF - Scenario tcpdump

```
[server] # tcpdump -i eth0 src client1
```

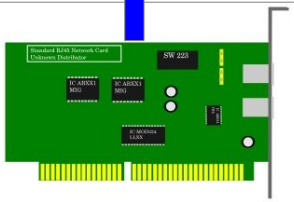
```
read(sockfd, buf, sizeof(buf));
```

kernel

```
BPF program: e370f050 ...

if (ip.src != client1)
    return 0x0000
else
    return 0xffff
```

[client1]



[client2]

BPF - Scenario tcpdump

```
[server] # tcpdump -i eth0 src client1
```

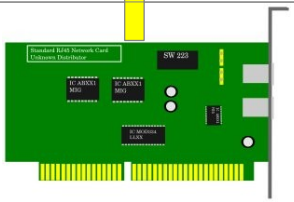
```
read(sockfd, buf, sizeof(buf));
```

kernel

SKB: src=client2

```
BPF program: e370f050 ...

if (ip.src != client1)
    return 0x0000
else
    return 0xffff
```



[client1]

[client2] # ping server

BPF - Scenario tcpdump

```
[server] # tcpdump -i eth0 src client1
```

```
read(sockfd, buf, sizeof(buf));
```

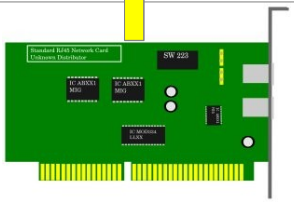
kernel

SKB: s client2

```
BPF program: e370f050 ...

if (ip.src != client1)
    return 0x0000
else
    return 0xffff
```

[client1]



[client2]

BPF - Scenario tcpdump

[server]

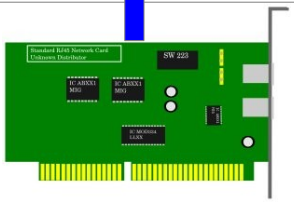
```
[server] # tcpdump -i eth0 src client1
```

```
read(sockfd, buf, sizeof(buf));
```

kernel

```
BPF program: e370f050 ...

if (ip.src != client1)
    return 0x0000
else
    return 0xffff
```



[client1]

[client2]

BPF - Scenario tcpdump

```
[server] # tcpdump -i eth0 src client1
```

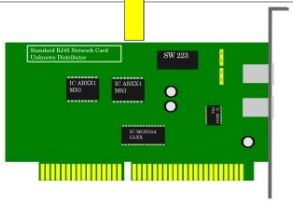
```
read(sockfd, buf, sizeof(buf));
```

kernel

SKB: src=client1

```
BPF program: e370f050 ...

if (ip.src != client1)
    return 0x0000
else
    return 0xffff
```



```
[client1] # ping server
```

```
[client2]
```

BPF - Scenario tcpdump

```
[server] # tcpdump -i eth0 src client1
```

```
read(sockfd, buf, sizeof(buf));
```

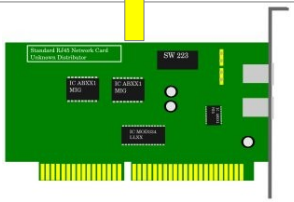
kernel

SKB: src=client1

```
BPF program: e370f050 ...

if (ip.src != client1)
    return 0x0000
else
    return 0xffff
```

[client1]



[client2]

BPF - Scenario tcpdump

```
[server] # tcpdump -i eth0 src client1
```

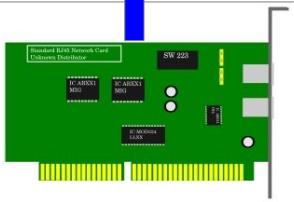
IP: src=client1

kernel

```
BPF program: e370f050 ...

if (ip.src != client1)
    return 0x0000
else
    return 0xffff
```

[client1]



[client2]

BPF - Scenario tcpdump

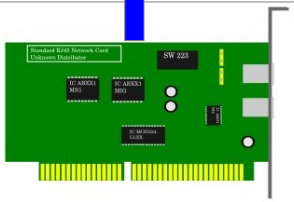
```
server: # tcpdump -i eth0 src client1
IP client1 > server: ICMP echo request,id 30146,seq 1,length 64
```

kernel

```
BPF program: e370f050 ...

if (ip.src != client1)
    return 0x0000
else
    return 0xffff
```

[client1]



[client2]

BPF - 32 bit machine language (ISA)

- Akkumulator: a
- Index register: x
- Immediate: k
- Jump targets: jt and jf
- Temporary memory: M[]

opcode:16	jt:8	jf:8
k:32		

```
/usr/include/linux/filter.h

struct sock_filter {
        __u16   code;
        __u8    jt;
        __u8    jf;
        __u32   k;
};
```

opcodes	*addr modes*				
ldb	[k]		[x+k]		
ldh	[k]		[x+k]		
ld	#k	#len	M[k]	[k]	[x+k]
ldx	#k	#len	M[k]	4*([k]	
st	M[k]				
stx	M[k]				
jmp	L				
jeq	#k, Lt, Lf				
jgt	#k, Lt, Lf				
jge	#k, Lt, Lf				
jset	#k, Lt, Lf				
add	#k		x		
sub	#k		x		
mul	#k		x		
div	#k		x		
and	#k		x		
or	#k		x		
lsh	#k		x		
rsh	#k		x		
ret	#k		a		
tax					
txa					

BPF - Example

T-MAC	S-MAC	T/L	DATA	FCS
6 Byte	6 Byte	2 Byte	46–1500 Byte	4 Byte
target MAC	source MAC	Type or Length	data	Frame Check Sequence

Ethernet Header

0	4	8	12	16	20	24	28	31 Bit
Version	IHL	TOS		Total Length				
Identification				Flags	Fragment Offset			
TTL		Protocol		Header Checksum				
Source Address								
Destination Address								
Options and Padding (optional)								

IP Header

```
# tcpdump -i eth0 src client1 -d
(000) ldh      [12]                          ## Read T/L
(001) jeq      #0x800        jt 2    jf 4 ## IP packet?
(002) ld       [26]                          ## Read src IP: offset 26
(003) jeq      #0x09000001   jt 8    jf 9 ## 9.0.0.1 ?
(004) jeq      #0x806        jt 6    jf 5 ## ARP packet?
(005) jeq      #0x8035       jt 6    jf 9 ## RARP packet?
(006) ld       [28]                          ## Read src ARP IP: offset 28
(007) jeq      #0x09000001   jt 8    jf 9 ## 9.0.0.1 ?
(008) ret      #65535                        ## client1     -> rc = 0xffff
(009) ret      #0                            ## not client1 -> rc = 0x0000
```

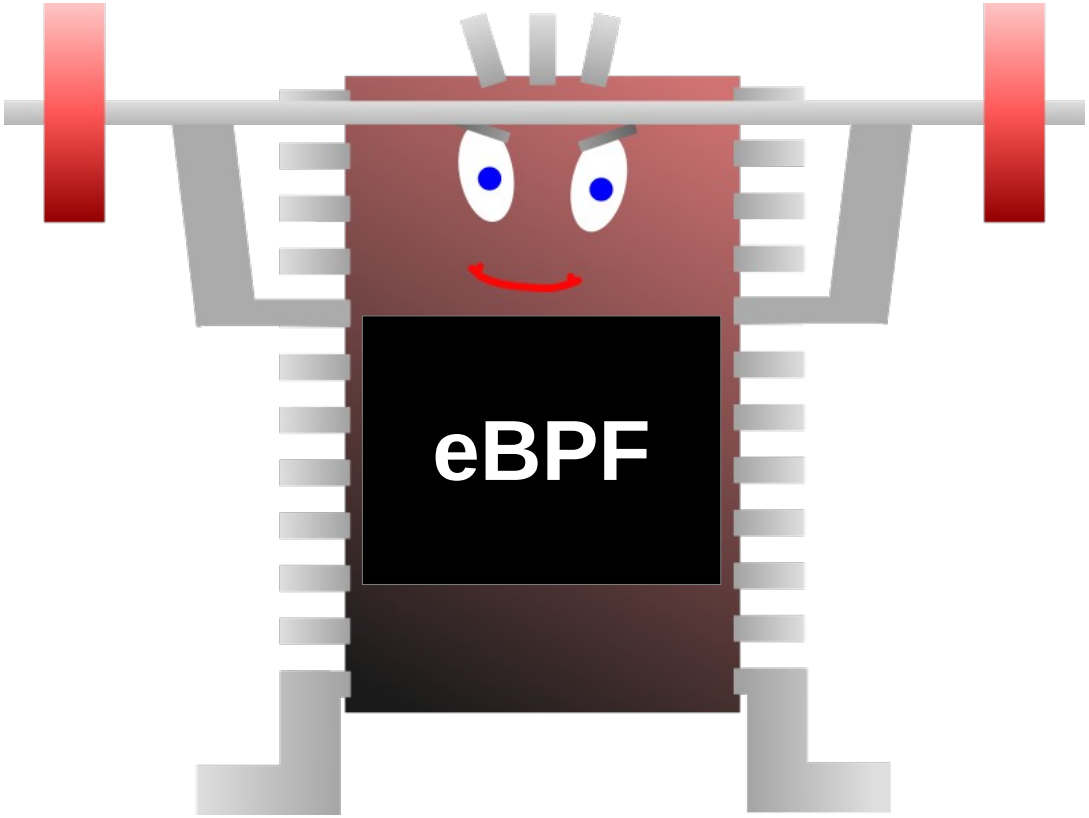
BPF - sysctl JIT setting

/proc/sys/net/core/bpf_jit_enable:

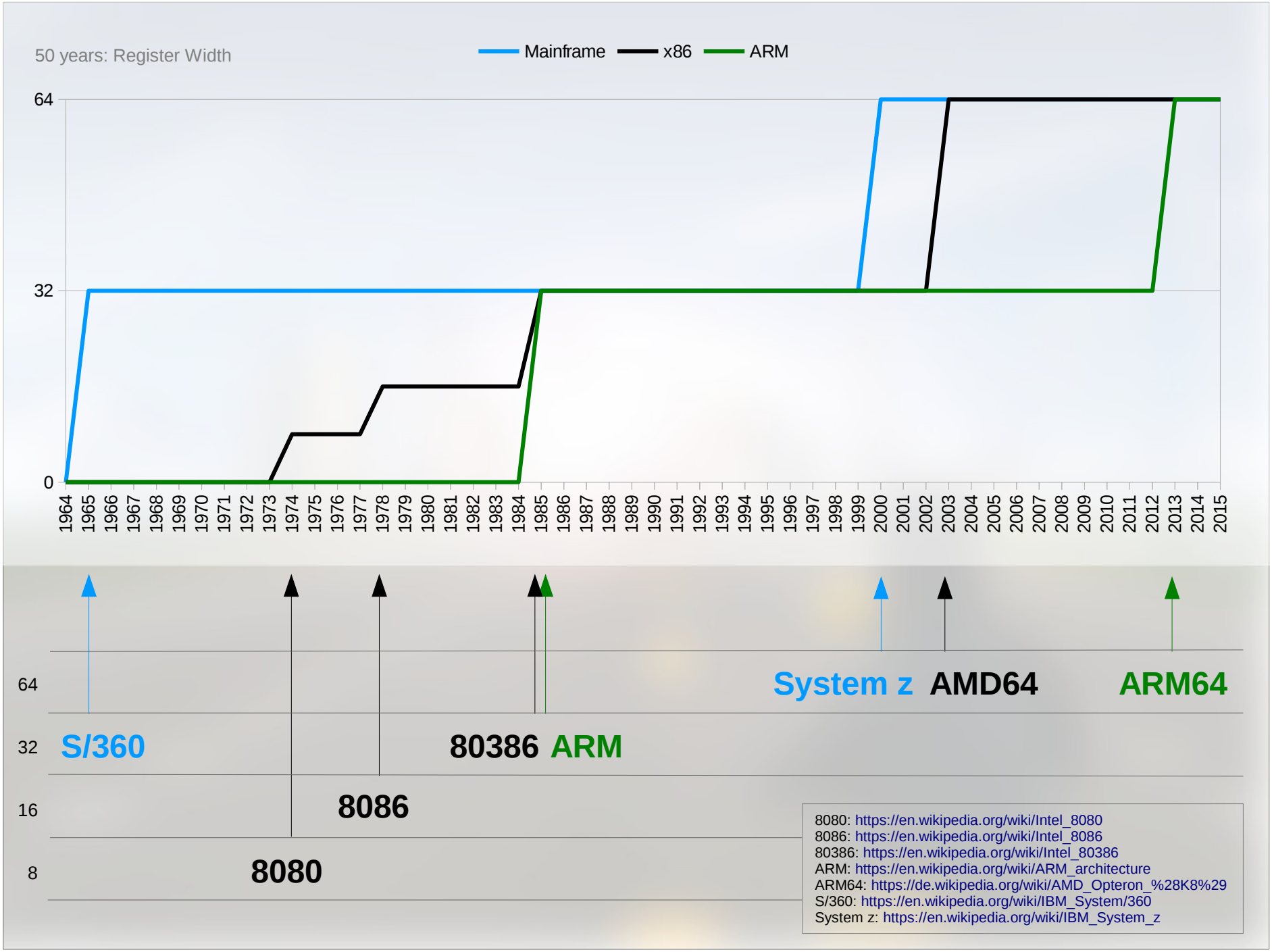
- 0: Use interpreter
- 1: Use JIT compiler
- 2: Use JIT compiler and print debug output

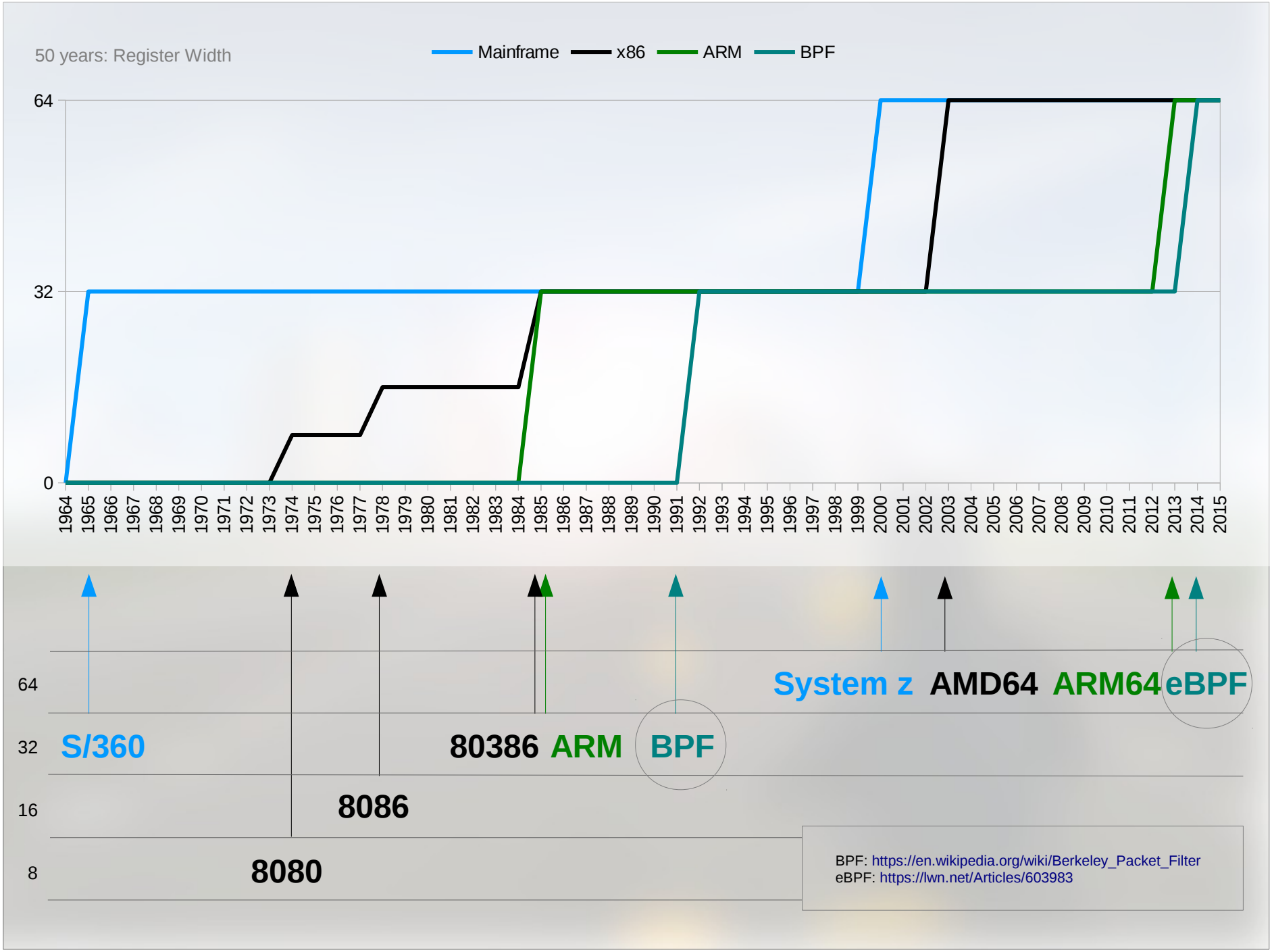
```
# echo 2 > /proc/sys/net/core/bpf_jit_enable
# tcpdump -i eth0 src client1
# dmesg

flen=10 proglen=188 pass=4 image=000003ff80016e10
JIT code: 00000000: eb 8f f0 58 00 24 b9 04 00 ...
...
JIT code: 000000b0: 00 00 00 1a 00 00 00 1c 00 00 ff ff
000003ff80016e10: eb8ff0580024  stmg    %r8,%r15,88(%r15)
000003ff80016e16: b90400ef      lgr     %r14,%r15
...
000003ff80016ea4: eb8ff0a80004  lmg     %r8,%r15,168(%r15)
000003ff80016eaa: 07fe          bcr     15,%r14
```


eBPF

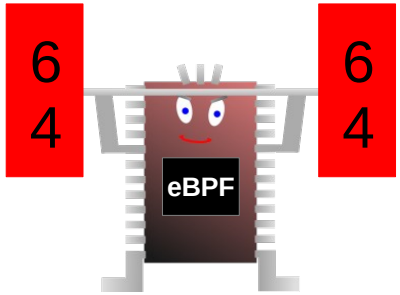
ISA History: Register Width


50 years: Register Width
Mainframe
x86
ARM
64
32
0
1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015
64
32
16
8
S/360
8080
8086
80386
ARM
System z
AMD64
ARM64
8080: https://en.wikipedia.org/wiki/Intel_8080
8086: https://en.wikipedia.org/wiki/Intel_8086
80386: https://en.wikipedia.org/wiki/Intel_80386
ARM: https://en.wikipedia.org/wiki/ARM_architecture
ARM64: https://de.wikipedia.org/wiki/AMD_Opteron_%28K8%29
S/360: https://en.wikipedia.org/wiki/IBM_System/360
System z: https://en.wikipedia.org/wiki/IBM_System_z


50 years: Register Width
Mainframe
x86
ARM
BPF
64
32
0
1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015
64
32
16
8
S/360
8080
8086
80386
ARM
BPF
System z
AMD64
ARM64
eBPF
BPF: https://en.wikipedia.org/wiki/Berkeley_Packet_Filter
eBPF: https://lwn.net/Articles/603983

eBPF - extended Berkley Packet Filter

- Linux 3.15 by Alexei Starovoitov <ast@kernel.org>
- Uses 64-bit registers
- New instructions:
  - 64 bit ALU plus old 32 bit ALU
  - Atomic: bpf_xadd
  - Function call: bpf_call
  - Endianness conversion functions
  - Singed compares
- First used only Linux internally
- BPF programs are converted into eBPF
- Now also externally usable with BPF system call
- LLVM eBPF backend



eBPF - 64 bit machine language (ISA)

- Eleven 64 bit registers:
  - B0: Return value from in-kernel function, and exit value for eBPF program
  - B1 - B5: Arguments from eBPF program to in-kernel function
  - B6 - B9: Callee saved registers that in-kernel function will preserve
  - B10: Read-only frame pointer to access stack
- Offset: off (s16)
- Immediate: imm (s32)

```
/usr/include/linux/bpf.h

struct bpf_insn {
        __u8    code;
        __u8    dst_reg:4;
        __u8    src_reg:4;
        __s16   off;
        __s32   imm;
};
```

op:8	**dst:4**	**src:4**	**off:16**
imm:32			

eBPF - Register mapping on s390x

eBPF register	s390x reg	s390x ABI	Notes
W0	R0	volatile	Work register (even)
W1	R1	volatile	Work register (odd)
B0	R14	volatile	• Exit value for eBPF program • Return value from in-kernel function • Return value for BPF_LD Has to be loaded after call: lgr %b0,r2
B1-B5	R2-R6	R2-R5 volatile R6 saved	• Parameter registers for kernel function calls • Scratch registers for BPF_LD SKB ops • B1 contains context parameter for BPF program (e.g. SKB pointer).
B6-B9	R7-R10	saved	Callee saved registers that in-kernel function will preserve
L	R11	saved	Literal pool
SKB_DATA	R12	saved	SKB data pointer
B10 (BFP)	R13	saved	Read-only frame pointer to access stack
	R15	saved	Stack pointer for C functions

eBPF - Converted BPF program

```
# tcpdump -i eth0 eth0 client1

BPF_LD  BPF_ABS BPF_H  src=0 dst=0 off=0 imm=12          # Load protocol type (2048 = IP)
                                                         # b0 = load_half(SKB, 12)
BPF_JMP BPF_JNE BPF_K  src=0 dst=0 off=3 imm=0x800       # if (b0 != 0x800)
                                                         #     goto [NOIP]
BPF_LD  BPF_ABS BPF_W  src=0 dst=0 off=0 imm=26          # b0 = load_word(SKB, 26)
                                                         # Load source address
                                                         # (14 + 12 = source IP address)
BPF_JMP BPF_JEQ BPF_K  src=0 dst=0 off=5 imm=0x090000001 # if (b0 == 9.0.0.1)
                                                         #     goto [FOUND]
BPF_JMP BPF_JA  BPF_K  src=0 dst=0 off=6 imm=0           # goto [NOT_FOUND]
BPF_JMP BPF_JEQ BPF_K  src=0 dst=0 off=1 imm=0x806       # [NOIP] if (b0 == 2054)
                                                         #     goto [ARP]
BPF_JMP BPF_JNE BPF_K  src=0 dst=0 off=4 imm=0x8035      # if (b0 != 32821)
                                                         #     goto [NOT_FOUND] (not RARP)
BPF_LD  BPF_ABS BPF_W  src=0 dst=0 off=0 imm=28          # [ARP] b0 = load_word(SKB, 28)
                                                         #       --> ARP IP source address
BPF_JMP BPF_JNE BPF_K  src=0 dst=0 off=2 imm=0x090000001 # if (b0 != 9.0.0.1)
                                                         #     goto [NOT_FOUND]
BPF_ALU BPF_MOV BPF_K  src=0 dst=0 off=0 imm=0xffff      # [FOUND] b0 = 0xffff
BPF_JMP BPF_EXI BPF_K  src=0 dst=0 off=0 imm=0           # [NOT_FOUND] exit
BPF_ALU BPF_MOV BPF_K  src=0 dst=0 off=0 imm=0           # [NOT FOUND] b0 = 0
BPF_JMP BPF_EXI BPF_K  src=0 dst=0 off=0 imm=0           # exit
```

eBPF - Converted BPF program

```
3ff80006312: eb67f0480024   stmg    %r6,%r7,72(%r15)          # Save registers on stack)
3ff80006318: ebbcf0700024   stmg    %r11,%r12,112(%r15)
3ff8000631e: ebeff0880024   stmg    %r14,%r15,136(%r15)
3ff80006324: b90400bf       lgr     %r11,%r15
3ff80006328: a7fbfda8       aghi    %r15,-600                 # Get stack space for BPF and function calls
3ff8000632c: e3b0f0980024   stg     %r11,152(%r15)            # Save backchain
3ff80006332: e31020800016   llgf    %r1,128(%r2)
3ff80006338: 5b102084       s       %r1,132(%r2)
3ff8000633c: e310f0a80024   stg     %r1,168(%r15)
3ff80006342: e3c020d80004   lg      %r12,216(%r2)
3ff80006348: b9040072       lgr     %r7,%r2                   # load input into %b6
3ff8000634c: c0310000000c   lgfi    %r3,12
3ff80006352: c01f00134624   llilf   %r1,1263140               # BPF_LD BPF_ABS BPF_H    src=0 dst=0 off=0 imm=12
3ff80006358: 0d61           basr    %r6,%r1
3ff8000635a: a7740043       brc     7,3ff800063e0
3ff8000635e: c01100000800   lgfi    %r1,2048                  # BPF_JMP BPF_JNE BPF_K   src=0 dst=0 off=3 imm=0x800
3ff80006364: b92100e1       clgr    %r14,%r1
3ff80006368: a7740014       brc     7,3ff80006390
3ff8000636c: c0310000001a   lgfi    %r3,26                    # BPF_LD BPF_ABS BPF_W    src=0 dst=0 off=0 imm=26
3ff80006372: c01f001345e0   llilf   %r1,1263072
3ff80006378: 0d61           basr    %r6,%r1
3ff8000637a: a7740033       brc     7,3ff800063e0
3ff8000637e: c0117f000001   lgfi    %r1,2415919105            # BPF_JMP BPF_JEQ BPF_K   src=0 dst=0 off=5 imm=0x090000001
3ff80006384: b92100e1       clgr    %r14,%r1
3ff80006388: a7840022       brc     8,3ff800063cc
3ff8000638c: a7f40025       brc     15,3ff800063d6            # BPF_JMP BPF_JA BPF_K    src=0 dst=0 off=6 imm=0
3ff80006390: c01100000806   lgfi    %r1,2054                  # BPF_JMP BPF_JEQ BPF_K   src=0 dst=0 off=1 imm=0x806
3ff80006396: b92100e1       clgr    %r14,%r1
3ff8000639a: a7840009       brc     8,3ff800063ac
3ff8000639e: c01100008035   lgfi    %r1,32821                 # BPF_JMP BPF_JNE BPF_K   src=0 dst=0 off=4 imm=0x8035
3ff800063a4: b92100e1       clgr    %r14,%r1
3ff800063a8: a7740017       brc     7,3ff800063d6
3ff800063ac: c0310000001c   lgfi    %r3,28                    # BPF_LD BPF_ABS BPF_W    src=0 dst=0 off=0 imm=28
3ff800063b2: c01f001345e0   llilf   %r1,1263072
3ff800063b8: 0d61           basr    %r6,%r1
3ff800063ba: a7740013       brc     7,3ff800063e0
3ff800063be: c0117f000001   lgfi    %r1,2415919105            # BPF_JMP BPF_JNE BPF_K   src=0 dst=0 off=2 imm=0x090000001
3ff800063c4: b92100e1       clgr    %r14,%r1
3ff800063c8: a7740007       brc     7,3ff800063d6
3ff800063cc: c0ef0000ffff   llilf   %r14,65535                 # BPF_ALU BPF_MOV BPF_K   src=0 dst=0 off=0 imm=0xffff
3ff800063d2: a7f40009       brc     15,3ff800063e4            # BPF_JMP BPF_EXIT BPF_K  src=0 dst=0 off=0 imm=0
3ff800063d6: c0ef00000000   llilf   %r14,0                     # BPF_ALU BPF_MOV BPF_K   src=0 dst=0 off=0 imm=0
3ff800063dc: a7f40004       brc     15,3ff800063e4            # BPF_JMP BPF_EXIT BPF_K  src=0 dst=0 off=0 imm=0
3ff800063e0: a7e90000       lghi    %r14,0                    # Exit 0 label
3ff800063e4: b904002e       lgr     %r2,%r14                  # Load return value
3ff800063e8: eb67f2a00004   lmg     %r6,%r7,672(%r15)         # Load saved register from stack
3ff800063ee: ebbcf2c80004   lmg     %r11,%r12,712(%r15)
3ff800063f4: ebeff2e00004   lmg     %r14,%r15,736(%r15)
3ff800063fa: 07fe           bcr     15,%r14                   # Return to caller
```

eBPF - Converted BPF program

```
3ff80006312: eb67f0480024   stmg    %r6,%r7,72(%r15)          # Save registers on stack)
3ff80006318: ebbcf0700024   stmg    %r11,%r12,112(%r15)
3ff8000631e: ebeff0880024   stmg    %r14,%r15,136(%r15)
3ff80006324: b90400bf       lgr     %r11,%r15
3ff80006328: a7fbfda8       aghi    %r15,-600                 # Get stack space for BPF and function calls
3ff8000632c: e3b0f0980024   stg     %r11,152(%r15)            # Save backchain
3ff80006332: e31020800016   llgf    %r1,128(%r2)
3ff80006338: 5b102084       s       %r1,132(%r2)
3ff8000633c: e310f0a80024   stg     %r1,168(%r15)
3ff80006342: e3c020d80004   lg      %r12,216(%r2)
3ff80006348: b9040072       lgr     %r7,%r2                   # load input into %b6
3ff8000634c: c0310000000c   lgfi    %r3,12
3ff80006352: c01f00134624   llilf   %r1,1263140               # BPF_LD BPF_ABS BPF_H    src=0 dst=0 off=0 imm=12
3ff80006358: 0d61           basr    %r6,%r1
3ff8000635a: a7740043       brc     7,3ff800063e0
3ff8000635e: c01100000800   lgfi    %r1,2048                  # BPF_JMP BPF_JNE BPF_K   src=0 dst=0 off=3 imm=0x800
3ff
3ff
3ff
3ff
3ff
3ff
3ff
3ff
3ff
3ff
3ff
3ff
3ff800063a4: b92100e1       clgr    %r14,%r1
3ff800063a8: a7740017       brc     7,3ff800063d6
3ff800063ac: c0310000001c   lgfi    %r3,28                    # BPF_LD BPF_ABS BPF_W    src=0 dst=0 off=0 imm=28
3ff800063b2: c01f001345e0   llilf   %r1,1263072
3ff800063b8: 0d61           basr    %r6,%r1
3ff800063ba: a7740013       brc     7,3ff800063e0
3ff800063be: c0117f000001   lgfi    %r1,2415919105            # BPF_JMP BPF_JNE BPF_K   src=0 dst=0 off=2 imm=0x090000001
3ff800063c4: b92100e1       clgr    %r14,%r1
3ff800063c8: a7740007       brc     7,3ff800063d6
3ff800063cc: c0ef0000ffff   llilf   %r14,65535                 # BPF_ALU BPF_MOV BPF_K   src=0 dst=0 off=0 imm=0xffff
3ff800063d2: a7f40009       brc     15,3ff800063e4            # BPF_JMP BPF_EXIT BPF_K  src=0 dst=0 off=0 imm=0
3ff800063d6: c0ef00000000   llilf   %r14,0                     # BPF_ALU BPF_MOV BPF_K   src=0 dst=0 off=0 imm=0
3ff800063dc: a7f40004       brc     15,3ff800063e4            # BPF_JMP BPF_EXIT BPF_K  src=0 dst=0 off=0 imm=0
3ff800063e0: a7e90000       lghi    %r14,0                    # Exit 0 label
3ff800063e4: b904002e       lgr     %r2,%r14                  # Load return value
3ff800063e8: eb67f2a00004   lmg     %r6,%r7,672(%r15)         # Load saved register from stack
3ff800063ee: ebbcf2c80004   lmg     %r11,%r12,712(%r15)
3ff800063f4: ebeff2e00004   lmg     %r14,%r15,736(%r15)
3ff800063fa: 07fe           bcr     15,%r14                   # Return to caller
```

```
if (b0 != 9.0.0.1) skip 2

lgfi  %r1,2415919105 # BPF_JMP BPF_JNE BPF_K dst=0 off=2 imm=0x090000001
clgr  %r14,%r1       # Note: R14: Mapped to B0
brc   7,3ff800063d6  #       R1:  Work register
llilf %r14,65535     # BPF_ALU BPF_MOV BPF_K dst=0 imm=0xffff
```

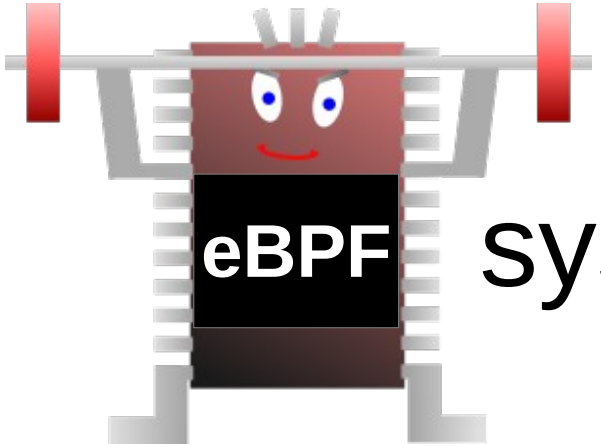
Performance: One filter run

```
# tcpdump -i lo src localhost
# ping -f localhost
```

BPF engine	Instructions (executed)	z196	EC12	z13
Interpreter	117	40 ns	30 ns	20 ns
eBPF JIT	55	55 ns	17 ns	8 ns

Notes:
Used STCK before and after filter call to measure time
Test done on development system with (not yet final) JIT.

Reason: Branch prediction for jumps from low kernel to high module addresses



system call

eBPF system call

- Integrated in Linux 3.19 (Febr. 2015)
- Useful for creating kernel statistics
- The eBPF system call is a multiplexer with the following functions:
  - BPF map functions: Create, iterate, lookup
  - BPF load function: Load eBPF program
  - Attach maps to eBPF programs
- With a new socket call the eBPF program can be attached to a socket:
  - setsockopt(sockfd, SOL_SOCKET, SO_ATTACH_BPF, &prog_id,...);

eBPF system call - Scenario

Task: Count network packets per protocol

T-MAC	S-MAC	T/L	DATA	FCS
6 Byte	6 Byte	2 Byte	46–1500 Byte	4 Byte
target MAC	source MAC	Type or Length	data	Frame Check Sequence

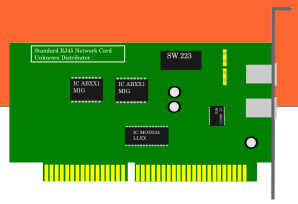
0	4	8	12	16	20	24	28	31 Bit
Version	IHL	TOS		Total Length				
Identification				Flags	Fragment Offset			
TTL		Protocol		Header Checksum				
Source Address								
Destination Address								
Options and Padding (optional)								

One byte protocol field in IP header

eBPF system call - Scenario

```
[server] # sockex1_user
```

kernel



eBPF system call - Scenario

```
[server] # sockex1_user
```

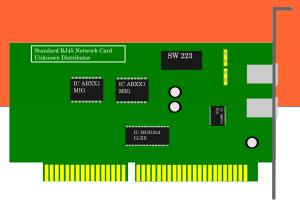
bpf_create_map()

kernel

map:

tcp:	0
udp:	0
icmp:	0

256 Elements



eBPF system call - Scenario

```
[server] # sockex1_user
```

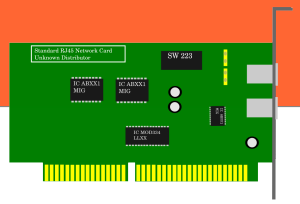
eBPF program: e370f050 ...

```
map[ip[prot]]++
```

kernel

map:

tcp:	0
udp:	0
icmp:	0



eBPF system call - Scenario

```
[server] # sockex1_user
```

```
prog_id = bpf_prog_load()
```

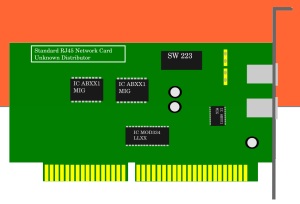
kernel

map:

tcp:	0
udp:	0
icmp:	0

eBPF program: e370f050 ...

```
map[ip[prot]]++
```



eBPF system call - Scenario

```
[server] # sockex1_user
```

```
setsockopt(sockfd, SOL_SOCKET,
           SO_ATTACH_BPF, &prog_id)
```

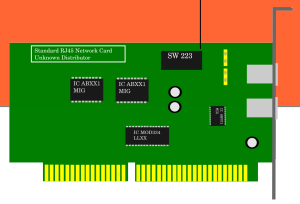
kernel

map:

tcp:	0
udp:	0
icmp:	0

eBPF program: e370f050 ...

```
map[ip[prot]]++
```



eBPF system call - Scenario

```
[server] # sockex1_user
```

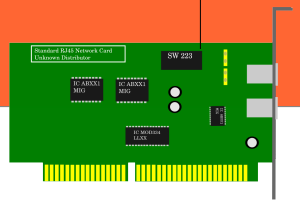
kernel

map:

tcp:	0
udp:	0
icmp:	***1***

```
eBPF program: e370f050 ...

map[ip[prot]]++
```



ICMP packet 1

```
[client] # ping server
```

eBPF system call - Scenario

```
[server] # sockex1_user
```

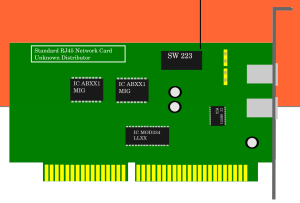
kernel

map:

tcp:	0
udp:	0
icmp:	***2***

```
eBPF program: e370f050 ...

map[ip[prot]]++
```



ICMP packet 2

```
[client] # ping server
```

eBPF system call - Scenario

```
[server] # sockex1_user
```

`bpf_lookup_elem(ICMP) = 2`

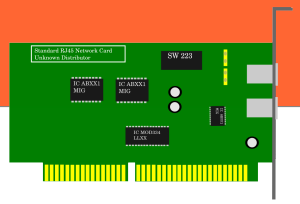
kernel

map:

tcp:	0
udp:	0
icmp:	***2***

eBPF program: e370f050 ...

`map[ip[prot]]++`



eBPF system call - Scenario

```
[server] # sockex1_user
```

TCP 0 UDP 0 **ICMP 2** packets

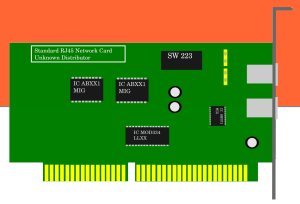
kernel

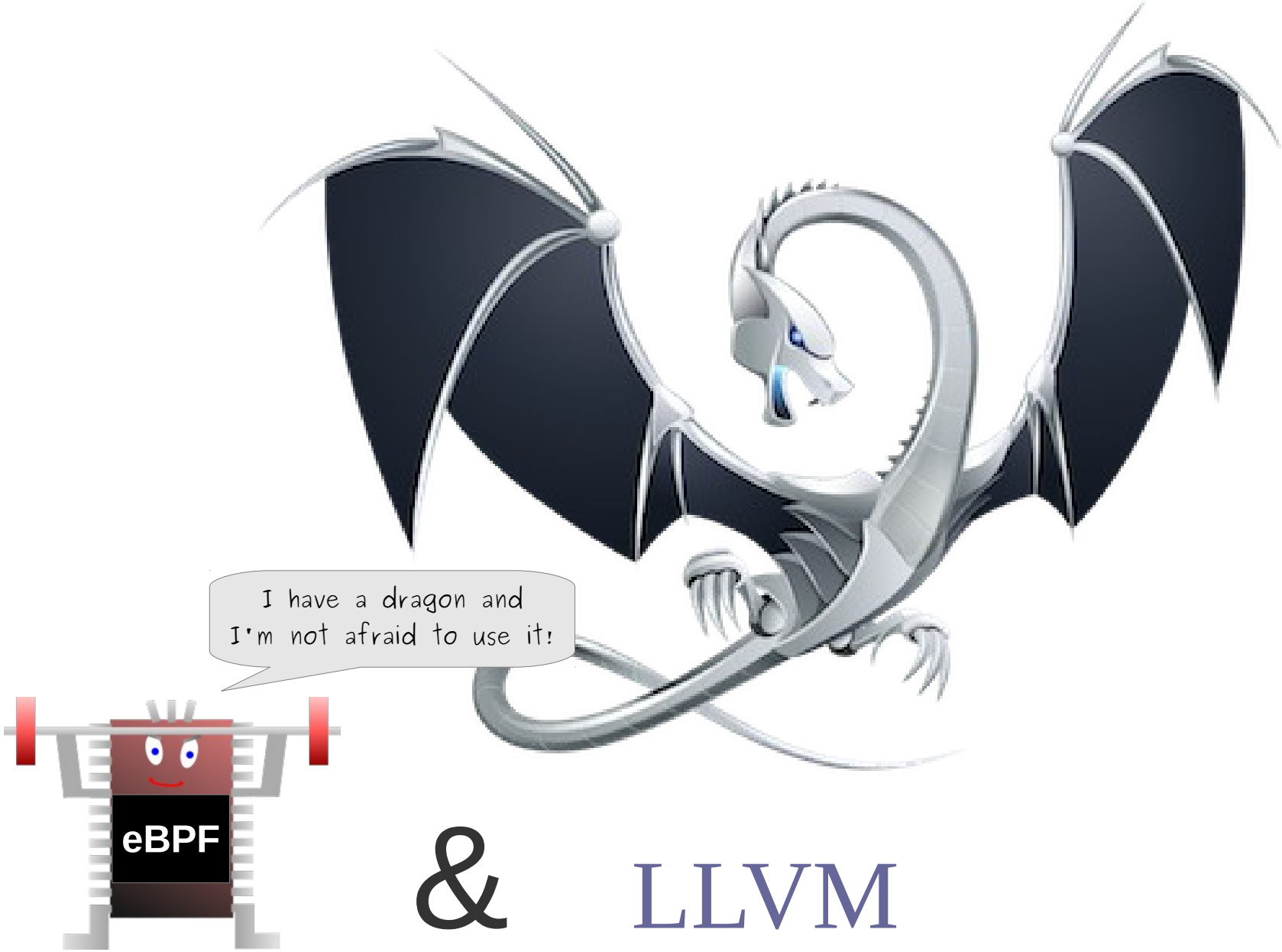
map:

tcp:	0
udp:	0
icmp:	***2***

eBPF program: e370f050 ...

```
map[ip[prot]]++
```





& LLVM

LLVM eBPF backend (kernel part)

```
/* Count packets of different protocols */

#include <uapi/linux/bpf.h>
#include <uapi/linux/if_ether.h>
#include <uapi/linux/ip.h>
#include "bpf_helpers.h"

struct bpf_map_def SEC("maps") my_map = {
        .type = BPF_MAP_TYPE_ARRAY,
        .key_size = sizeof(u32),
        .value_size = sizeof(long),
        .max_entries = 256,
};

SEC("socket1")
int bpf_prog1(struct sk_buff *skb)
{
        int index = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));
        long *value;

        value = bpf_map_lookup_elem(&my_map, &index);
        if (value)
                __sync_fetch_and_add(value, 1);

        return 0;
}
char _license[] SEC("license") = "GPL";
```

LLVM eBPF backend (userspace part)

```
int main(int ac, char **argv)
{
    FILE *f;
    int i, sock;

    /* Load eBPF code and create map */
    load_bpf_file("sockex1_kern.o");
    sock = open_raw_sock("lo");
    setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF,
               prog_fd, sizeof(prog_fd[0]);

    for (i = 0; i < 5; i++) {
        long long tcp_cnt, udp_cnt, icmp_cnt;
        int key;

        key = IPPROTO_TCP;
        bpf_lookup_elem(map_fd[0], &key, &tcp_cnt);
        key = IPPROTO_UDP;
        bpf_lookup_elem(map_fd[0], &key, &udp_cnt);
        key = IPPROTO_ICMP;
        bpf_lookup_elem(map_fd[0], &key, &icmp_cnt);

        printf("TCP %lld UDP %lld ICMP %lld packets\n",
                       tcp_cnt, udp_cnt, icmp_cnt);
        sleep(1);
    }
    return 0;
}
```

```
# ./sockex1
TCP 0 UDP 0 ICMP 0 packets
TCP 0 UDP 0 ICMP 4 packets
TCP 0 UDP 0 ICMP 8 packets
TCP 0 UDP 0 ICMP 12 packets
TCP 0 UDP 0 ICMP 16 packets
```

Other BPF and eBPF exploiters

- **xt_bpf**: Kernel module for iptables
  - Match rules with BPF

- **Traffic control:**
  - **cls_bpf**: Kernel module for classifier for traffic shaping
  - **act_bpf:** BPF based action (since Linux 4.1)

- **seccomp-bpf:** Secure computing (since Linux 3.5)
  - Syscall filter for sandboxes (e.g. chromium browser or docker)

- **Kprobes support** (Linux 4.1)

Availability for Linux on z Systems

- **Kernel: BPF**
  - Upstream: Linux 3.7
  - SLES12.0
  - RHEL7.0
- **Kernel: eBPF**
  - Upstream: Linux 4.1
- **LLVM**
  - Upstream: 3.7
  - git commit: ac73683b1 (“[bpf] add big- and host- endian support“)


Thank you!

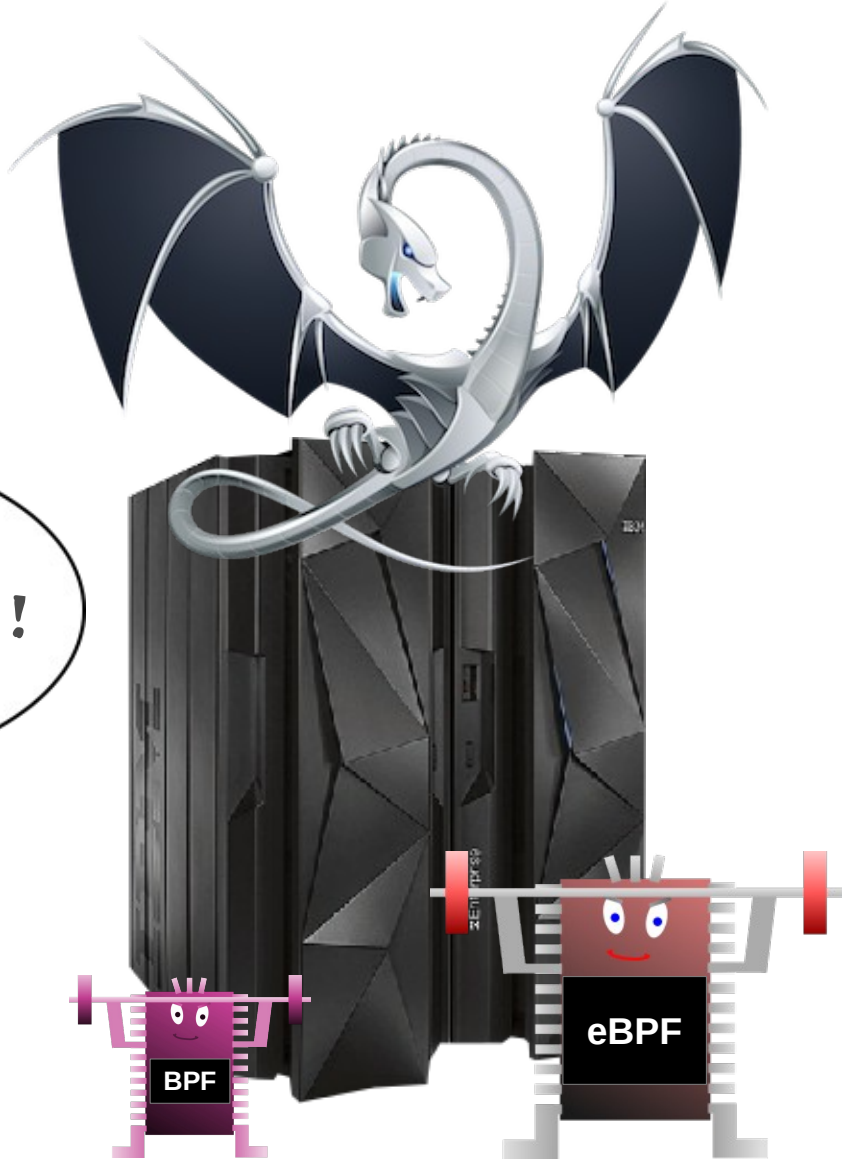

BPF
eBPF

Image sources

- Network card (Public Domain): https://openclipart.org/detail/97009/network-card
- Cross (Public Domain): https://openclipart.org/detail/16982/cross-by-jean_victor_balin
- OK (Public Domain): https://openclipart.org/detail/23156/ok-by-dholler
- Computer (Public Domain): https://openclipart.org/detail/25340/Computer_1-by-And
- Wizard (IBM): https://ibm.biz/BdHd9t
- White Board (Public Domain): https://openclipart.org/detail/32389/white-board
- LLVM logo dragon (Apple Inc.): http://llvm.org/Logo.html
- BPF CPU (Public Domain): https://openclipart.org/detail/28105/processoractive
- Flower (Public Domain): https://openclipart.org/detail/172600/geraldton-wav
- Text Bubble (Public Domain): https://openclipart.org/detail/48421/talk-bubble