[KISTI 기술문서] ISBN 978-89-294-1799-4

리눅스 eBPF 기술 분석

2024. 12. 1.

한국과학기술정보연구원
슈퍼컴퓨팅기술개발센터

저자소개

김상완

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
sangwan@kisti.re.kr

정기문

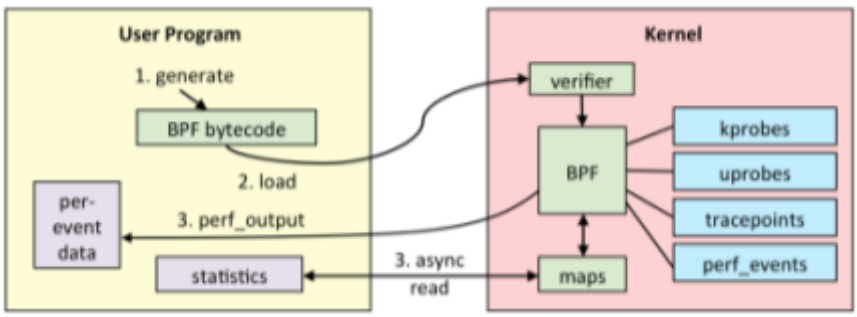
한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
kmjeong@kisti.re.kr

이 기술보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24L2M1C6
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

목 차

1. 개요

- Berkeley Packet Filter(BPF)[1]는 Linux 운영 체제가 겨우 1년밖에 되지 않은 1992년에 시작된 Unix 유사 운영체제용 기술이다. 개발 목적은 이름에서 알 수 있듯이 네트워크 패킷 필터링을 효율적으로 수행하기 위함이었다. 커널 공간에서 네트워크 패킷을 필터링하기 위해 사용자 공간 응용 프로그램이 커널에서 직접 실행할 수 있는 간단한 필터링 프로그램을 정의할 수 있게 하였다.
- 역사가 오래된 만큼 BPF 필터링 메커니즘은 대부분의 Unix 계열 운영 체제에서 사용할 수 있다. FreeBSD, NetBSD, WinPcap(Windows)과 같은 플랫폼에서도 유사한 기능을 제공한다.
- 네트워크 패킷 필터의 기능을 뛰어넘는 기술인 DTrace(디트레이스)는 Sun Microsystems에 의해 Solaris용으로 개발되어(2005년 공식 출시), 이후 다른 Unix 계열 시스템으로 이식되었다. DTrace는 메모리, CPU, 파일 시스템 및 활성 프로세스에 사용하는 네트워크와 같은 시스템 전반적인 개요를 얻는데 사용가능하다. 이러한 확장된 트레이싱 기능은 리눅스의 BPF에도 영향을 주었다고 할 수 있다. BPF의 기능을 확장한 eBPF가 2014년 커널 3.18에 처음 도입되었다. 기능이 크게 추가된 eBPF는 리눅스 커널 4.x 이상에서만 사용가능하다.
- BPF는 사용자가 정의한 규칙을 컴파일하여 BPF 바이트코드(bytecode)로 변환하고, 이를 통해 네트워크 인터페이스에서 오는 패킷을 필터링 한다. 컴파일된 바이트코드는 BPF 머신에서 실행되고, 처리 결과 필터링 함수가 True(non-zero)를 반환하면 패킷은 ACCEPT되어 사용자 공간 애플리케이션으로 전달된다.
- BPF 머신은 32비트 누산기와 레지스터로 제한적이었으나. eBPF는 BPF의 확장 버전으로 더 큰 64비트 레지스터와 스택, 맵 등을 도입하여 기능을 확장함. 패킷 필터링을 넘어 리눅스 시스템의 관측을 위한 도구로 발전됨. eBPF 동작 흐름도:



- eBPF는 다양한 오픈소스 프로젝트와 상용 솔루션에 활용되고 있다:
  * Cilium [2]: 컨테이너 네트워킹 및 보안을 위한 오픈소스 소프트웨어. Kubernetes와 연동
  * Falco [3] 클라우드 네이티브 런타임 보안 도구, 컨테이너와 호스트 시스템의 행동을 모니터링
  * Katran [4] 페이스북에서 개발한 L4 로드 밸런서, eBPF를 사용하여 고성능 네트워크 로드 밸런싱 구현
  * Hubble [5] Cilium의 네트워킹 및 보안 관찰 플랫폼. 네트워크 플로우와 보안 이벤트를 모니터링
  * Pixie [6] Kubernetes 클러스터의 애플리케이션 수준 관찰성 제공
  * Tracee [7] 시스템 이벤트 추적 및 런타임 보안 도구
- 본 문서에서 사용된 예제코드는 다음 깃허브 주소에서 찾을 수 있다:

```
git clone https://github.com/swkim85/linux-drill ;  cd linux-drill/ebpf
```

1) https://en.wikipedia.org/wiki/Berkeley_Packet_Filter
2) https://cilium.io/
3) https://falco.org/
4) https://github.com/facebookincubator/katran
5) https://docs.cilium.io/en/stable/overview/intro/
6) https://px.dev/
7) https://aquasecurity.github.io/tracee/v0.6.4/

2. DTrace 튜토리얼

- FreeBSD의 DTrace[8]는 원래 Solaris에서 개발되었으며, FreeBSD로 포팅되었음.
- 본 절에서는 VirtualBox에서 FreeBSD 14.1을 설치하여 테스트하였음.
- DTrace Tutorial [9]을 참고한다.

```
# more /etc/os-release
NAME=FreeBSD
VERSION="14.1-RELEASE"
VERSION_ID="14.1"
ID=freebsd
[...]
# uname -a
FreeBSD freebsd14 14.1-RELEASE FreeBSD 14.1-RELEASE releng/14.1-n267679-10e31f0946d8 GENERIC amd64
  dtrace 버전 확인
# dtrace -V
dtrace: Sun D 1.13
```

※ FreeBSD의 DTrace 소스 코드는 FreeBSD 소스 트리내에 존재함

```
# cd /usr/src
# find . -name "*dtrace*"
# ls sys/modules/dtrace/ [10]
# ls cddl/contrib/opensolaris/cmd/dtrace/ [11]
```

2.1. DTrace 프로브

- 프로브 리스트 조회하기

```
  -l 옵션은 모든 프로브를 출력한다. probe란 event data를 캡쳐(capture)하기 위한 측정점(instrume
ntation point) 이라고 할 수 있음. 출력되는 5개의 필드는 다음과 같다: probe ID, prover name, mod
ule name, function name, probe name
# dtrace -l
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
    2     dtrace                                                     END
    3     dtrace                                                     ERROR
    4        fbt            kernel                camamstatusentrycomp entry
    5        fbt            kernel                camamstatusentrycomp return
[...]
# dtrace -l -n  syscall:::     # syscall provider 만 출력

  provider의 종류를 출력하기
# dtrace -l ¦ awk '{print $2}' ¦ sort ¦ uniq -c ¦ sort -nr
78698 fbt
2320 syscall
 874 dtmalloc
 242 mac_framework
 185 vfs
 172 nfscl
[...]
```

8) https://en.wikipedia.org/wiki/DTrace
9) https://wiki.freebsd.org/DTrace/Tutorial
10) https://cgit.freebsd.org/src/tree/sys/modules/dtrace
11) https://cgit.freebsd.org/src/tree/cddl/contrib/opensolaris/cmd/dtrace

- DTrace에는 다양한 종류의 제공자(provider)가 있다. 제공자의 종류를 정리하면 다음과 같다:

DTrace Provider	Description
fbt	(function boundary tracing) 커널 함수의 진입과 반환 지점을 추적
syscall	시스템 콜의 진입과 반환을 추적, 사용자 공간과 커널 공간 사이의 상호작용을 모니터링
profile	주기적으로 실행된느 프로브를 제공. 시스템 상태를 주기적으로 샘플링하기 위한 용도
sdt	(statically defined tracing) 개발자가 코드에 명시적으로 정의한 정적 트레이스 포인트를 제공
io	디스크 I/O 작업을 추적
proc	프로세스 생성, 종료, 실행 등의 이벤트를 추적
sched	CPU 스케줄링 이벤트를 추적
vminfo	가상 메모리 시스템 이벤트를 추적
lockstat	커널 동기화 프리미티브(락, 뮤텍스 등) 사용을 추적
priv	시스템에서 권한(privilege) 관련 이벤트를 추적

2.2. DTrace 사용하기

- DTrace 기본 사용법

```
  프로브 이름은 콜론(:)으로 구분된 4개의 필드로 구성됨. (provider:module:function:name)
  {} 의 내용은 action으로 프로브와 연결된다.

# dtrace -n 'dtrace:::BEGIN { printf("Hello FreeBSD!\n"); }'
dtrace: description 'dtrace:::BEGIN ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  3      1                           :BEGIN Hello FreeBSD!
^C
  파일 오픈에 관하여 추적을 시작함.  프로세스 이름과 파일의 경로를 출력한다.
 execname은 builtin 변수[12]로 현재 프로세스의 이름을 의미함. pid, tid, ppid 도 builtin 변수에 해당함. arg0는 첫번째 probe argument를 의미함.
 open() 시스템호출[13]의 arguments는 (const char *pathname, int flags, mode_t mode) 임.
  copyinstr()은 user-level string을 kernel 로 복사하여 DTrace가 읽을 수 있도록 한다.
# dtrace -n 'syscall::open*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
CPU     ID                    FUNCTION:NAME
[...]    다른 터미털에서 ls 명령을 실행함.
                                        네번째 컬럼이 execname, 5번째컬럼이 arg0 임
  0  80487                      open:entry ls /etc/libmap.conf
  0  80487                      open:entry ls /usr/local/etc/libmap.d
  0  80487                      open:entry ls /var/run/ld-elf.so.hints
  0  80487                      open:entry ls /lib/libutil.so.9
  0  80487                      open:entry ls /lib/libtinfow.so.9
  0  80487                      open:entry ls /lib/libc.so.7
  0  80487                      open:entry ls /usr/share/locale/C.UTF-8/LC_CTYPE
  0  80487                      open:entry ls .
  0  80487                      open:entry ls .
  0  80487                      open:entry ls .
  시스템 호출에 관한 추적을 시작
# dtrace -n 'syscall:::entry { printf("%s %s", execname, probefunc); }'
dtrace: description 'syscall:::entry ' matched 1160 probes
CPU     ID                    FUNCTION:NAME
```

```
  3  80585                       ioctl:entry dtrace ioctl
  3  80585                       ioctl:entry dtrace ioctl
  3  81139                sigprocmask:entry dtrace sigprocmask
[...]
^C
   카운터(cnt)를 이용하여 처음 10개의 이벤트만 출력하기
# dtrace -n 'dtrace:::BEGIN { cnt=0; } syscall:::entry { printf("%s %s", execname, probefunc);
           cnt++; if (cnt >= 10) exit(0); }'
dtrace: description 'dtrace:::BEGIN ' matched 4642 probes
CPU     ID                    FUNCTION:NAME
  1  80585                       ioctl:entry dtrace ioctl
  1  80585                       ioctl:entry dtrace ioctl
  1  81139                sigprocmask:entry dtrace sigprocmask
  1  81291                  sigaction:entry dtrace sigaction
  1  81139                sigprocmask:entry dtrace sigprocmask
  1  81139                sigprocmask:entry dtrace sigprocmask
  1  81291                  sigaction:entry dtrace sigaction
  1  81139                sigprocmask:entry dtrace sigprocmask
  1  81139                sigprocmask:entry dtrace sigprocmask
  1  81291                  sigaction:entry dtrace sigaction
  1  81291                  sigaction:entry

  아래와 같이 D program의 내용을 파일(.d)로 만들어서 실행(dtrace -s)하여도 결과는 같다.
# more  syscall_ten.d
dtrace:::BEGIN {
 cnt=0;
}
syscall:::entry {
  printf("%s %s", execname, probefunc);
  cnt++;
  if (cnt >= 10) exit(0);
}
# dtrace -s  syscall_ten.d
dtrace: script 'syscall_ten.d' matched 4642 probes
[...]

  시스템 호출 횟수에 대한 통계를 수집한다.
  ^C를 누를 때까지 syscall의 name과 type을 기준으로 카운트를 계산.
 @는 aggregation을 의미하는 특별한 변수로 데이터 통계를 수집한다.
 []는 aggregation을 할때 사용되는 key를 제공한다.
 probefunc는 builtin variable로 프로브의 function name을 의미함.
 count()는 aggregation을 수행하는 함수임.
# dtrace -n 'syscall:::entry { @[execname, probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 1160 probes
^C    잠시 대기후 ^C를 누르면 결과가 출력됨
  첫번째 컬럼은 실행파일의 이름  두번째 컬럼은 probfunc            세번째 컬럼은 count값
  devd                          read                                 1
  devd                          select                               1
  dtrace                        fstat                                1
  dtrace                        sigreturn                            1
  dtrace                        write                                1
  sshd                          ioctl                                1
[...]
  dtrace                        __sysctlbyname                       5
  sshd                          sigprocmask                          8
```

```
  dtrace                          sigaction                                 10
  dtrace                          clock_gettime                             16
  sshd                            clock_gettime                             18
  dtrace                          sigprocmask                               21
  dtrace                          ioctl                                     38
```

-c cmd 옵션을 사용하여 특정 명령을 실행하고, 명령이 끝나면 트레이스를 종료할 수 있다.

```
# dtrace -c 'cat /etc/passwd' -n 'syscall:::entry { @[execname, probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 1160 probes
root:*:0:0:Charlie &:/root:/bin/sh
[...]  /etc/passwd 내용이 출력됨
dtrace: pid 6220 has exited    커맨드로 수행한 프로세스가 종료되고, dtrace 결과가 출력됨
  cat               copy_file_range                                         1
  cat               exit                                                    1
  cat               mmap                                                    1
[...]
```

- 히스토그램을 이용한 추적

read() 시스템 호출의 리턴 값을 요약하여 히스토그램(histogram)으로 출력한다.
/.../는 predicate 로 action을 위한 filter에 해당한다. action은 predicate 표현식이 참(true)일 경우만 실행된다. 논리 연산자("&&", "||")를 사용할 수 있다.
quantize()함수는 집계함수(aggregation function)로 power-of-two 히스토그램으로 요약한다.
다른 집계함수로는 lquantize(), avg(), min(), max() 등이 있다.

```
# dtrace -n 'syscall::read:return /execname == "sshd"/ { @ = quantize(arg0); }'
dtrace: description 'syscall::read:return ' matched 2 probes
^C    다른 터미널에서 몇가지 명령을 수행한 후 C^로 종료하면 결과가 출력됨
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@                                 10
               2 |@@@                                      4
               4 |                                         0
               8 |@@@                                      4
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@                        20
              64 |@@                                       2
             128 |                                         0
             256 |                                         0
             512 |@@@                                      4
            1024 |@@                                       2
            2048 |@@@                                      4
            4096 |                                         0
```

read() 시스템 호출에 소요된 시간을 nanoseconds 단위로 히스토그램을 출력.
self-> 는 스레드 로컬 변수를 나타낸다. syscall의 시작 시간을 ts 라는 변수에 저장함.
timestamp는 부팅 이후 나노초를 나타내는 고해상도 타임스탬프 카운터.
/self->ts/ 는 ts가 NULL 또는 0이 아닌지 확인하기 위한 predicate 임.

```
# cat read_syscall_duration.d
syscall::read:entry { self->ts = timestamp; }
syscall::read:return /self->ts/ { @ = quantize(timestamp - self->ts); self->ts = 0; }
# dtrace  -s read_syscall_duration.d
dtrace: script 'read_syscall_duration.d' matched 4 probes
dtrace: 16 dynamic variable drops with non-empty rinsing list
dtrace: 119 dynamic variable drops with non-empty dirty list
```

12) https://docs.oracle.com/cd/E18752_01/html/819-5488/gcfpz.html
13) https://man7.org/linux/man-pages/man2/open.2.html

```
dtrace: 6 dynamic variable drops with non-empty rinsing list
           value  ------------- Distribution ------------- count
             128 ¦                                         0
             256 ¦                                         4
             512 ¦                                         6
            1024 ¦@                                        213
            2048 ¦@@@@@@                                   2001
            4096 ¦@@@                                      1248
            8192 ¦@                                        244
           16384 ¦@@@@@@@@@@@                             3947
           32768 ¦@@@@@@@@@@                              3068
           65536 ¦@@@@@                                   1935
          131072 ¦@                                       281
          262144 ¦@@                                      733
          524288 ¦@                                       350
         1048576 ¦                                        156
         2097152 ¦                                        80
         4194304 ¦                                        8
         8388608 ¦                                        2
        16777216 ¦                                        1
        33554432 ¦                                        0
        67108864 ¦                                        0
       134217728 ¦                                        1
       268435456 ¦                                        0
```

```
  read() 호출에서 사용된 CPU 시간 측정.
 vtimestamp 카운터는 현재 스레드가 CPU에서 실행될때만 카운트가 증가한다.
 단위는 CPU시간의 나노초단위임.
 lquantize()는 linear quantize함수로 lquantize(value, min, max, step)을 의미함.
 “On-CPU us"는 출력내용에 라벨을 붙기기 위한 방법이다.

# cat read_syscall_cputime.d
syscall::read:entry { self->vts = vtimestamp; }
syscall::read:return /self->vts/   {
 @["On-CPU us:"] = lquantize((vtimestamp - self->vts) / 1000, 0, 10000, 10);
 self->vts = 0;
}

# dtrace -s read_syscall_cputime.d
dtrace: script 'read_syscall_cputime.d' matched 4 probes
dtrace: 310 dynamic variable drops with non-empty dirty list
dtrace: 1335 dynamic variable drops with non-empty dirty list
dtrace: 1180 dynamic variable drops with non-empty dirty list
dtrace: 1331 dynamic variable drops with non-empty dirty list
^C
  On-CPU us:
           value  ------------- Distribution ------------- count
             < 0 ¦                                         0
               0 ¦@@@@@@@@@@@@@@@                        40
              10 ¦                                         0
              20 ¦@@@@@@@@@@@                            32
              30 ¦@@@@@@@@@@@                            33
              40 ¦                                         0
              50 ¦                                         0
              60 ¦@                                        4
              70 ¦@                                        2
              80 ¦                                         1
              90 ¦                                         1
```

```
           100 |                                         0   [...]
 5초동안 프로세스 수준의 이벤트를 계산하여 요약을 출력함.
proc 제공자는 프로세스 및 스레드 생성 및 파괴와 같은 상위 수준 프로세스 이벤트를 갖는다.
dtrace -l -P proc 명령을 이용하여 해당 프로브를 나열할 수 있다.
probename은 프로브의 이름임.
tick-5s는 profile:::tick-5s의 단축형으로, 5초마다 한 CPU에서만 실행되는 프로필 제공자임.
exit(0)는 Dtrace를 종료함

# dtrace -l -P proc     또는 dtrace -ln 'proc:::'
   ID   PROVIDER            MODULE                          FUNCTION NAME
78712       proc            kernel                              none exec
78713       proc            kernel                              none exec-failure
78714       proc            kernel                              none exec-success
78715       proc            kernel                              none exit
78716       proc            kernel                              none create
78760       proc            kernel                              none signal-send
78761       proc            kernel                              none signal-clear
78762       proc            kernel                              none signal-discard
78764       proc            kernel                              none lwp-exit

# cat probe_count_5sec.d
proc::: { @[probename] = count(); }
tick-5s { exit(0); }

# dtrace -s probe_count_5sec.d
dtrace: script 'probe_count_5sec.d' matched 10 probes
CPU     ID                    FUNCTION:NAME
  0  82683                         :tick-5s

  exit                                                              1
  lwp-exit                                                          1
  signal-discard                                                    1
  create                                                            2
  exec                                                              2
  exec-success                                                      2
  signal-send                                                       2
```

- 스택 트레이싱

```
# more stack.d

syscall::read:entry /pid == $target/ {
 printf("pid=%d", pid);
 stack();
 printf("----------");
 ustack();
}
# more do.sh
cat /etc/passwd > out1
cp /etc/passwd out2

# dtrace -c './do.sh' -s stack.d
dtrace: script 'stack.d' matched 2 probes
dtrace: pid 6379 has exited
CPU     ID                    FUNCTION:NAME
  0  80483                       read:entry pid=6379
```

```
              kernel`0xffffffff80fd735b
----------
              libc.so.7`_read+0xa
              sh`xxreadtoken+0xe5
              sh`readtoken+0x35
              sh`parsecmd+0xd5
              sh`cmdloop+0xb2
              sh`main+0x268
              libc.so.7`__libc_start1+0x12a
              sh`_start+0x2d
              `0x1f5668a03008

  0  80483                      read:entry pid=6379
              kernel`0xffffffff80fd735b
----------
              libc.so.7`_read+0xa
              sh`xxreadtoken+0xe5
              sh`readtoken+0x35
              sh`parsecmd+0xd5
              sh`cmdloop+0xb2
              sh`main+0x268
              libc.so.7`__libc_start1+0x12a
              sh`_start+0x2d
              `0x1f5668a03008
```

2.3. DTrace 툴킷

- DTrace 툴킷 스크립트 활용

```
  DTrace Toolkit 패키지를 설치하여 DTrace에서 제공하는 스크립트를 이용할 수 있다.
# pkg install dtrace-toolkit
```

```
  hotkernel: 가장 많은 커널 시간을 사용하는 함수 식별
# /usr/local/share/dtrace-toolkit/hotkernel
Sampling... Hit Ctrl-C to end.
^C
FUNCTION                                               COUNT   PCNT
kernel`ohci_roothub_exec                                   1   0.0%
kernel`ata_begin_transaction                               1   0.0%
kernel`ehci_roothub_exec                                   1   0.0%
kernel`lock_delay                                          1   0.0%
kernel`softclock                                           1   0.0%
kernel`em_update_stats_counters                            2   0.0%
kernel`em_if_update_admin_status                           7   0.1%
kernel`cpu_idle                                           10   0.1%
kernel`acpi_timer_get_timecount                           16   0.1%
kernel`spinlock_exit                                     191   1.6%
kernel`cpu_idle_hlt                                    11814  98.1%
```

```
# cat /usr/local/share/dtrace-toolkit/hotkernel
[...]      perl 스크립트로 작성됨
my $dtrace = <<END;
/usr/sbin/dtrace -n '
        #pragma D option quiet          dtrace의 기본 출력을 억제한다.(quiet)
        profile:::profile-1001hz        초당 1001회 실행되는 프로파일링 프로브를 설정.
                                        CPU 사용량을 샘플링하는데 사용된다.
        /arg0/          arg0는 프로브가 발생한 명령어 주소이며 0이 아닐때만 다음 작업을 실행
        {
```

```
                \@pc[arg0] = count();       프로그램 카운터 주소를 키로 하여 집계를 수행
        }
        dtrace:::END
        {                스크립트 종료시 실행되며, printa 함수를 사용하여 집계된 결과를 출력
                printa("%a %\@d\\n", \@pc);
        }
'
END

my %Count;
my $total;
open DTRACE, "$dtrace |" or die "ERROR1: Can't run dtrace (perms?): $!\n";
print "Sampling... Hit Ctrl-C to end.\n";
while (my $line = <DTRACE>) {    printa의 출력은 파일 <DTRACE>를 통해 전달된다.
    next if $line =~ /^\s*$/;
    my ($addr, $count) = split ' ', $line;
    my ($name, $offset) = split /\+/, $addr;
    next if $name eq "0x0";
    $name =~ s/\`.*// if $mods;
    $Count{$name} += $count;
    $total += $count;    각 항목별로 percent 계산을 위해 total 카운트를 구함
}
close DTRACE;
```

```
  새로 실행되는 프로세스를 추적하기
# cd /usr/local/share/dtrace-toolkit ; ./execsnoop -h
USAGE: execsnoop [-a|-A|-ehjsvJ] [-c command]
       execsnoop               # default output
                -a             # print all data
                -A             # dump all data, space delimited
                -e             # safe output, parseable
                -s             # print start time, us
                -v             # print start time, string
                -J             # print jail ID
                -c command     # command name to snoop
  eg,
       execsnoop -v            # human readable timestamps
       execsnoop -J            # print jail ID
       execsnoop -c ls         # snoop ls commands only

# ./execsnoop -a     다른 터미털에서 ls, pwd, date 명령을 실행
TIME           STRTIME              JAIL ID      UID    PID   PPID ARGS
3184665758     2024 Nov  6 19:54:00 0              0  77148    932 -csh
3184665900     2024 Nov  6 19:54:00 0              0  77148    932 ls
3185706854     2024 Nov  6 19:54:01 0              0  77149    932 pwd
3186723602     2024 Nov  6 19:54:02 0              0  77150    932 date
^C
# more execsnoop
[...]
/usr/sbin/dtrace -n '
[...]
 syscall::execve:return     execve 시스템콜을 추적함
 /(FILTER == 0) || (OPT_cmd == 1 && COMMAND == execname)/
 {
        /* 옵션에 따라서 값을 출력한다. */
```

```
        OPT_time ? printf("%-14d ", timestamp/1000) : 1;      타임스탬프 출력 (마이크로초)
        OPT_timestr ? printf("%-20Y ", walltimestamp) : 1;    wall 시간 출력
        OPT_jailid ? printf("%-10d ", curpsinfo->pr_jailid) : 1;     jail ID를 출력
        /* print main data */
        OPT_dump ? printf("%d %d %d %d %d %s ", timestamp/1000,
            curpsinfo->pr_jailid, uid, pid, ppid, execname) :   uid, pid, ppid 를 출력
            printf("%5d %6d %6d ", uid, pid, ppid);
        OPT_safe ? printf("%S\n", curpsinfo->pr_psargs) :       pr_pargs 프로세스 인자
            printf("%s\n", curpsinfo->pr_psargs);
}
'
```

3. BPFTrace 사용하기

- 커널에 BPF 가 활성화 되어 있는지 체크하기:

```
  디렉터리가 존재하면 BPF 가 활성화 된것임
# sudo ls /sys/fs/bpf
  커널 설정에 CONFIG_BPF 를 확인
# grep CONFIG_BPF /boot/config-$(uname -r)    # CONFIG_DEBUG_INFO_BTF
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
CONFIG_BPF_JIT=y
CONFIG_BPF_JIT_ALWAYS_ON=y
CONFIG_BPF_JIT_DEFAULT_ON=y
CONFIG_BPF_UNPRIV_DEFAULT_OFF=y
# CONFIG_BPF_PRELOAD is not set
CONFIG_BPF_LSM=y
CONFIG_BPF_STREAM_PARSER=y
CONFIG_BPF_EVENTS=y
CONFIG_BPF_KPROBE_OVERRIDE=y
```

- bpftrace [14][15]는 eBPF를 활용한 고수준 추적 및 디버깅 도구이다. 패키지에는 함께 제공하는 스크립트(.bt)와 예제도 포함되어 있다.

```
# dpkg-query -L bpftrace
[...]
/usr/bin/bpftrace
/usr/bin/bpftrace-aotrt
[...]
/usr/sbin/bashreadline.bt
/usr/sbin/biolatency-kp.bt
[...]
/usr/share/doc/bpftrace/examples
/usr/share/doc/bpftrace/examples/bashreadline_example.txt
/usr/share/doc/bpftrace/examples/biolatency_example.txt
# ls /usr/sbin/*.bt
   몇가지 .bt 스크립트에 대한 설명
tcpconnect.bt    TCP 연결 시도를 추적
tcpaccept.bt     수락된 TCP 연결을 추적
bashreadline.bt  bash 커맨트를 추적
biolatency.bt    block I/O 의 지연시간 분포를 측정
opensnoop.bt     파일 open() 시스템 호출을 추적
vfscount.bt      VFS 동작을 추적
cpuwalk.bt       CPU 사용량을 샘플링하여 각 CPU 코어 활용도를 보여준다
```

3.1. 프로브 리스트

- bpftrace 튜토리얼 [16] [17]

```
# bpftrace --version
bpftrace v0.20.2

형식
 bpftrace [options] FILENAME
```

14) https://bpftrace.org/
15) https://github.com/bpftrace/bpftrace

```
 bpftrace [options] -e 'program code'
  -l 옵션은 사용가능한 프로브(probe)들의 목록을 나열한다. (root 권한이 필요함)
# sudo bpftrace -l | wc -l
115838
  특정 패턴과 일치하는 항목만 나열하기. 프로브 목록에서 맨 처음 컬럼은 프로브 타입임
# sudo bpftrace -l 'tracepoint:syscalls:sys_enter_*'
tracepoint:syscalls:sys_enter_accept
tracepoint:syscalls:sys_enter_accept4
tracepoint:syscalls:sys_enter_access
tracepoint:syscalls:sys_enter_acct
[...]
# bpftrace -l 'kprobe:*nanosleep'
kprobe:__ia32_sys_clock_nanosleep
kprobe:__ia32_sys_nanosleep
kprobe:__x64_sys_clock_nanosleep
kprobe:__x64_sys_nanosleep
kprobe:do_cpu_nanosleep
kprobe:do_nanosleep
kprobe:hrtimer_nanosleep
# bpftrace -l 'interval:*'
interval:hz:
interval:ms:
interval:s:
interval:us:

 -lv 옵션은 해당 프로브의 더 상세한 정보를 볼수 있다.
# sudo bpftrace -lv 'tracepoint:syscalls:sys_enter_open'
tracepoint:syscalls:sys_enter_open
    int __syscall_nr
    const char * filename
    int flags
    umode_t mode
```

- 제공되는 프로브종류는 다음과 같다: [18][19] ()안은 alias(축약어)임

프로브 종류	설명
kprobe (k)	커널 함수 진입점이나 반환점에 동적으로 삽입되는 프로브 커널 내부 동작을 상세히 추적할 수 있음
kretprobe (kr)	커널 동적 함수 리턴 추적
tracepoint (t)	커널 개발자들이 미리 정의해 놓은 정적 트레이스 포인트 안정적이고 잘 문서화된 인터페이스를 제공
uprobe (u)	사용자 공간 애플리케이션의 함수 진입점이나 반환점에 동적으로 삽입되는 프로브 특정 애플리케이션의 동작을 추적할 수 있음
uretprobe(ur)	사용자 레벨 동적 함수 리턴 추적
USDT	User Statically-Defined Tracing 애플리케이션 개발자가 미리 정의해 놓은 정적 트레이스 포인트
hardware (h)	CPU 성능 모니터링 카운터(PMC)와 관련된 이벤트를 추적
software (s)	소프트웨어 정의 이벤트를 추적
interval (i)	주기적으로 실행되는 프로브입니다.
profile (p)	특정 주기로 모든 CPU에서 실행되는 프로브
BEGIN/END	프로그램 시작과 종료 시 실행되는 특별한 프로브.

16) https://github.com/bpftrace/bpftrace/blob/master/docs/tutorial_one_liners.md
17) https://man.archlinux.org/man/extra/bpftrace/bpftrace.8.en

3.2. kprobe 프로브

```
  cgroup 빌트인 변수는 현재 프로세스의 cgroup ID를 반환한다. exit()로 바로 종료
# bpftrace -e 'BEGIN { printf("Current pid=%d cgroupID=%d\n", pid, cgroup); exit() }'
Attaching 1 probe...
Current pid=10887 cgroupID=53322

 vfs_write 커널프로브 추적하기 (해당 cgroup만 해당)
# bpftrace -e 'kprobe:vfs_write /cgroup==53322/ { printf("%s opened %s\n", comm, str(arg1)); }'
Attaching 1 probe...
sshd opened
sudo opened
bash opened
sudo opened
[...] ^C
  sys_read() 커널 함수의 리턴값을 수집하여 log2 히스토그램으로 출력 (해당 cgroup만)
# bpftrace -e 'kretprobe:vfs_read /cgroup==53322/ { @bytes = hist(retval); }'
Attaching 1 probe...
^C
@bytes:
[0]                10 |                                                    |
[1]               124 |@@                                                  |
[2, 4)             38 |                                                    |
[4, 8)              0 |                                                    |
[8, 16)            17 |                                                    |
[16, 32)           69 |@                                                   |
[32, 64)          326 |@@@@@@@                                             |
[64, 128)          90 |@@                                                  |
[128, 256)         65 |@                                                   |
[256, 512)         66 |@                                                   |
[512, 1K)          59 |@                                                   |
[1K, 2K)            6 |                                                    |
[2K, 4K)          243 |@@@@@                                               |
[4K, 8K)          474 |@@@@@@@@@@                                          |
[8K, 16K)        2281 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[16K, 32K)          0 |                                                    |
[32K, 64K)          0 |                                                    |
[64K, 128K)         0 |                                                    |
[128K, 256K)        1 |                                                    |
[256K, 512K)       25 |                                                    |
```

3.3. tracepoint 프로브

- tracepoint:syscalls:sys_enter_openat 예제

```
   tracepoint(t) 프로브 리스트
# bpftrace -l 't:*'

// openat.c      openat() 시스템 호출을 사용하는 예제 코드
[...]
int main(int argc, char *argv[]) {
[...]
  pid_t pid = getpid();
  printf("Current Process ID: %d\n", pid);  // PID 값을 출력하고 10초간 대기
  sleep(10);
```

18) https://bpftrace.org/docs/v0.21.x.html#_probes
19) https://www.brendangregg.com/BPF/bpftrace-cheat-sheet.html

```
  int dir_fd = AT_FDCWD; // 현재 작업 디렉토리 기준
  int fd = openat(dir_fd, argv[1], O_RDONLY); // openat() 호출
  if (fd == -1) { perror("openat"); return EXIT_FAILURE; }
[...]
$ gcc -o openat  openat.c
$ ./openat /etc/group            openat()으로 파일을 읽어서 터미널에 출력
Current Process ID: 37773
[...]
# export targetpid=37773               /pid==$targetpid/ 는 predicate로 필터기능을 함
# bpftrace -e 'tracepoint:syscalls:sys_enter_openat /pid=='"$targetpid"'/
    { printf("%s %s\n", comm, str(args.filename)); }'
                    printf()함수는 builtin 함수[20]
Attaching 1 probe...
openat /etc/group          <-- openat() 시스템호출이 될때 출력됨
^C
```

- tracepoint:ext4:ext4_allocate_blocks 예제

```
# bpftrace -lv 'tracepoint:ext4:ext4_allocate_blocks'
tracepoint:ext4:ext4_allocate_blocks
    dev_t dev
    ino_t ino
    __u64 block
    unsigned int len
    __u32 logical
    __u32 lleft
    __u32 lright
    __u64 goal
    __u64 pleft
    __u64 pright
    unsigned int flags
  ext4 파일시스템에 새로운 블럭이 할당될 때. args를 사용할 수 있음.
# bpftrace -e 'tracepoint:ext4:ext4_allocate_blocks { printf("PID: %d, Blocks Allocated: %d, In
ode: %d\n", pid, args.block, args.len); }'
Attaching 1 probe...        다른 터미널에서 실행하기: dd if=/dev/zero of=file bs=1024 count=2
PID: 38592, Blocks Allocated: 984190, Inode: 1
PID: 38593, Blocks Allocated: 10911501, Inode: 1
PID: 38594, Blocks Allocated: 984191, Inode: 1
PID: 38595, Blocks Allocated: 984192, Inode: 1
PID: 38596, Blocks Allocated: 984193, Inode: 1
PID: 38597, Blocks Allocated: 10911502, Inode: 1
^C
```

3.4. software 프로브

- software:page-faults 프로브

```
  software 프로브 목록
# bpftrace -l 'software:*'
software:alignment-faults:
software:bpf-output:
software:context-switches:
software:cpu-clock:
```

20) https://github.com/bpftrace/bpftrace/blob/master/man/adoc/bpftrace.adoc#functions

```
software:cpu-migrations:
software:cpu:
software:cs:
software:dummy:
software:emulation-faults:
software:faults:
software:major-faults:
software:minor-faults:
software:page-faults:
software:task-clock:
  페이지 폴트 이벤트를 카운트하고 각 프로세스 별로 출력하기
# bpftrace -e 'software:page-faults { @[comm] = count(); }'
Attaching 1 probe...
^C
@[ls]: 1
@[dir]: 1
@[bash]: 2
@[vim]: 18

 페이지 폴트를 카운트하고, 5초마다 출력하고, 초기화 하기
# bpftrace -e 'software:page-faults { @[comm] = count(); } interval:s:5 { print(@);clear(@); }'

 프로세스가 사용한 CPU 클럭을 카운트하기
# bpftrace -e 'software:cpu-clock { @[comm] = count(); }'
^C
@[sshd]: 3
@[swapper/0]: 695
@[busy_single]: 6775
@[swapper/1]: 7467
```

4. eBPF 작동 원리

- DTrace를 개발자인 Brendan Gregg의 USENIX LISA 2021의 발표 eBPF Internals [21]을 참고함
- bpftrace 소스 코드 받기

```
# git clone https://github.com/bpftrace/bpftrace.git
# cd bpftrace/src
```

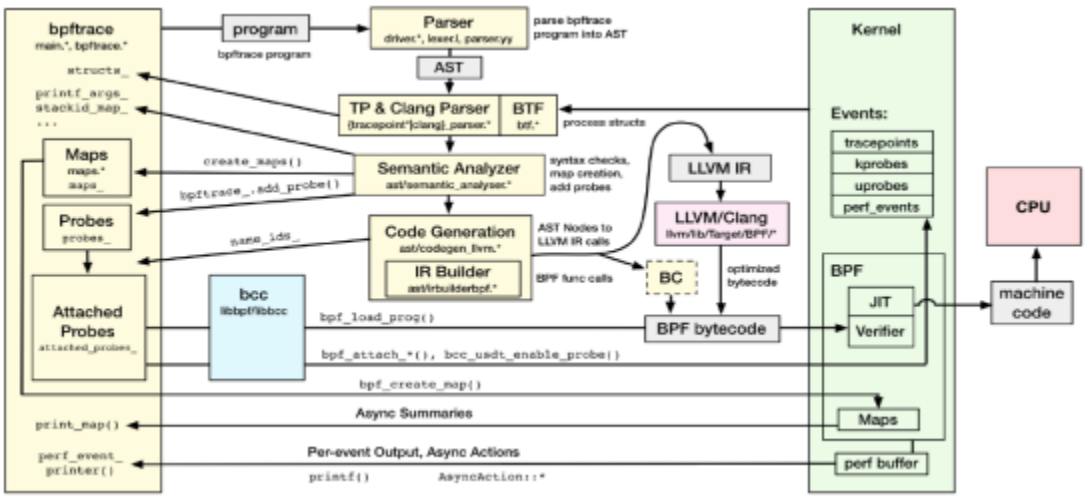
- eBPF(Extended BPF)는 BPF를 현대화한 것으로, 다음과 같은 차이가 존재함. 기존의 BPF를 "Classic BPF"라는 의미로 cBPF라고 명칭하기도 한다 :

구분	cBPF	eBPF
기능 범위	네트워크 패킷 필터링에 한정됨	성능 모니터링, 보안 등 광범위한 기능을 제공
프로그래밍 모델	2개의 32비트 레지스터만 사용 제한된 명령어 세트	10개의 64비트 레지스터 사용
실행환경	주로 네트워크 스택에서 동작	커널의 다양한 부분에 연결 가능 더 넓은 범위의 시스템 리소스에 접근 가능
성능	제한된 최적화	JIT(just-in-time) 컴파일을 통한 향상된 성능
프로그래밍 지원	어셈블리 수준의 지원	C/C++ 기반의 LLVM/Clang 지원
데이터 저장	상태 저장기능 없음	맵(maps)을 통한 상태 저장 및 사용자 공간 데이터 공유 가능

- tcpdump는 UNIX 계열에서 사용되는 패킷 분석 도구로, BSD, 솔라리스, 리눅스, 맥OSX등 대부분의 유닉스 계열 운영체제에서 동작한다. -d 옵션은 주어진 패킷 매칭 규칙을 컴파일된 코드로 출력한다.

```
# tcpdump -d host 127.0.0.1 and port 80
(000) ldh      [12]                    이더넷 헤더의 타입 필드를 로드
(001) jeq      #0x800           jt 2    jf 18     IPv4 패킷인지 확인
(002) ld       [26]                               목적지 IP 주소 로드
(003) jeq      #0x7f000001      jt 6    jf 4      0x7f=128. 0x7f000001 => 127.0.0.1
(004) ld       [30]
[...]
(013) ldh      [x + 14]
(014) jeq      #0x50            jt 17   jf 15     0x50=80. 포트번호를 확인
(015) ldh      [x + 16]
(016) jeq      #0x50            jt 17   jf 18
(017) ret      #262144              패킷을 수락함
(018) ret      #0                   패킷을 거부함
```

- BPF 내부 구조도



21) https://www.brendangregg.com/Slides/LISA2021_BPF_Internals/

4.1. BPF 시스템 호출

- 시스템 호출 `bpf()`[22]는 사용자 공간에서 eBPF 프로그램을 커널에 로드하는 등 다양한 작업을 수행한다

  형식: `#include <linux/bpf.h>`

  `int bpf(int cmd, union bpf_attr *attr, unsigned int size);`

- 커맨드(cmd)의 종류는 다음과 같다: [23]

커맨드 (일부만)	설명
BPF_MAP_CREATE	eBPF 맵을 생성. 맵은 프로그램 내에서 데이터를 저장하고 조회하는데 사용. 주로 사용자와 커널 간의 데이터를 공유하는 용도로 사용. 맵 유형은 해시맵, 배열, LRU 해시, 스택트레이스, 링버퍼 등이 있다.
BPF_MAP_LOOKUP_ELEM	맵에서 요소를 조회. 주어진 키에 대한 값을 반환
BPF_MAP_UPDATE_ELEM	맵의 요소를 업데이트하거나 삽입함
BPF_MAP_DELETE_ELEM	맵에서 요소를 삭제함
BPF_PROG_LOAD	eBPF 프로그램을 커널에 로드함. eBPF 바이트코드로 컴파일된 프로그램을 로드함.
BPF_PROG_ATTACH BPF_PROG_DETACH	eBPF 프로그램을 cgroup이나 트레이싱 이벤트(kprobe, tracepoint)와 같은 특정 시스템 이벤트에 연결하거나 분리함.
BPF_OBJ_PIN BPF_OBJ_GET	eBPF 객체를 파일 시스템에 핀하고 접근할 수 있게 함. 가상 파일 시스템(/sys/fs/bpf/)에 핀하면 다른 프로그램에서도 참조할 수 있음

- `bpf_attr` 구조체(union of struct)[24]는 `bpf()` 시스템호출을 위한 인터페이스임.

필드(일부만)	설명
map_type	맵의 타입을 지정. BPF_MAP_TYPE_HASH, BPF_MAP_TYPE_ARRAY 등
map_size	맵의 크기(요소 개수)를 지정
key_size / value_size	맵의 키와 값의 크기를 지정
map_entries	맵의 최대 항목 수를 지정
map_flags	맵 생성시 사용할 수 있는 플래그. BPF_F_RDONLY 등
prog_type	프로그램 타입. BPF_PROG_TYPE_KPROBE, BPF_PROG_TYPE_XDP 등.
insns_cnt / insns	프로그램 명령 수와 해당 명령이 저장된 메모리 주소(insns)
license	라이선스 문자열을 지정. 예 “GPL"
log_level / log_size / log_buf	BPF 프로그램의 디버그 로그 설정. level은 상세 수준. size는 로그 버퍼의 크기, buf는 로그를 담는 버퍼
prog_flags	BPF 프로그램에 대한 플래그. eg. BPF_F_SLEEPABLE (sleep 가능)
attach_type	BPF 프로그램이 어떤 이벤트나 트리거와 연결될지를 지정
target_fd	BPF 프로그램이 연결될 대상 파일 디스크립터. kprobe가 특정 함수에 연결될 때 사용
prog_id	BPF 프로그램의 ID로, 이미 로드된 프로그램을 참조할 때 사용
map_id	BPF 맵의 ID로, 이미 생성된 맵을 참조할 때 사용
attr	명령에 따라 추가적인 속성이나 데이터를 설정할 수 있는 필드

4.2. 커널 프로브 예제

- 커널에서 발생하는 `kprobe:do_nanosleep` 이벤트를 탐지. `do_nanosleep`은 커널 내부에 구현되어 있는 함수로 `kernel/time/hrtimer.c` [25]에 정의되어 있다.

22) https://man7.org/linux/man-pages/man2/bpf.2.html
23) https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h#L922
24) https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h#L1461
25) https://github.com/torvalds/linux/blob/master/kernel/time/hrtimer.c#L2023

```
# bpftrace  -e 'kprobe:do_nanosleep { printf("PID %d sleeping...\n", pid); }'
Attaching 1 probe...
PID 2168 sleeping...
PID 2168 sleeping...
PID 5264 sleeping...
PID 2168 sleeping...
^C
```

※ bpftrace에서 kprobe[26] 가 동작하는 원리는 다음과 같다:
- 커널 함수 후킹: 지정된 커널 함수의 시작 부분에 중단점(breakpoint)를 삽입함. 이를 위해서 원래 함수의 첫번째 바이트를 INT3 명령어(X86 시스템)로 교체함.
- 예외처리: CPU는 INT3 명령에 의해 kprobe 핸들러로 제어를 넘긴다.
- BPF 프로그램 실행: kprobe 핸들러는 bpftrace에서 정의한 BPF 프로그램을 실행함. 이 프로그램은 함수의 인자, 스텍 트레이스 등의 정보에 접근가능 함.
- 함수 흐름을 복원: BPF 프로그램이 종료 후, 커널은 원래 함수의 첫 번째 명령어를 실행함. (single-step 디버깅). 이후에 원래 함수로 돌아가 나머지 코드를 계속 실행한다.

4.3. LLVM AST

- bpftrace는 LLVM/Clang 기반으로 구현된다. -d(또는 -dd) 옵션은 LLVM 내부에서 이루어지는 AST 생성과, IR 코드 생성을 출력한다.

```
# bpftrace -d -e 'kprobe:do_nanosleep { printf("PID %d sleeping...\n", pid); }'
[DEBUG] libbpf: loading kernel BTF '/sys/kernel/btf/vmlinux': 0
[DEBUG] libbpf: loading kernel BTF '/sys/kernel/btf/vmlinux': 0
AST after: parser
------------------
Program
 kprobe:do_nanosleep
  call: printf :: type[none, ctx: 0]
   string: PID %d sleeping...\n :: type[none, ctx: 0]
   builtin: pid :: type[none, ctx: 0]

AST after: ConfigAnalyser
------------------
Program
 kprobe:do_nanosleep
  call: printf :: type[none, ctx: 0]
   string: PID %d sleeping...\n :: type[string[20], ctx: 0]
   builtin: pid :: type[none, ctx: 0]

AST after: Semantic
------------------
Program
 kprobe:do_nanosleep
  call: printf :: type[none, ctx: 0]
   string: PID %d sleeping...\n :: type[string[20], ctx: 0]
   builtin: pid :: type[unsigned int64, ctx: 0]

AST after: NodeCounter
------------------
```

26) https://docs.kernel.org/trace/kprobes.html

```
Program
 kprobe:do_nanosleep
  call: printf :: type[none, ctx: 0]
   string: PID %d sleeping...\n :: type[string[20], ctx: 0]
   builtin: pid :: type[unsigned int64, ctx: 0]

AST after: ResourceAnalyser
--------------------
Program
 kprobe:do_nanosleep
  call: printf :: type[none, ctx: 0]
   string: PID %d sleeping...\n :: type[string[20], ctx: 0]
   builtin: pid :: type[unsigned int64, ctx: 0]

    아래는 LLVM IR 코드
; ModuleID = 'bpftrace'
source_filename = "bpftrace"
target datalayout = "e-m:e-p:64:64-i64:64-i128:128-n32:64-S128"
target triple = "bpf-pc-linux"

%printf_t = type { i64, i64 }

; Function Attrs: nounwind
declare i64 @llvm.bpf.pseudo(i64 %0, i64 %1) #0

define noundef i64 @"kprobe:do_nanosleep"(ptr nocapture readnone %0) local_unnamed_addr section
"s_kprobe:do_nanosleep_1" !dbg !3 {
entry:
  %key = alloca i32, align 4
  %printf_args = alloca %printf_t, align 8   ; printf 를 위한 argument 공간
  call void @llvm.lifetime.start.p0(i64 -1, ptr nonnull %printf_args)
  store i64 0, ptr %printf_args, align 8
  %get_pid_tgid = tail call i64 inttoptr (i64 14 to ptr)()
  ; 14는 bpf_get_current_pid_tgid()를 의미하며 현재 프로세서 pid와  thread group ID를 얻는다.
  %1 = lshr i64 %get_pid_tgid, 32     ; lshr=logical shift right
  %2 = getelementptr inbounds %printf_t, ptr %printf_args, i64 0, i32 1
  store i64 %1, ptr %2, align 8
  %pseudo = tail call i64 @llvm.bpf.pseudo(i64 1, i64 0)
   ; 130은 bpf_ringbuf_output()을 의미하며, 데이터를 링버퍼에 기록한다
  %ringbuf_output = call i64 inttoptr (i64 130 to ptr)(i64 %pseudo, ptr nonnull %printf_args, i
64 16, i64 0)
  %ringbuf_loss = icmp slt i64 %ringbuf_output, 0
  br i1 %ringbuf_loss, label %event_loss_counter, label %counter_merge

event_loss_counter:                               ; preds = %entry
  call void @llvm.lifetime.start.p0(i64 -1, ptr nonnull %key)
  store i32 0, ptr %key, align 4
  %pseudo1 = call i64 @llvm.bpf.pseudo(i64 1, i64 1)
  %lookup_elem = call ptr inttoptr (i64 1 to ptr)(i64 %pseudo1, ptr nonnull %key)
  %map_lookup_cond.not = icmp eq ptr %lookup_elem, null
  br i1 %map_lookup_cond.not, label %lookup_merge, label %lookup_success

counter_merge:                                    ; preds = %lookup_merge, %entry
  call void @llvm.lifetime.end.p0(i64 -1, ptr nonnull %printf_args)
```

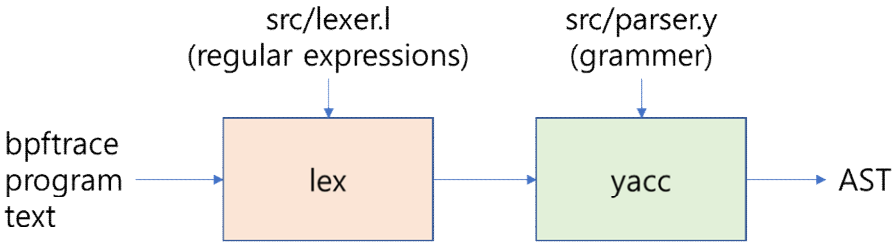
```
  ret i64 0

lookup_success:                                   ; preds = %event_loss_counter
  %3 = atomicrmw add ptr %lookup_elem, i64 1 seq_cst, align 8
  br label %lookup_merge

lookup_merge:                                     ; preds = %event_loss_counter, %lookup_succes
s
  call void @llvm.lifetime.end.p0(i64 -1, ptr nonnull %key)
  br label %counter_merge
}
[...]
```

4.4. LLVM IR 생성

- bpftrace에서 어떤 과정을 거쳐 위와 같은 IR 코드가 생성되는지 알아본다.

  코드: `kprobe:do_nanosleep { printf("PID %d sleeping...\n", pid); }`

  위 코드는 먼저 파서(parser)에 의해 처리된다. bpftrace에서 파서는 `lexer.l` [27]과 `parser.yy` [28]로 구현된다. Lex(lexical analyzer generator)는 입력 스트림을 토큰으로 분해한다. 토큰은 정규표현식을 이용하여 정의된다. `lexer.l`에 정의된 규칙에따라 C코드로 작성된 렉서를 생성한다. Yacc(yet another compiler compiler)는 구문분석기(parser)를 생성하는 도구이다.



- 렉서와 파서의 결과 다음과 같은 AST를 얻게 됨:

```
AST after: parser
------------------
Program
 kprobe:do_nanosleep
  call: printf :: type[none, ctx: 0]
   string: PID %d sleeping...\n :: type[none, ctx: 0]
   builtin: pid :: type[none, ctx: 0]
```

- 의미분석기(semantic analyzer)는 코드의 에러를 체크한다. `ast/passes/semantic_analyser.cpp` [29]

```
  pid가 아닌 pidd와 같은 정의되지 않은 토큰은 다음과 같은 오류를 발생한다:
# bpftrace -d -e 'kprobe:do_nanosleep { printf("PID %d sleeping...\n", pidd); }'
[...]
stdin:1:54-58: ERROR: Unknown identifier: 'pidd'
kprobe:do_nanosleep { printf("PID %d sleeping...\n", pidd); }
                                                     ~~~~
// src/ast/passes/semantic_analyser.cpp
[...]
```

27) https://github.com/bpftrace/bpftrace/blob/master/src/lexer.l
28) https://github.com/bpftrace/bpftrace/blob/master/src/parser.yy
29) https://github.com/bpftrace/bpftrace/blob/master/src/ast/passes/semantic_analyser.cpp

```
void SemanticAnalyser::visit(Identifier &identifier)
{
  if (bpftrace_.enums_.count(identifier.ident) != 0) {
    const auto &enum_name = std::get<1>(bpftrace_.enums_[identifier.ident]);
    identifier.type = CreateEnum(64, enum_name);
[...]
  } else {
    identifier.type = CreateNone();
    LOG(ERROR, identifier.loc, err_)
        << "Unknown identifier: '" + identifier.ident + "'";
  }
} [...]
```

- 다음 단계는 생성된 AST를 LLVM IR 표현으로 변환한다

```
// src/ast/passes/codegen_llvm.cpp
namespace bpftrace::ast {
[...]
void CodegenLLVM::visit(Builtin &builtin)
{
  if (builtin.ident == "nsecs") {
[...]
  } else if (builtin.ident == "pid") {
    expr_ = b_.CreateGetPid(builtin.loc);
  } else if (builtin.ident == "tid") {
    expr_ = b_.CreateGetTid(builtin.loc);
[...]
```

```
// src/ast/irbuilderbpf.cpp
Value *IRBuilderBPF::CreateGetPid(const location &loc)
{
  Value *pidtgid = CreateGetPidTgid(loc);
  Value *pid = CreateTrunc(CreateLShr(pidtgid, 32), getInt32Ty(), "pid");
  return pid;                BPF logical shift right instruction
}
CallInst *IRBuilderBPF::CreateGetPidTgid(const location &loc)
{
  // u64 bpf_get_current_pid_tgid(void)
  // Return: current->tgid << 32 | current->pid
  FunctionType *getpidtgid_func_type = FunctionType::get(getInt64Ty(), false);
  return CreateHelperCall(libbpf::BPF_FUNC_get_current_pid_tgid,
                          getpidtgid_func_type,
                          {},
                          "get_pid_tgid",
                          &loc);
}
```

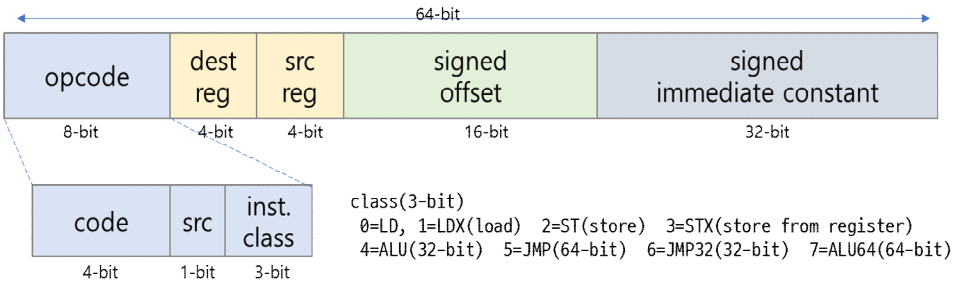
```
// src/libbpf/bpf.h
#define __BPF_FUNC_MAPPER(FN)                   \
        FN(unspec),                             \  <--- #0
        FN(map_lookup_elem),                    \
        FN(map_update_elem),                    \
        FN(map_delete_elem),                    \
[...]
        FN(clone_redirect),                     \
        FN(get_current_pid_tgid),               \ <--- #14
```

```
        FN(get_current_uid_gid),                \
[...]
  위 AST는 다음과 같은 LLVM IR로 변환된다.
# bpftrace -d -e 'kprobe:do_nanosleep { printf("PID %d sleeping...\n", pid); }'
[...]
  %key = alloca i32, align 4
  %printf_args = alloca %printf_t, align 8
     notnull은 printf_args가 널이 아님을 의미. -1은 object의 길이를 알수 없다는 의미.
   @llvm.lifetime.start.p0 는 llvm intrinsic 함수로 optimizer에게 stack 할당에 관한 정보를 제공
  call void @llvm.lifetime.start.p0(i64 -1, ptr nonnull %printf_args)
  store i64 0, ptr %printf_args, align 8
  %get_pid_tgid = tail call i64 inttoptr (i64 14 to ptr)()  #14 get_current_pid_tgid
  %1 = lshr i64 %get_pid_tgid, 32              logical shift right
  getelementptr 명령은 포인터 계산을 위한 명령임.
  inbounds 키워드는 포인터 계산시 오버플로우를 피하도록 지시하여 최적화 단계에서 힌트를 제공함.
  printf_t는 구조체 타입으로 %printf_t = type { i64, i64 } 로 정의되어 있음.
  i64 0는 첫번째 인덱스임. i32 1은 두번째 인덱스임
  %2 = getelementptr inbounds %printf_t, ptr %printf_args, i64 0, i32 1
  store i64 %1, ptr %2, align 8
  %pseudo = tail call i64 @llvm.bpf.pseudo(i64 1, i64 0)
  %ringbuf_output = call i64 inttoptr (i64 130 to ptr)(i64 %pseudo, ptr nonnull %printf_args, i
64 16, i64 0)
  %ringbuf_loss = icmp slt i64 %ringbuf_output, 0
  br i1 %ringbuf_loss, label %event_loss_counter, label %counter_merge
```

4.5. eBPS 바이트코드 구조

- eBPF 명령어 바이트코드 포맷[30]

 * eBPF 명령어는 64/128비트 길이로, opcode 8비트, dest_reg 4비트(0-10), src_reg 4비트(0-10), offset 16비트, imm 32비트(추가64비트)로 구성됨. 일부 명령은 imm 값으로 추가 64비트를 더하여 총 128비트 길이가 됨
 * eBPF를 10개의 범용 레지스터와 1개의 읽기전용 프레임 포인터 레지스터를 가진다. R0: 함수 호출시 리턴값을 저장하는 용도. R1~R5(5개): 함수 호출시 인자를 전달. R6~R9(4개): 함수 호출시 유지되어야 하는 레지스터를 보관하는 용도. R10: 읽기전용의 스텍프레임 포인터임. 레지스터의 폭은 64비트이다.
 * 8비트의 opcode는 3비트(LSB)의 instruction class 값(총 8개의 클래스가 존재), 1비트의 src값, 그리고, 4비트의 code(MSB)로 구성됨.



※ 64-bit immediate 명령 참고[31]

30) https://docs.kernel.org/6.5/bpf/instruction-set.html
31) https://docs.kernel.org/6.5/bpf/instruction-set.html#bit-immediate-instructions

- 커널내부에서 BPF를 처리하기 위한 bpf_insn 구조체는 include/uapi/linux/bpf.h[32]에 정의됨

```
// include/uapi/linux/bpf.h

struct bpf_insn {
        __u8    code;           /* opcode */
        __u8    dst_reg:4;      /* dest register */
        __u8    src_reg:4;      /* source register */
        __s16   off;            /* signed offset */
        __s32   imm;            /* signed immediate constant */
};
```

- C언어로 작성된 eBPF 프로그램을 BPF 객체 파일로 컴파일하고 역어셈블하는 방법

```
$ cat my_bpf_program.c
int filter_func(int arg)
{
  if (arg > 0) return 1;
  return 0;
}

$ clang -target bpf  -c my_bpf_program.c  -o my_bpf_objfile.o
$ file my_bpf_objfile.o
my_bpf_objfile.o: ELF 64-bit LSB relocatable, eBPF, version 1 (SYSV), not stripped

$ readelf -x .text  my_bpf_objfile.o
Hex dump of section '.text':
  0x00000000 631af8ff 00000000 61a2f8ff 00000000 c.......a.......
  0x00000010 67020000 20000000 c7020000 20000000 g... ....... ...
  0x00000020 b7010000 01000000 6d210400 00000000 ........m!......
  0x00000030 05000000 00000000 b7010000 01000000 ................
  0x00000040 631afcff 00000000 05000300 00000000 c...............
  0x00000050 b7010000 00000000 631afcff 00000000 ........c.......
  0x00000060 05000000 00000000 61a0fcff 00000000 ........a.......
  0x00000070 95000000 00000000                   ........

$ llvm-objdump --disassemble   my_bpf_objfile.o

my_bpf_objfile.o:                                       file format elf64-bpf
Disassembly of section .text:
0000000000000000 <filter_func>:
       0:  63 1a f8 ff 00 00 00 00 *(u32 *)(r10 - 0x8) = r1
                                          r1의 값을 스택포인터 r10에서 오프셋 -8 위치에 저장
       1:  61 a2 f8 ff 00 00 00 00 r2 = *(u32 *)(r10 - 0x8)
                                            r10 오프셋 -8위치의 메모리에서 읽어서 r2에 저장
       2:  67 02 00 00 20 00 00 00 r2 <<= 0x20     r2의 값을 32비트 왼쪽으로 이동
       3:  c7 02 00 00 20 00 00 00 r2 s>>= 0x20    r2의 값을 32비트 오른쪽으로 이동
       4:  b7 01 00 00 01 00 00 00 r1 = 0x1       r1이 r2보다 크면(s>는 부호가 있는 비교) 분기
       5:  6d 21 04 00 00 00 00 00 if r1 s> r2 goto +0x4 <LBB0_2>
       6:  05 00 00 00 00 00 00 00 goto +0x0 <LBB0_1>

0000000000000038 <LBB0_1>:
       7:  b7 01 00 00 01 00 00 00 r1 = 0x1
```

32) https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h

```
       8:  63 1a fc ff 00 00 00 00 *(u32 *)(r10 - 0x4) = r1    1을 리턴하기 위해 메모리에 저장
       9:  05 00 03 00 00 00 00 00 goto +0x3 <LBB0_3>

0000000000000050 <LBB0_2>:
      10:  b7 01 00 00 00 00 00 00 r1 = 0x0
      11:  63 1a fc ff 00 00 00 00 *(u32 *)(r10 - 0x4) = r1    0을 리턴하기 위해 메모리에 저장
      12:  05 00 00 00 00 00 00 00 goto +0x0 <LBB0_3>

0000000000000068 <LBB0_3>:
      13:  61 a0 fc ff 00 00 00 00 r0 = *(u32 *)(r10 - 0x4)    r0 에 값을 넣고 리턴한다
      14:  95 00 00 00 00 00 00 00 exit
```

```
  -O2 옵션을 이용하면 다른 결과를 얻는다.
$ clang -target bpf  -O2  -c my_bpf_program.c  -o my_bpf_objfile.o
$ llvm-objdump --disassemble   my_bpf_objfile.o
my_bpf_objfile.o:                                       file format elf64-bpf
Disassembly of section .text:
0000000000000000 <filter_func>:
       0:  67 01 00 00 20 00 00 00 r1 <<= 0x20
       1:  c7 01 00 00 20 00 00 00 r1 s>>= 0x20
       2:  b7 00 00 00 01 00 00 00 r0 = 0x1
       3:  65 01 01 00 00 00 00 00 if r1 s> 0x0 goto +0x1 <LBB0_2>  조건을 만족하면 r0=1을 리턴
       4:  b7 00 00 00 00 00 00 00 r0 = 0x0   만족하지 않으면 r0=0을 리턴
0000000000000028 <LBB0_2>:
       5:  95 00 00 00 00 00 00 00 exit
```

4.6. eBPF 바이트코드로 변환

- 앞의 nanosleep 예제에서 생성된 IR에 의해 최종적으로 BPF 바이트코드로 변환된다.
  llvm/lib/Target/BPF/BPFInstrInfo.td [33]

```
// llvm/lib/Target/BPF/BPFInstrInfo.td
class CALL<string OpcodeStr>
    : TYPE_ALU_JMP<BPF_CALL.Value, BPF_K.Value,
                   (outs),
                   (ins calltarget:$BrDst),
                   !strconcat(OpcodeStr, " $BrDst"),
                   []> {
  bits<32> BrDst;

  let Inst{31-0} = BrDst;
  let BPFClass = BPF_JMP;
}
```

- 시스템 호출 추적 도구인 strace를 이용하여 확인해 본다.
  bpf() 시스템 호출에서 cmd=BPF_PROG_LOAD가 호출됨. eBPF 명령어는 2개임을 알 수 있다.

```
  -f 옵션은 자식 프로세스의 호출도 함께 추적함. -e 옵션은 특정 시스템 호출만 추적함.
# sudo strace -fe bpf \
   bpftrace  -e 'kprobe:do_nanosleep { printf("PID %d sleeping...\n", pid); }'
[...]
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_KPROBE, insn_cnt=2, insns=0x7fff4cf948f0, license="
GPL", log_level=0, log_size=0, log_buf=NULL, kern_version=KERNEL_VERSION(0, 0, 0), prog_flags=
```

33) https://github.com/llvm/llvm-project/blob/main/llvm/lib/Target/BPF/BPFInstrInfo.td#L649

```
0, prog_name="ksys_read", prog_ifindex=0, expected_attach_type=BPF_TRACE_KPROBE_MULTI, prog_btf
_fd=0, func_info_rec_size=0, func_info=NULL, func_info_cnt=0, line_info_rec_size=0, line_info=N
ULL, line_info_cnt=0, attach_btf_id=0, attach_prog_fd=0, fd_array=NULL}, 144) = 6
```

5. BCC

- BCC(BPF Compiler Collection) 소스코드[34]와 bpfcc-tools/examples 을 참고
- BCC는 커널 계측을 위한 BPF 프로그램을 C언어로 작성하고, Python, Lua 등의 고수준 언어로 사용자 공간의 인터페이스를 제공함. eBPF 프로그램의 컴파일, 로딩, 실행을 위한 툴체인을 제공함. JIT컴파일된 프로그램의 동적 로딩/언로딩을 지원한다.

```
  파이썬 패키지 설치 경로
/usr/lib/python3/dist-packages/bcc
   예제 코드 경로
/usr/share/doc/bpfcc-tools/examples
   bcc 깃허브 주소
git clone https://github.com/iovisor/bcc
```

- 관련 패키지 및 설치 파일

```
# dpkg-query -L bpfcc-tools
/usr/sbin
/usr/sbin/argdist-bpfcc
/usr/sbin/bashreadline-bpfcc
/usr/sbin/bindsnoop-bpfcc
/usr/sbin/biolatency-bpfcc
[...]
/usr/share/doc/bpfcc-tools/examples
/usr/share/doc/bpfcc-tools/examples/doc
/usr/share/doc/bpfcc-tools/examples/doc/argdist_example.txt
/usr/share/doc/bpfcc-tools/examples/doc/bashreadline_example.txt
# dpkg-query -L python3-bpfcc
[...]
/usr/lib/python3/dist-packages/bcc
/usr/lib/python3/dist-packages/bcc/__init__.py
/usr/lib/python3/dist-packages/bcc/containers.py
/usr/lib/python3/dist-packages/bcc/disassembler.py
/usr/lib/python3/dist-packages/bcc/libbcc.py
/usr/lib/python3/dist-packages/bcc/perf.py
/usr/lib/python3/dist-packages/bcc/syscall.py
/usr/lib/python3/dist-packages/bcc/table.py
/usr/lib/python3/dist-packages/bcc/tcp.py
/usr/lib/python3/dist-packages/bcc/usdt.py
/usr/lib/python3/dist-packages/bcc/utils.py
/usr/lib/python3/dist-packages/bcc/version.py
$ python3 -c "import bcc ; print(bcc.__version__)"
0.29.1
```

5.1. BCC 작동원리

- sys_openat() 커널 함수를 추적하는 프로그램 예시

```
# cat hello.py
#!/usr/bin/python3
from bcc import BPF   <-- bcc 모듈을 로딩함
prog = """
int kprobe__sys_openat(void *ctx) {
```

34) https://github.com/iovisor/bcc

```
  bpf_trace_printk("Hello, World!\\n");  bpf_trace_printk 핼퍼 함수 kernel/trace/bpf_trace.c 35)
  return 0;
}
"""
b = BPF(text=prog, debug=True)  <-- 위 코드를 BPF 바이트코드로 변환하고 커널로 전달
b.trace_print()      <-- kernel debug trace pipe 에 연결하고 읽어서 출력함
```

```
   strace 를 이용하여 실행하기
# strace -fe bpf  ./hello.py
[...]   bpf() 시스템 호출로 바이트 코드가 로딩됨
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_SOCKET_FILTER, insn_cnt=2, insns=0x7fffe4477b30, li
cense="GPL", log_level=0, log_size=0, log_buf=NULL, kern_version=KERNEL_VERSION(0, 0, 0), prog_
flags=0, prog_name="", prog_ifindex=0, expected_attach_type=BPF_CGROUP_INET_INGRESS, prog_btf_f
d=0, func_info_rec_size=0, func_info=NULL, func_info_cnt=0, line_info_rec_size=0, line_info=NUL
L, line_info_cnt=0, attach_btf_id=0, attach_prog_fd=0}, 116) = 3
bpf(BPF_BTF_LOAD, {btf="\237\353\1\0\30\0\0\0\0\0\0\0\324\0\0\0\324\0\0\0\17\1\0\0\0\0\0\0\0
\0\2"..., btf_log_buf=NULL, btf_size=507, btf_log_size=0, btf_log_level=0}, 32) = 3
[...]   openat() 시스템호출이 발생될 때 마다 메시지가 출력됨
b'            bash-40229   [001] ....1 303483.790741: bpf_trace_printk: Hello, World!'
b''
b'            bash-40229   [001] ....1 303487.079707: bpf_trace_printk: Hello, World!'
b''
[...]
```

- 위 예제 코드를 분석하기

```
   bcc 파이썬 모듈에 포함된 BCC 객체
; /usr/lib/python3/dist-packages/bcc/__init__.py  36)
[...]
  같은 패키지에 있는 libbcc.py 로 부터 객체를 가져옴
from .libbcc import lib, bcc_symbol, bcc_symbol_option, bcc_stacktrace_build_id, _SYM_CB_TYPE
[...]
class BPF(object):
[...]         BPF 클래스 호출로 프로그램 텍스트(text)가 전달됨. 이것을 BPF 바이트 코드로 변환
    def __init__(self, src_file=b"", hdr_file=b"", text=None, debug=0,
            cflags=[], usdt_contexts=[], allow_rlimit=True, device=None,
            attach_usdt_ignore_pid=False):
[...]        text는 lib에서 제공하는 메소드를 통해서 바이트코드로 변환됨
        self.module = lib.bpf_module_create_c_from_string(text,
                                                           self.debug,
                                                           cflags_array, len(cflags_array),
                                                           allow_rlimit, device)
[...]   함수명이 kprobe__, kretprobe__, tracepoint__ 등일때
        # If any "kprobe__" or "tracepoint__" or "raw_tracepoint__"
        # prefixed functions were defined,
        # they will be loaded and attached here.
        self._trace_autoload()
[...]
    def _trace_autoload(self):
        for i in range(0, lib.bpf_num_functions(self.module)):
            func_name = lib.bpf_function_name(self.module, i)
            if func_name.startswith(b"kprobe__"):   # 커널 프로브를 로딩하기
                fn = self.load_func(func_name, BPF.KPROBE)
```

35) https://github.com/torvalds/linux/blob/master/kernel/trace/bpf_trace.c#L398

```
                self.attach_kprobe(     #  "kprobe__" 이후의 이름 = "sys_openat"
                    event=self.fix_syscall_fnname(func_name[8:]),
                    fn_name=fn.name)
[...]
; /usr/lib/python3/dist-packages/bcc/libbcc.py  37)
  libbcc의 lib 객체는 C언어로 작성된 shared 라이브러리임.
   ctypes은 C언어로 작성된 함수와 라이브러리를 사용할 수 있게 해준다
import ctypes as ct
lib = ct.CDLL("libbcc.so.0", use_errno=True)
[...]
  libbcc.so.0 라이브러리는 아래 경로에 존재함
$ readlink /usr/lib/x86_64-linux-gnu/libbcc.so.0
libbcc.so.0.29.1
  bpf_module_create_c_from_string 함수 정의는 다음과 같다. 38)
; bcc/src/cc/bcc_common.c
void * bpf_module_create_c_from_string(const char *text, unsigned flags, const char *cflags[],
                                       int ncflags, bool allow_rlimit, const char *dev_name) {
    BPFModule에서 주어진 코드를 바이트코드로 처리한다
  auto mod = new ebpf::BPFModule(flags, nullptr, true, "", allow_rlimit, dev_name);
  if (mod->load_string(text, cflags, ncflags) != 0) {
    delete mod;
    return nullptr;
  }
  return mod;
}
  bpf_module.cc 39) 은 다음과 같다
[...]
namespace ebpf {
[...]
BPFModule::BPFModule(unsigned flags, TableStorage *ts, bool rw_engine_enabled,
                     const std::string &maps_ns, bool allow_rlimit,
                     const char *dev_name)
    : flags_(flags),
      rw_engine_enabled_(rw_engine_enabled && bpf_module_rw_engine_enabled()),
      used_b_loader_(false),
      allow_rlimit_(allow_rlimit),
      ctx_(new LLVMContext),
      id_(std::to_string((uintptr_t)this)),
      maps_ns_(maps_ns),
      ts_(ts), btf_(nullptr) {
  ifindex_ = dev_name ? if_nametoindex(dev_name) : 0;
  initialize_rw_engine();    <-- bpf_module_rw_engine.cc 참고

  LLVMInitializeBPFTarget();  <-- LLVM에서 제공하는 초기화 루틴. BPF 타겟을 초기화함
  LLVMInitializeBPFTargetMC();
  LLVMInitializeBPFTargetInfo();
  LLVMInitializeBPFAsmPrinter();
#if LLVM_VERSION_MAJOR >= 6
  LLVMInitializeBPFAsmParser();
  if (flags & DEBUG_SOURCE)
    LLVMInitializeBPFDisassembler();
#endif
  LLVMLinkInMCJIT(); /* call empty function to force linking of MCJIT */
  if (!ts_) {
```

```
    local_ts_ = createSharedTableStorage();
    ts_ = &*local_ts_;
  }
  prog_func_info_ = ebpf::make_unique<ProgFuncInfo>();
}
[...]
// load a C text string
int BPFModule::load_string(const string &text, const char *cflags[], int ncflags) {
[...]
```

```
; /usr/lib/python3/dist-packages/bcc/__init__.py
[...]    eBPF 커널 프로브가 로딩되면 파이프를 통해 출력을 읽어옴
    def trace_print(self, fmt=None):
        """trace_print(self, fmt=None)

        Read from the kernel debug trace pipe and print on stdout.
        If fmt is specified, apply as a format string to the output. See
        trace_fields for the members of the tuple
        example: trace_print(fmt="pid {1}, msg = {5}")
        """
        while True:
            if fmt:   # fmt 포매팅 형식으로 출력할 경우
                fields = self.trace_fields(nonblocking=False)
                if not fields: continue
                line = fmt.format(*fields)
            else:    # readline으로 파이프에서 읽기
                line = self.trace_readline(nonblocking=False)
            print(line)
            sys.stdout.flush()
[...]      # readline으로 파이프에서 읽기
    def trace_readline(self, nonblocking=False):
        trace = self.trace_open(nonblocking)  # 파이프가 열려있지 않으면 open해줌
     line = None
        try:
            line = trace.readline(1024).rstrip()   # 1024 바이트씩 읽음
[...]
    def trace_open(self, nonblocking=False):
        """trace_open(nonblocking=False)
        Open the trace_pipe if not already open
        """
        if not self.tracefile:  # 파이프의 위치 TRACEFS 참조
            self.tracefile = open("%s/trace_pipe" % TRACEFS, "rb")
            if nonblocking:
                fd = self.tracefile.fileno()
                fl = fcntl.fcntl(fd, fcntl.F_GETFL)
                fcntl.fcntl(fd, fcntl.F_SETFL, fl | os.O_NONBLOCK)
        return self.tracefile
[...]      /sys/kernel/debug/tracing/trace_pipe 와 연결된다.
DEBUGFS = "/sys/kernel/debug"
TRACEFS = os.path.join(DEBUGFS, "tracing")   # /sys/kernel/debug/tracing
```

36) https://github.com/iovisor/bcc/blob/master/src/python/bcc/__init__.py#L26
37) https://github.com/iovisor/bcc/blob/master/src/python/bcc/libbcc.py#L15
38) https://github.com/iovisor/bcc/blob/master/src/cc/bcc_common.cc#L30C1-L32C62
39) https://github.com/iovisor/bcc/blob/master/src/cc/bpf_module.cc

- 시스템호출이 실행될 때 원하는 함수를 실행하는 예제

```
# hello2.py
#!/usr/bin/python3
from bcc import BPF
prog = """
int hello(void *ctx) {
  bpf_trace_printk("Hello, World!\\n");
  return 0;
}
"""
b = BPF(text=prog, debug=True)    # openat 시스템콜이 발생될때 hello() 함수를 실행
b.attach_kprobe(event=b.get_syscall_fnname("openat"), fn_name="hello")
print("PID MESSAGE")
try:
  b.trace_print(fmt="{1} {5}") # format string의 각 필드는 [0]:Timestamp [1]Process ID [2] CPU
ID [3] Flags [4] Timestamp in ns [5] Message 이다.
except KeyboardInterrupt:
  exit()
```

```
# ./hello2.py
PID MESSAGE
43893 b'Hello, World!'    PID와 hello() 함수로 부터의 메시지가 출력된다
647 b'Hello, World!'
647 b'Hello, World!'
647 b'Hello, World!'
647 b'Hello, World!'
^C
```

```
; /usr/lib/python3/dist-packages/bcc/__init__.py
    def attach_kprobe(self, event=b"", event_off=0, fn_name=b"", event_re=b""):
        event = _assert_is_bytes(event)
        fn_name = _assert_is_bytes(fn_name)
        event_re = _assert_is_bytes(event_re)

        # allow the caller to glob multiple functions together
        if event_re:
            matches = BPF.get_kprobe_functions(event_re)
                    event_re는 regular expression으로 매치되는 event를 모두 연결
[...]
        self._check_probe_quota(1)
            BPF.KPROBE는 프로브타입으로 uapi/linux/bpf.h[40] 에 정의되어 있음
        fn = self.load_func(fn_name, BPF.KPROBE)  # 해당 fn_name("hello")를 커널에 로드함
        ev_name = b"p_" + event.replace(b"+", b"_").replace(b".", b"_")
        fd = lib.bpf_attach_kprobe(fn.fd, 0, ev_name, event, event_off, 0)
        if fd < 0:
            raise Exception([...])
        self._add_kprobe_fd(ev_name, fn_name, fd)
        return self
[...]
    def load_func(self, func_name, prog_type, device = None, attach_type = -1):
[...]       libbcc 에서 제공하는 기능을 통해 해당 함수를 로드함
        fd = lib.bcc_func_load(self.module, prog_type, func_name,
                lib.bpf_function_start(self.module, func_name),
                lib.bpf_function_size(self.module, func_name),
```

```
                lib.bpf_module_license(self.module),
                lib.bpf_module_kern_version(self.module),
                log_level, None, 0, device, attach_type)
```

```
// bpf_module.cc [41]
int BPFModule::bcc_func_load(int prog_type, const char *name,     name: 프로그램 이름
                const struct bpf_insn *insns, int prog_len, insns: BPF instruction array의 주소
                const char *license, unsigned kern_version,   kern_version: BPF를 컴파일한 커널
                int log_level, char *log_buf, unsigned log_buf_size,
                const char *dev_name, unsigned flags, int expected_attach_type) {
[...]
  ret = bcc_prog_load_xattr((enum bpf_prog_type)prog_type, name, license, insns, &opts, prog_le
n, log_buf, log_buf_size, allow_rlimit_);
```

```
// libbpf.c [42]
int bcc_prog_load_xattr(enum bpf_prog_type prog_type, const char *prog_name,
                        const char *license, const struct bpf_insn *insns,
                        struct bpf_prog_load_opts *opts, int prog_len,
                        char *log_buf, unsigned log_buf_size, bool allow_rlimit)
[...]
  ret = libbpf_bpf_prog_load(prog_type, new_prog_name, license, insns, insns_cnt, opts, opts_lo
g_buf, opts_log_buf_size);
```

```
// libbpf/src/bpf.c [43]
  최종적으로 libbpf 의 bpf() 시스템 호출 cmd=BPF_PROG_LOAD 로 처리됨. x86_64 아키텍처에서 bpf()
시스템 호출 번호는 321 이며[44]. 같은 파일에 __NR_bpf 에 정의됨
# elif defined(__x86_64__)
#  define __NR_bpf 321
[...]
int sys_bpf_prog_load(union bpf_attr *attr, unsigned int size, int attempts)
{
        int fd;
        do {
                fd = sys_bpf_fd(BPF_PROG_LOAD, attr, size);
        } while (fd < 0 && errno == EAGAIN && --attempts > 0);
        return fd;
}
```

- trace_pipe 를 통해 읽어오기

```
# more hello3.py       hello2.py 를 수정. 프로그램 로딩까지만 하고 대기한다
#!/usr/bin/python3
from bcc import BPF
prog = """
int hello(void *ctx) {
  bpf_trace_printk("Hello, world!\\n");
  return 0;
}
"""
```

40) https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h#L1024
41) https://github.com/iovisor/bcc/blob/master/src/cc/bpf_module.cc#L997
42) https://github.com/iovisor/bcc/blob/master/src/cc/libbpf.c#L779
43) https://github.com/libbpf/libbpf/blob/42065ea6627ff6e1ab4c65e51042a70fbf30ff7c/src/bpf.c#L72
44) https://github.com/torvalds/linux/blob/v4.17/arch/x86/entry/syscalls/syscall_64.tbl#L332

```
b = BPF(text=prog, debug=False)
b.attach_kprobe(event=b.get_syscall_fnname("openat"), fn_name="hello")
print("CTRL-C to stop")
try:       # trace_print 를 하지 않고, 대기만 함
  while True: pass
except KeyboardInterrupt:
  exit()
# ./hello3.py
CTRL-C to stop
  다른 터미널에서 /sys/kernel/debug/tracing 으로 이동
# cd /sys/kernel/debug/tracing/ ; ls
README
available_events
available_filter_functions
available_filter_functions_addrs
[...]

   사용가능한 트레이서의 목록. function: 개별 함수 호출을 추적, function_graph: 함수 호출 그래프를 생성, irqsoff: 인터럽트가 비활성화된 기간을 측정, preemptoff: 선점이 비활성화된 기간을 측정, wakeup: 태스크를 깨우기 지연을 측정 등이 있다.
# cat  /sys/kernel/debug/tracing/available_tracers
timerlat osnoise hwlat blk mmiotrace function_graph wakeup_dl wakeup_rt wakeup function nop

  현재 활성화된 트레이서를 표시함
# cat /sys/kernel/debug/tracing/current_tracer
nop

   trace_pipe 를 통해 트레이스 정보를 읽어옴. current_tracer 가 nop로 설정되어 있어도 trace_printk() 호출을 차단하지는 않음
# cat /sys/kernel/debug/tracing/trace_pipe
      irqbalance-558     [000] ....1 94041.388345: bpf_trace_printk: Hello, world!
      irqbalance-558     [000] ....1 94041.388356: bpf_trace_printk: Hello, world!
      irqbalance-558     [000] ....1 94041.388366: bpf_trace_printk: Hello, world!
^C

  current_tracer 를 function 으로 변경. 수많은 메시지가 출력되나, grep로 검색할 수 있음
# echo function > /sys/kernel/debug/tracing/current_tracer
# cat /sys/kernel/debug/tracing/trace_pipe ¦ grep Hello
      irqbalance-558     [000] ....1 94101.387920: bpf_trace_printk: Hello, world!
              ls-12457   [001] ....1 94107.432569: bpf_trace_printk: Hello, world!
              ls-12457   [001] ....1 94107.432648: bpf_trace_printk: Hello, world!
[...]

  커널의 트레이싱 기능을 모두 끄려면 tracing_on 에 0을 쓰면 된다. 이후에는 아무런 트레이싱 정보가 출력되지 않음.
# echo 0 > /sys/kernel/debug/tracing/tracing_on
# cat /sys/kernel/debug/tracing/tracing_on
0
   다시 활성화 하기:  echo 1 > /sys/kernel/debug/tracing/tracing_on
```

5.2. killsnoop 예제

```
   bpfcc-tools 패키지에 포함된 bpfcc 파일들
# ls /usr/sbin/*-bpfcc
```

```
/usr/sbin/argdist-bpfcc         /usr/sbin/javaflow-bpfcc        /usr/sbin/runqlat-bpfcc
/usr/sbin/bashreadline-bpfcc    /usr/sbin/javagc-bpfcc          /usr/sbin/runqlen-bpfcc
/usr/sbin/bindsnoop-bpfcc       /usr/sbin/javaobjnew-bpfcc      /usr/sbin/runqslower-bpfcc
/usr/sbin/biolatency-bpfcc      /usr/sbin/javastat-bpfcc        /usr/sbin/shmsnoop-bpfcc
/usr/sbin/biolatpcts-bpfcc      /usr/sbin/javathreads-bpfcc     /usr/sbin/slabratetop-bpfcc
/usr/sbin/biopattern-bpfcc      /usr/sbin/killsnoop-bpfcc       /usr/sbin/sofdsnoop-bpfcc
[...]
# more  /usr/sbin/killsnoop-bpfcc
[...]
# arguments
examples = """examples:
    ./killsnoop           # trace all kill() signals
    ./killsnoop -x        # only show failed kills
    ./killsnoop -p 181    # only trace PID 181
    ./killsnoop -T 189    # only trace target PID 189
    ./killsnoop -s 9      # only trace signal 9
"""
[...]
bpf_text = """
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>

struct val_t { // 해시맵의 데이터 형식
   u32 pid;
   int sig;
   int tpid;
   char comm[TASK_COMM_LEN];
};

struct data_t {  // 사용자 공간으로 전송할 데이터 형식
   u32 pid;
   int tpid;
   int sig;
   int ret;
   char comm[TASK_COMM_LEN];
};

 # infotmp 라는 이름으로 해시 맵을 생성 키는 u32, value는 val_t
BPF_HASH(infotmp, u32, struct val_t);
BPF_PERF_OUTPUT(events);   # 이벤트 버퍼를 생성

 # kill 시스템 호출을 트리거함
int syscall__kill(struct pt_regs *ctx, int tpid, int sig) {
    u64 pid_tgid = bpf_get_current_pid_tgid();
    u32 pid = pid_tgid >> 32;   # process ID(pid), thread ID(tid)를 조회
    u32 tid = (u32)pid_tgid;

    TPID_FILTER
    PID_FILTER
    SIGNAL_FILTER

    struct val_t val = {.pid = pid};  bpf_get_current_comm 은 프로세서의 이름을 수집함
    if (bpf_get_current_comm(&val.comm, sizeof(val.comm)) == 0) {
        val.tpid = tpid;
        val.sig = sig;
```

```
        infotmp.update(&tid, &val);
    }

    return 0;
};

int do_ret_sys_kill(struct pt_regs *ctx) {    kill 시스템 호출로 부터의 반환을 트리거함
    struct data_t data = {};  // 사용자 공간으로 전송할 데이터
    struct val_t *valp;
    u64 pid_tgid = bpf_get_current_pid_tgid();
    u32 pid = pid_tgid >> 32;   # process ID(pid), thread ID(tid)를 조회
    u32 tid = (u32)pid_tgid;
    valp = infotmp.lookup(&tid); # infotmp 해시맵을 참조한다. valp 를 설정
    if (valp == 0) {  return 0;  } // missed entry

      해시맵의 command name을 data.comm 으로 복사
    bpf_probe_read_kernel(&data.comm, sizeof(data.comm), valp->comm);
    data.pid = pid;
    data.tpid = valp->tpid;
    data.ret = PT_REGS_RC(ctx); // kill() 호출의 return value를 얻음
    data.sig = valp->sig;

    events.perf_submit(ctx, &data, sizeof(data));  // data 내용을 사용자 공간으로 전송
    infotmp.delete(&tid);  // 해시맵을 정리함
    return 0;
}
"""

if args.tpid:
    bpf_text = bpf_text.replace('TPID_FILTER',
        'if (tpid != %s) { return 0; }' % args.tpid)
else:
    bpf_text = bpf_text.replace('TPID_FILTER', '')

if args.pid:
    bpf_text = bpf_text.replace('PID_FILTER',
        'if (pid != %s) { return 0; }' % args.pid)
else:
    bpf_text = bpf_text.replace('PID_FILTER', '')

if args.signal:
    bpf_text = bpf_text.replace('SIGNAL_FILTER',
        'if (sig != %s) { return 0; }' % args.signal)
else:
    bpf_text = bpf_text.replace('SIGNAL_FILTER', '')

if debug or args.ebpf:
    print(bpf_text)
    if args.ebpf:
        exit()

# initialize BPF
b = BPF(text=bpf_text)
kill_fnname = b.get_syscall_fnname("kill")
```

```
b.attach_kprobe(event=kill_fnname, fn_name="syscall__kill")
b.attach_kretprobe(event=kill_fnname, fn_name="do_ret_sys_kill")

# detect the length of PID column
pid_bytes = 6
[...]
# header
print("%-9s %-*s %-16s %-4s %-*s %s" % (
    "TIME", pid_bytes, "PID", "COMM", "SIG", pid_bytes, "TPID", "RESULT"))

# 이벤트 처리 함수
def print_event(cpu, data, size):
    event = b["events"].event(data)  # 버퍼에서 event 정보를 가져온다.

    if (args.failed and (event.ret >= 0)):
        return
            시간  PID  COMM   SIG TPID  RESULT
    printb(b"%-9s %-*d %-16s %-4d %-*d %d" % (strftime("%H:%M:%S").encode('ascii'),
        pid_bytes, event.pid, event.comm, event.sig, pid_bytes, event.tpid, event.ret))

# loop with callback to print_event
b["events"].open_perf_buffer(print_event)
while 1:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
        exit()
```

```
  프로그램을 실행
# killsnoop-bpfcc
TIME      PID      COMM             SIG  TPID     RESULT
07:49:54  35513    bash             15   35600    0

  TPID는 target 프로세스의 pid임. PID는 bash 쉘의 pid 임. TPID는 kill을 당한 프로세스의 pid임
  SIG 15는 SIGTERM 으로 kill 명령으로 종료되었음을 의미
  다른 터미널에서 sleep 프로세스를 kill 함
$ sleep 1000 &
[1] 35600
$ kill 35600
$
[1]+  Terminated      sleep 1000
```

5.3. TCPv4 모니터링 예제

- TCP 접속을 모니터링 하는 예제

```
; tcpv4connect.py
#!/usr/bin/python3
[...]
bpf_text = """
#include <uapi/linux/ptrace.h>
#include <net/sock.h>
```

```
#include <bcc/proto.h>
  해쉬 맵을 선언한다. 형식: name, key_type, value_type
BPF_HASH(currsock, u32, struct sock *);

   tcp_v4_connect 함수가 호출될때 트리거됨
int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk) {
  u32 pid = bpf_get_current_pid_tgid();
  // stash the sock ptr for lookup on return
  currsock.update(&pid, &sk);  // socket(sk)를 해시맵에 저장함
  return 0;
};
         TCP 접속이 연결(connect) 호출이 종료될때 (kretprobe) 트리거됨
int kretprobe__tcp_v4_connect(struct pt_regs *ctx) {
  int ret = PT_REGS_RC(ctx);    리턴값을 얻어오기 위한 매크로. pt_regs 구조체에는 커널 함수의
진입과 종료시에 CPU레지스터의 상태가 저장되어 있다.
  u32 pid = bpf_get_current_pid_tgid();

  struct sock **skpp;
  skpp = currsock.lookup(&pid);  해쉬맵을 참조함
  if (skpp == 0) {
    return 0;  // missed entry
  }
  if (ret != 0) {
    // failed to send SYNC packet, may not have populated
    // socket __sk_common.{skc_rcv_saddr, ...}
    currsock.delete(&pid);
    return 0;
  }

  // pull in details
  struct sock *skp = *skpp;
  u32 saddr = skp->__sk_common.skc_rcv_saddr; // source address
  u32 daddr = skp->__sk_common.skc_daddr;  // destination address
  u16 dport = skp->__sk_common.skc_dport; // destination port

  // output 형식 (msg, saddr_hs, daddr_hs, dport_s)
  bpf_trace_printk("trace_tcp4connect %x %x %d\\n", saddr, daddr, ntohs(dport));
  currsock.delete(&pid);
  return 0;
}
"""
# initialize BPF
b = BPF(text=bpf_text, debug=False)

# header
print("%-6s %-12s %-16s %-16s %-4s" % ("PID", "COMM", "SADDR", "DADDR", "DPORT"))

def inet_ntoa(addr):    IP 주소를 dotted-decimal 형식으로 변환
  dq = b''
  for i in range(0, 4):
    dq = dq + str(addr & 0xff).encode()
    if (i != 3):
      dq = dq + b'.'
    addr = addr >> 8
```

```
  return dq

# filter and format output
while 1:
  # Read messages from kernel pipe
  try:
      (task, pid, cpu, flags, ts, msg) = b.trace_fields()  # BCC 에서 제공하는 함수
      (_tag, saddr_hs, daddr_hs, dport_s) = msg.split(b" ") # 메시지를 분리함
  except ValueError:
      # Ignore messages from other tracers
      continue
  except KeyboardInterrupt:
      exit()

  # Ignore messages from other tracers
  if _tag.decode() != "trace_tcp4connect":
      continue

  printb(b"%-6d %-12.12s %-16s %-16s %-4s" % (pid, task,
      inet_ntoa(int(saddr_hs, 16)),
      inet_ntoa(int(daddr_hs, 16)),
      dport_s))
```

```
$ sudo ./tcpv4connect.py
PID    COMM         SADDR            DADDR            DPORT
1595   wget         172.31.4.209     142.XXX.XXX.XXX  443          <-- wget https://google.com
[...]
```

6. BPFTool

- bpftool [45]은 리눅스의 eBPF 프로그램과 맵을 관리하고 상호작용하기 위한 커맨드 임.

```
# dpkg-query -L  linux-tools-common
[...]
/usr/sbin
/usr/sbin/bpftool
[...]
/usr/share/man/man8
/usr/share/man/man8/bpftool-btf.8.gz
/usr/share/man/man8/bpftool-cgroup.8.gz
/usr/share/man/man8/bpftool-feature.8.gz
/usr/share/man/man8/bpftool-gen.8.gz
/usr/share/man/man8/bpftool-iter.8.gz
/usr/share/man/man8/bpftool-link.8.gz
/usr/share/man/man8/bpftool-map.8.gz
/usr/share/man/man8/bpftool-net.8.gz
/usr/share/man/man8/bpftool-perf.8.gz
/usr/share/man/man8/bpftool-prog.8.gz
/usr/share/man/man8/bpftool-struct_ops.8.gz
/usr/share/man/man8/bpftool.8.gz
[...]
# bpftool  version
bpftool v7.4.0
using libbpf v1.4
features:
```

- 위에서 작성한 hello.py 를 실행하고 bpftool 로 확인하기:

```
# ./hello.py
647 b'Hello, World!'
647 b'Hello, World!'
[...]
$ sudo bpftool prog show
[...]         프로그램 조회하기
906: kprobe  name sys_openat  tag c6a80fd10c5ff759  gpl
        loaded_at 2024-11-08T03:53:25+0000  uid 0
        xlated 224B  jited 134B  memlock 4096B
        btf_id 215
  id <program_id> 옵션은 특정 프로그램만 조회. --json 으로 json 형식으로 출력
$ sudo bpftool prog show  id 906 --json   # 또는 --pretty
{"id":906,"type":"kprobe","name":"sys_openat","tag":"c6a80fd10c5ff759","gpl_compatible":true,"l
oaded_at":1731038005,"uid":0,"orphaned":false,"bytes_xlated":224,"jited":true,"bytes_jited":13
4,"bytes_memlock":4096,"btf_id":215}

  prog dump 명령으로 프로그램을 덤프할 수 있다. xlated(Translated)는 eBPF 바이트 코드의 중간 표
현을 의미함. 커널 verifir 가 검증 및 최적화한 후의 명령어임.    jited(Just-In-Time compiled)는
실제 CPU 아키텍처에 맞는 네이티브 기계어로 변환한 것. opcodes 옵션을 추가하면 바이트코드가 함께
표시된다.
$ sudo bpftool prog  dump xlated name sys_openat  opcodes
int kprobe__sys_openat(void * ctx):
; int kprobe__sys_openat(void *ctx) {
   0: (b7) r1 = 2593
```

45) https://github.com/libbpf/bpftool

```
       b7 01 00 00 21 0a 00 00      b7은 BPF_MOV 명령. 01은 R1, 0x0a21은 imm 값
; ({ char _fmt[] = "Hello, World!\n"; bpf_trace_printk_(_fmt, sizeof(_fmt)); });
   1: (6b) *(u16 *)(r10 -4) = r1
       6b 1a fc ff 00 00 00 00
   2: (b7) r1 = 1684828783
       b7 01 00 00 6f 72 6c 64        b7은 load imm 명령, 6f 72 6c 64는 "orld"
   3: (63) *(u32 *)(r10 -8) = r1
       63 1a f8 ff 00 00 00 00
   4: (18) r1 = 0x57202c6f6c6c6548   18은 lddw명령, 01은 R1 레지스터 48 65 6c 6c 는 "Hell" 00 0
0 00 00 은 패딩을 의미 6f 2c 20 57 은 "o, W"
       18 01 00 00 48 65 6c 6c 00 00 00 00 6f 2c 20 57
   6: (7b) *(u64 *)(r10 -16) = r1
       7b 1a f0 ff 00 00 00 00
   7: (b7) r1 = 0
       b7 01 00 00 00 00 00 00
   8: (73) *(u8 *)(r10 -2) = r1
       73 1a fe ff 00 00 00 00
   9: (bf) r1 = r10
       bf a1 00 00 00 00 00 00
;
  10: (07) r1 += -16
       07 01 00 00 f0 ff ff ff
; ({ char _fmt[] = "Hello, World!\n"; bpf_trace_printk_(_fmt, sizeof(_fmt)); });
  11: (b7) r2 = 15
       b7 02 00 00 0f 00 00 00
  12: (85) call bpf_trace_printk#-112560
       85 00 00 00 50 48 fe ff
; return 0;
  13: (b7) r0 = 0
       b7 00 00 00 00 00 00 00
  14: (95) exit
       95 00 00 00 00 00 00 00
```

```
  BTF(BPF Type Format) 확인하기. 위에서 btf_id:215 로 확인되었음.
$ sudo bpftool btf
1: name [vmlinux]  size 6110058B   <---- 리눅스 커널의 BTF 정보
2: name <anon>  size 1162B  prog_ids 2
[...]
215: name <anon>  size 595B  prog_ids 906

$ sudo bpftool btf show id 215
215: name <anon>  size 595B  prog_ids 906
  BTF 내용을 덤프하기
$ sudo bpftool btf dump id 215
[1] PTR '(anon)' type_id=0           익명의 포인터 타입
[2] FUNC_PROTO '(anon)' ret_type_id=3 vlen=1   함수 프로토타입으로 반환타입은 int(type_id=3)
        'ctx' type_id=1             ctx 라는 하나의 매개변수(type_id=1)를 가짐
[3] INT 'int' size=4 bits_offset=0 nr_bits=32 encoding=SIGNED     부호있는 4바이트 정수형
[4] FUNC 'kprobe__sys_openat' type_id=2 linkage=static         kprobe__sys_openat 정적 함수
[5] INT 'unsigned int' size=4 bits_offset=0 nr_bits=32 encoding=(none)
[6] INT '(anon)' size=4 bits_offset=0 nr_bits=32 encoding=(none)
[7] INT 'char' size=1 bits_offset=0 nr_bits=8 encoding=SIGNED
[8] ARRAY '(anon)' type_id=7 index_type_id=9 nr_elems=4     4개의 요소를 가진 char 배열
[9] INT '__ARRAY_SIZE_TYPE__' size=4 bits_offset=0 nr_bits=32 encoding=(none)
```

```
[10] INT '(anon)' size=4 bits_offset=0 nr_bits=32 encoding=(none)
[11] PTR '(anon)' type_id=0
[12] PTR '(anon)' type_id=0
[13] PTR '(anon)' type_id=0
[14] PTR '(anon)' type_id=0
```

※ BPF 프로그램은 사용자가 제공한 코드로, 커널에 직접 주입되어 커널 컨텍스트에서 실행된다. 이는 커널 메모리 공간에서 작동하며, 모든 내부 커널 상태에 접근할 수 있는 강력한 기능을 가진다. 하지만 이러한 강력한 기능은 동시에 이식성 문제를 야기한다. 커널의 구조와 데이터 형식은 지속적으로 변화하는 반면, BPF 프로그램은 주변 커널 환경의 메모리 구조를 직접 제어할 수 없기 때문임. 예를 들어, 커널의 버전이 바뀌면서 구조체 내의 필드 순서가 바뀌거나 새로운 내부 구조체로 이동할 수 있으며, 때로는 특정 타입이 조건부 컴파일로 제거될 수도 있다. 이러한 문제를 해결하기 위해 BPF CO-RE(Compile Once, Run Everywhere) 매커니즘이 도입되는데, 이 메커니즘은 BPF 프로그램을 이식 가능한 방식으로 작성할 수 있게 해주며, 다양한 구성 요소들 간의 협력을 통해 작동한다.

* BTF 유형정보를 통해 커널과 BPF 프로그램 유형 및 코드에 대한
* 컴파일러(Clang)는 BPF C코드가 의도를 표현하고 재배치 정보를 기록할 수 있는 수단을 제공해야함
* BPF 로더(libbpf)는 커널과 BPF 프로그램의 BTF를 연결하여 컴파일된 BPF 코드를 대상 호스트의 특정 커널에 맞게 조정한다
* 커널은 BPF CO-RE에 완전히 독립적으로 동작함

```
  리눅스 커널의 BTF 정보를 C 형식으로 출력하기
# bpftool btf dump file /sys/kernel/btf/vmlinux format c  >  vmlinux.h
```

7. Eunomia BPF

- eunomia-bpf [46]는 eBPF 프로그램의 개발, 배포, 운영을 단순화 하는 것을 목표로 하는 프로젝트임. eBPF 개발과정을 단순화하고, eBPF 프로그램을 JSON 객체나 WebAssembly 모듈로 패키징하여 재배포 기능을 지원. 컴파일을 위한 ecc 컴파일러 툴체인을 제공. Rust 언어로 작성되었음.

```
  ecli 설치 (다운로드)
# wget https://aka.pw/bpf-ecli -O ecli && chmod +x ./ecli
# ./ecli -h
ecli subcommands, including run, push, pull

Usage: ecli [COMMAND_LINE]... [COMMAND]
Commands:
  run     run ebpf program
  client  Client operations
  push    Operations about pushing image to registry
  pull    Operations about pulling image from registry
  help    Print this message or the help of the given subcommand(s)

Arguments:
  [COMMAND_LINE]...  Not preferred. Only for compatibility to older versions. Command line to r
un. The executable could either be a local path or URL or `-` (read from stdin). The following
arguments will be passed to the program

Options:
  -h, --help  Print help
```

```
  ecc 컴파일러 다운로드
# wget https://github.com/eunomia-bpf/eunomia-bpf/releases/latest/download/ecc
# chmod +x ./ecc
# ./ecc -h
eunomia-bpf compiler

Usage: ecc [OPTIONS] <SOURCE_PATH> [EXPORT_EVENT_HEADER]
Arguments:
  <SOURCE_PATH>          path of the bpf.c file to compile
  [EXPORT_EVENT_HEADER]  path of the bpf.h header for defining event struct [default: ]
```

```
//  minimal.bpf.c
/* SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause) */
#define BPF_NO_GLOBAL_DATA
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

typedef unsigned int u32;
typedef int pid_t;
const pid_t pid_filter = 0;

char LICENSE[] SEC("license") = "Dual BSD/GPL";

SEC("tp/syscalls/sys_enter_write")
int handle_tp(void *ctx)
{
```

46) https://github.com/eunomia-bpf/bpf-developer-tutorial/blob/main/src/1-helloworld/README.md

```
 pid_t pid = bpf_get_current_pid_tgid() >> 32;
 if (pid_filter && pid != pid_filter)
  return 0;
 bpf_printk("BPF triggered sys_enter_write from PID %d.\n", pid);
 return 0;
}
# ./ecc  minimal.bpf.c
INFO [ecc_rs::bpf_compiler] Compiling bpf object...
INFO [ecc_rs::bpf_compiler] Generating package json..
INFO [ecc_rs::bpf_compiler] Packing ebpf object and config into package.json...
# file minimal.bpf.o
minimal.bpf.o: ELF 64-bit LSB relocatable, eBPF, version 1 (SYSV), not stripped
# cat package.json  ¦ jq '.'
{
  "bpf_object": "eJytVU9PE0E [..................] /9X/wbKs/7dw==",
  "bpf_object_size": 2280,
  "meta": {
    "bpf_skel": {
      "data_sections": [
        {
          "name": ".rodata",
          "variables": [
            {
              "name": "pid_filter",
              "type": "pid_t"
            }
          ]
        }
      ],
      "maps": [
[...]
      "obj_name": "minimal_bpf",
      "progs": [
        {
          "attach": "tp/syscalls/sys_enter_write",
          "link": true,
          "name": "handle_tp"
        }
      ]
    },
    "eunomia_version": "0.3.4"
  }
}
# ./ecli run package.json
INFO [faerie::elf] strtab: 0x254 symtab 0x290 relocs 0x2d8 sh_offset 0x2d8
INFO [bpf_loader_lib::skeleton::preload::section_loader] User didn't specify custom value for v
ariable pid_filter, use the default one in ELF
INFO [bpf_loader_lib::skeleton::poller] Running ebpf program...

# cat /sys/kernel/debug/tracing/trace_pipe ¦ grep "BPF triggered sys_enter_write"
           grep-13017   [001] ....1 99005.070098: bpf_trace_printk: BPF triggered sys_enter_write from PID 13017.
           grep-13017   [001] ....1 99005.070109: bpf_trace_printk: BPF triggered sys_enter_write from PID 13017.
           grep-13017   [001] ....1 99005.070117: bpf_trace_printk: BPF triggered sys_enter_write from PID 13017.
[...]
```

8. LibBPF

- libbpf는 eBPF 프로그램을 관리하고 로드하는 데 사용되는 라이브러리로 주요 특징은 다음과 같다: 컴파일된 BPF 객체 파일을 리눅스 커널에 로드함. 사용자 영역 프로그램이 eBPF 프로그램과 상호 작용할 수 있도록 API를 제공함. BPF 애플리케이션의 라이프사이클(open, load, attach, teardown) 관리, BPF 맵 관리. libbpf-bootstrap 은 BPF 프로그램을 작성하기 위한 예제 코드를 제공한다.

```
   libbpf-bootstrap을 다운받기. submodule로 libbpf 가 포함되어 있음
$ git clone --recurse-submodules  https://github.com/libbpf/libbpf-bootstrap
$ cd libbpf-bootstrap     examples/c 폴더에 예제가 존재함
$ export BS=`pwd`   현재 경로를 기준으로 하기 위해 변수를 설정

  libbpf 를 빌드하기. libbpf.a 를 생성
$ cd libbpf-bootstrap/libbpf/src ; make
$ ar t libbpf.a | tr '\n'  ' '
bpf.o btf.o libbpf.o libbpf_errno.o netlink.o nlattr.o str_error.o libbpf_probes.o bpf_prog_lin
fo.o btf_dump.o hashmap.o ringbuf.o strset.o linker.o gen_loader.o relo_core.o usdt.o zip.o el
f.o features.o

  헤더파일을 include/bpf 로 복사해 준다.
$ mkdir -p $BS/include/bpf
$ cp bpf.h libbpf.h btf.h libbpf_common.h libbpf_legacy.h \
          bpf_helpers.h bpf_helper_defs.h bpf_tracing.h \
          bpf_endian.h bpf_core_read.h skel_internal.h libbpf_version.h \
          usdt.bpf.h     $BS/include/bpf
```

8.1. minimal_legacy 예제

```
// minimal_legacy.bpf.c         eBPF 커널 코드
/* SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause) */
#define BPF_NO_GLOBAL_DATA
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

typedef unsigned int u32;  타입 선언
typedef int pid_t;

char LICENSE[] SEC("license") = "Dual BSD/GPL";

// 요소가 1개 있는 배열(array) 타입의 맵을 생성. 맵을 사용하여 커널 공간의 eBPF 프로그램과 사용
자 공간 프로그램 사이의 데이터 교환이 가능하다.
struct {
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __uint(max_entries, 1);   엔트리의 개수는 1개
  __type(key, u32);         키는 unsigned int(u32)임
  __type(value, pid_t);     값은 PID 를 저장한다.
} my_pid_map  SEC(".maps");

SEC("tp/syscalls/sys_enter_write")    write 시스템 호출시 이 Tracepoint 핸들러가 사용됨
int handle_tp(void *ctx)
{
  u32 index = 0;
```

```
  pid_t pid = bpf_get_current_pid_tgid() >> 32;     write syscall을 요청한 프로세서의 pid
  pid_t *my_pid = bpf_map_lookup_elem(&my_pid_map, &index);   map 에서 0번째 항목을 조회함
  if (!my_pid || *my_pid != pid) return 1;      map에서 조회한 특정 pid가 아니면 무시함

  bpf_printk("BPF triggered from PID %d.\n", pid);   메시지를 출력함
  return 0;
}
```

위 커널 코드에는 map에서 데이터를 읽어오는 기능만 있을뿐, map에 데이터를 write하는 것은 사용자 공간 프로그램에서 수행한다.

```
//  minimal_legacy.c       사용자 공간 프로그램
/* SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause) */
#include <stdio.h>
#include <unistd.h>
#include <sys/resource.h>
#include <bpf/libbpf.h>
#include "minimal_legacy.skel.h"  <-- 이 헤더파일에 컴파일된 eBPF 커널 코드가 포함됨

static int libbpf_print_fn(enum libbpf_print_level level, const char *format, va_list args) {
  return vfprintf(stderr, format, args);
}

int main(int argc, char **argv) {
  struct minimal_legacy_bpf *skel;
  int err;
  pid_t pid;
  unsigned index = 0;

  /* Set up libbpf errors and debug info callback */
  libbpf_set_print(libbpf_print_fn);

  /* Load and verify BPF application */
  skel = minimal_legacy_bpf__open_and_load();
  if (!skel) {
    fprintf(stderr, "Failed to open and load BPF skeleton\n");
    return 1;
  }
  pid = getpid();      현재 프로세스의 pid 값을 map 에 등록한다.
  err = bpf_map__update_elem(skel->maps.my_pid_map, &index, sizeof(index), &pid,
          sizeof(pid_t), BPF_ANY);
  if (err < 0) {
    fprintf(stderr, "Error updating map with pid: %s\n", strerror(err));
    goto cleanup;
  }
  // 커널 코드를 로드하고, tracepoint 를 연결한다
  err = minimal_legacy_bpf__attach(skel);
  if (err) {
    fprintf(stderr, "Failed to attach BPF skeleton\n");
    goto cleanup;
  }
  //대기상태에서 trace_pipe 를 통해 출력을 확인할 수 있다
  printf("Successfully started! Please run `sudo cat /sys/kernel/debug/tracing/trace_pipe` "
         "to see output of the BPF programs.\n");
  for (;;) {
    /* trigger our BPF program */
```

```
    fprintf(stderr, ".");
    sleep(1);
  }
cleanup:
  minimal_legacy_bpf__destroy(skel);
  return -err;
}
  clang -target bpf 를 이용하여 BPF 커널 코드를 컴파일
$ clang -g -O2 -target bpf   -D__TARGET_ARCH_x86                  \
     -I$BS/include   -idirafter /usr/include/x86_64-linux-gnu \
     -c minimal_legacy.bpf.c    -o minimal_legacy.bpf.o

  bpftool gen skeleton 명령을 이용하여 헤더 파일 스켈레톤을 생성
$ $BS/bpftool/src/bpftool gen skeleton minimal_legacy.bpf.o > minimal_legacy.skel.h

  사용자 공간 프로그램을 컴파일 하고 링크함
$ cc -g -Wall -I$BS/include    -c minimal_legacy.c -o minimal_legacy.o
$ cc -g -Wall minimal_legacy.o  $BS/libbpf/src/libbpf.a   -lelf -lz -o minimal_legacy

  minimal_legacy.skel.h 의 내용은 다음과 같다. (일부)

struct minimal_legacy_bpf {
        struct bpf_object_skeleton *skeleton;   // 오브젝트파일의 내용
        struct bpf_object *obj;  // bpf_object 구조체는 컴파일된 BPF 프로그램의 ELF 객체. libbp
f 라이브러리 내부 구현에 정의되어 의도적으로 불투명(opaque)하게 유지됨. libbpf.h에는 전방선언(f
orward declaration)만 제공함
        struct {
                struct bpf_map *my_pid_map;
                struct bpf_map *rodata_str1_1;
        } maps;
        struct {
                struct bpf_program *handle_tp;
        } progs;
        struct {
                struct bpf_link *handle_tp;
        } links;
};
[...]
  워크플로우는 다음과 같다: 스켈레톤 생성 -> open -> load -> attach -> destroy
static void
minimal_legacy_bpf__destroy(struct minimal_legacy_bpf *obj)
[...]
static inline int
minimal_legacy_bpf__create_skeleton(struct minimal_legacy_bpf *obj);
[...]
static inline struct minimal_legacy_bpf *
minimal_legacy_bpf__open_and_load(void)
[...]
static inline int
minimal_legacy_bpf__attach(struct minimal_legacy_bpf *obj)
[...]
static inline void
minimal_legacy_bpf__detach(struct minimal_legacy_bpf *obj)
```

```
  실행하고 trace_pipe 를 통해 모니터링
$ sudo ./minimal_legacy
libbpf: loading object 'minimal_legacy_bpf' from buffer
[...]
Successfully started! Please run `sudo cat /sys/kernel/debug/tracing/trace_pipe` to see output
of the BPF programs.
.......
$ sudo cat /sys/kernel/debug/tracing/trace_pipe
[...]
  minimal_legacy-27932   [000] ....1 75812.797498: bpf_trace_printk: BPF triggered from PID 27932.

  minimal_legacy-27932   [000] ....1 75813.797669: bpf_trace_printk: BPF triggered from PID 27932.
^C
```

8.2. minimal 예제

```
// minimal.bpf.c
// SPDX-License-Identifier: GPL-2.0 OR BSD-3-Clause
/* Copyright (c) 2020 Facebook */
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

char LICENSE[] SEC("license") = "Dual BSD/GPL";

int my_pid = 0;  // 여기서는 map을 사용하지 않고, 글로벌 변수로 my_pid 를 선언

SEC("tp/syscalls/sys_enter_write")
int handle_tp(void *ctx)
{
  int pid = bpf_get_current_pid_tgid() >> 32;
  if (pid != my_pid) return 0;
  bpf_printk("BPF triggered from PID %d.\n", pid);
  return 0;
}
```

```
//  minimal.c
// SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause)
/* Copyright (c) 2020 Facebook */
#include <stdio.h>
#include <unistd.h>
#include <sys/resource.h>
#include <bpf/libbpf.h>
#include "minimal.skel.h"

static int libbpf_print_fn(enum libbpf_print_level level, const char *format, va_list args)
{
  return vfprintf(stderr, format, args);
}

int main(int argc, char **argv)
{
  struct minimal_bpf *skel;
  int err;
```

```
  /* Set up libbpf errors and debug info callback */
  libbpf_set_print(libbpf_print_fn);

  /* Open BPF application */
  skel = minimal_bpf__open();
  if (!skel) {
    fprintf(stderr, "Failed to open BPF skeleton\n");
    return 1;
  }

  // 현재 프로세스의 pid 값을 BPF 맵에 쓰는 방식이 아니라,
  //  skel->bss 객체에 넣고, 로딩하는 방식. 전역 변수를 초기화 해 주는 역할을 한다.
  /* ensure BPF program only handles write() syscalls from our process */
  skel->bss->my_pid = getpid();
  /* Load & verify BPF programs */
  err = minimal_bpf__load(skel);
[...]

  /* Attach tracepoint handler */
  err = minimal_bpf__attach(skel);
[...]
```

```
  clang -target bpf 를 이용하여 BPF 커널 코드를 컴파일
$ clang -g -O2 -target bpf   -D__TARGET_ARCH_x86                  \
      -I$BS/include   -idirafter /usr/include/x86_64-linux-gnu \
      -c minimal.bpf.c    -o minimal.bpf.o

  bpftool gen skeleton 명령을 이용하여 헤더 파일 스켈레톤을 생성
$ $BS/bpftool/src/bpftool gen skeleton minimal.bpf.o > minimal.skel.h

  사용자 공간 프로그램을 컴파일 하고 링크함
$ cc -g -Wall -I$BS/include    -c minimal.c -o minimal.o
$ cc -g -Wall minimal.o  $BS/libbpf/src/libbpf.a   -lelf -lz -o minimal
```

```
  실행 방법은 minimal_legacy의 경우와 같다.
$ sudo ./minimal
$ sudo cat /sys/kernel/debug/tracing/trace_pipe
```

```
minimal.skel.h 의 내용
struct minimal_bpf {
        struct bpf_object_skeleton *skeleton;
        struct bpf_object *obj;
        struct {
                struct bpf_map *bss;
                struct bpf_map *rodata;
        } maps;
        struct {
                struct bpf_program *handle_tp;
        } progs;
        struct {
                struct bpf_link *handle_tp;
        } links;
        struct minimal_bpf__bss {       초기화되지 않은 전역 변수를 포함하는 섹션
                int my_pid;               (위의 minimal_legacy 예제와 다른 부분)
```

```
        } *bss;
};
[...]

static inline int
minimal_bpf__create_skeleton(struct minimal_bpf *obj)
[...]
        s->maps[0].name = "minimal_.bss";
        s->maps[0].map = &obj->maps.bss;
        s->maps[0].mmaped = (void **)&obj->bss;  BSS 맵은 메모리 매핑된 포인터를 가진다. mmaped 필드는 BPF맵의 내용을 사용자 공간 메모리에 직접 매핑하는 데 사용된다. 메모리 매핑을 통해 사용자 공간 프로그램은 시스템 콜 없이 BPF 맵의 데이터를 읽고 쓸수 있게됨.
```

8.3. minimal_ns 예제

```
//  minimal_ns.bpf.c
// SPDX-License-Identifier: GPL-2.0 OR BSD-3-Clause
/* Copyright (c) 2023 Hosein Bakhtiari */
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <linux/sched.h>

char LICENSE[] SEC("license") = "Dual BSD/GPL";

int my_pid = 0;
unsigned long long dev;
unsigned long long ino;

SEC("tp/syscalls/sys_enter_write")
int handle_tp(void *ctx)
{  // bpf_get_ns_current_pid_tgid()는 eBPF 프로그램에서 현재 프로세스의 PID와 TGID를 특정 네임스페이스 관점에서 가져오는 헬퍼 함수임. 컨테이너나 네임스페이스 내부에서의 pid를 알고 싶을때 유용함
  struct bpf_pidns_info ns;
  bpf_get_ns_current_pid_tgid(dev, ino, &ns, sizeof(ns));
  if (ns.pid != my_pid) return 0;
  bpf_printk("BPF triggered from PID %d.\n", ns.pid);
  return 0;
}
```

```
// minimal_ns.c
// SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause)
/* Copyright (c) 2023 Hosein Bakhtiari */
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <bpf/libbpf.h>
#include "minimal_ns.skel.h"

static int libbpf_print_fn(enum libbpf_print_level level, const char *format, va_list args) {
  return vfprintf(stderr, format, args);
}
```

```
int main(int argc, char **argv)
{
  struct minimal_ns_bpf *skel;
  int err;
  struct stat sb;

  /* Set up libbpf errors and debug info callback */
  libbpf_set_print(libbpf_print_fn);

  /* Open BPF application */
  skel = minimal_ns_bpf__open();
  if (!skel) {
    fprintf(stderr, "Failed to open BPF skeleton\n");
    return 1;
  }

  /* ensure BPF program only handles write() syscalls from our process */
  if (stat("/proc/self/ns/pid", &sb) == -1) {
    fprintf(stderr, "Failed to acquire namespace information");
    return 1;
  }
  skel->bss->dev = sb.st_dev;
  skel->bss->ino = sb.st_ino;
  skel->bss->my_pid = getpid();

[...]
  clang -target bpf 를 이용하여 BPF 커널 코드를 컴파일
$ clang -g -O2 -target bpf   -D__TARGET_ARCH_x86                  \
     -I$BS/include   -idirafter /usr/include/x86_64-linux-gnu \
     -c minimal_ns.bpf.c    -o minimal_ns.bpf.o

  bpftool gen skeleton 명령을 이용하여 헤더 파일 스켈레톤을 생성
$ $BS/bpftool/src/bpftool gen skeleton minimal_ns.bpf.o > minimal_ns.skel.h

  사용자 공간 프로그램을 컴파일 하고 링크함
$ cc -g -Wall -I$BS/include    -c minimal_ns.c -o minimal_ns.o
$ cc -g -Wall minimal_ns.o  $BS/libbpf/src/libbpf.a   -lelf -lz -o minimal_ns

  네임스페이스 없이 실행할 경우. pid 값은 쉘의 네임스페이스내부에 존재
$ sudo ./minimal_ns
[...]
$ sudo cat /sys/kernel/debug/tracing/trace_pipe
  minimal_ns-33231   [000] ....1 138382.142045: bpf_trace_printk: BPF triggered from PID 33231.
^C
  unshare 명령으로 새로운 pid 네임스페이스에서 실행할 경우. pid 값은 1이 된다.
$ sudo unshare --pid --fork ./minimal_ns
[...]
$ sudo cat /sys/kernel/debug/tracing/trace_pipe
  minimal_ns-33342   [001] ....1 138706.812611: bpf_trace_printk: BPF triggered from PID 1.
^C
```

8.4. bootstrap 예제

```
// bootstrap.h
/* SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause) */
/* Copyright (c) 2020 Facebook */
#ifndef __BOOTSTRAP_H
#define __BOOTSTRAP_H

#define TASK_COMM_LEN   16
#define MAX_FILENAME_LEN 127

struct event {
  int pid;
  int ppid;
  unsigned exit_code;
  unsigned long long duration_ns;
  char comm[TASK_COMM_LEN];
  char filename[MAX_FILENAME_LEN];
  bool exit_event;
};

#endif /* __BOOTSTRAP_H */
```

```
  리눅스 커널의 BTF 정보를 C 형식으로 출력하기
$ bpftool btf dump file /sys/kernel/btf/vmlinux format c  >  vmlinux.h
```

```
// bootstrap.bpf.c
// SPDX-License-Identifier: GPL-2.0 OR BSD-3-Clause
/* Copyright (c) 2020 Facebook */
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>
#include "bootstrap.h"

char LICENSE[] SEC("license") = "Dual BSD/GPL";

struct {
  __uint(type, BPF_MAP_TYPE_HASH);
  __uint(max_entries, 8192);
  __type(key, pid_t);
  __type(value, u64);
} exec_start SEC(".maps");  // 시작 시간을 기록하는 맵

struct {
  __uint(type, BPF_MAP_TYPE_RINGBUF);
  __uint(max_entries, 256 * 1024);
} rb SEC(".maps");    // 링 버퍼를 위한 맵

const volatile unsigned long long min_duration_ns = 0;

SEC("tp/sched/sched_process_exec")    // 프로세스의 exec() 시스템 호출을 트리거
int handle_exec(struct trace_event_raw_sched_process_exec *ctx) {
  struct task_struct  *task;  // vmlinux.h 에 정의됨
```

```
  unsigned fname_off;
  struct event *e; // 이벤트 객체
  pid_t pid;
  u64 ts;

  /* remember time exec() was executed for this PID */
  pid = bpf_get_current_pid_tgid() >> 32;  // pid를 구함
  ts = bpf_ktime_get_ns();  // 시스템 시간 ns단위로 구하는 헬퍼함수
  bpf_map_update_elem(&exec_start, &pid, &ts, BPF_ANY);

  /* don't emit exec events when minimum duration is specified */
  if (min_duration_ns) return 0;

  /* reserve sample from BPF ringbuf */
  e = bpf_ringbuf_reserve(&rb, sizeof(*e), 0);  // 링버퍼에 공간을 예약해 둠
  if (!e) return 0;

  // 실행중인 프로세스의 task_struct 포인터를 반환
  task = (struct task_struct *)bpf_get_current_task();

  e->exit_event = false; // 이벤트 구조체의 exit_event=false 는 종료 이벤트가 아님을 의미
  e->pid = pid;
  e->ppid = BPF_CORE_READ(task, real_parent, tgid); // 현재 프로세스의 부모 프로세스(ppid)
  bpf_get_current_comm(&e->comm, sizeof(e->comm)); // 현재 프로세스의 이름(comm)을 이벤트 구조
체에 저장
  fname_off = ctx->__data_loc_filename & 0xFFFF;
  bpf_probe_read_str(&e->filename, sizeof(e->filename), (void *)ctx + fname_off);

  /* successfully submit it to user-space for post-processing */
  bpf_ringbuf_submit(e, 0); // 링 버퍼에 데이터를 제출하여 사용자 공간으로 전달함
  return 0;
}

SEC("tp/sched/sched_process_exit")     // 프로세스가 종료될때 트리거됨
int handle_exit(struct trace_event_raw_sched_process_template *ctx)
{
  struct task_struct *task;
  struct event *e;
  pid_t pid, tid;
  u64 id, ts, *start_ts, duration_ns = 0;

  /* get PID and TID of exiting thread/process */
  id = bpf_get_current_pid_tgid();
  pid = id >> 32;
  tid = (u32)id;

  /* ignore thread exits */
  if (pid != tid) return 0;

  /* if we recorded start of the process, calculate lifetime duration */
  start_ts = bpf_map_lookup_elem(&exec_start, &pid); //맵을 참고하여 프로세스의 시작시간을 구함
  if (start_ts)
    duration_ns = bpf_ktime_get_ns() - *start_ts;
  else if (min_duration_ns)
```

```
    return 0;
  bpf_map_delete_elem(&exec_start, &pid);

  /* if process didn't live long enough, return early */
  if (min_duration_ns && duration_ns < min_duration_ns) return 0;

  /* reserve sample from BPF ringbuf */
  e = bpf_ringbuf_reserve(&rb, sizeof(*e), 0);// 링버퍼에 공간을 예약해 둠
  if (!e) return 0;

  // 실행중인 프로세스의 task_struct 포인터를 반환
  task = (struct task_struct *)bpf_get_current_task();

  e->exit_event = true;// 이벤트 구조체의 exit_event=true 는 종료 이벤트임을 의미
  e->duration_ns = duration_ns;
  e->pid = pid;
  e->ppid = BPF_CORE_READ(task, real_parent, tgid);
  e->exit_code = (BPF_CORE_READ(task, exit_code) >> 8) & 0xff;
  bpf_get_current_comm(&e->comm, sizeof(e->comm));

  /* send data to user-space for post-processing */
  bpf_ringbuf_submit(e, 0);  // 링 버퍼에 데이터를 제출하여 사용자 공간으로 전달함
  return 0;
}
```

```
//  bootstrap.c
// SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause)
/* Copyright (c) 2020 Facebook */
#include <argp.h>
#include <signal.h>
#include <stdio.h>
#include <time.h>
#include <sys/resource.h>
#include <bpf/libbpf.h>
#include "bootstrap.h"
#include "bootstrap.skel.h"

static struct env {
  bool verbose;
  long min_duration_ms;
} env;

const char *argp_program_version = "bootstrap 0.0";
const char *argp_program_bug_address = "<bpf@vger.kernel.org>";
const char argp_program_doc[] = "BPF bootstrap demo application.\n"
        "\n"
        "It traces process start and exits and shows associated \n"
        "information (filename, process duration, PID and PPID, etc).\n"
        "\n"
        "USAGE: ./bootstrap [-d <min-duration-ms>] [-v]\n";

static const struct argp_option opts[] = {
  { "verbose", 'v', NULL, 0, "Verbose debug output" },
  { "duration", 'd', "DURATION-MS", 0, "Minimum process duration (ms) to report" },
```

```
  {},
};

static error_t parse_arg(int key, char *arg, struct argp_state *state) {
  switch (key) {
  case 'v':
    env.verbose = true;
    break;
  case 'd':   // 프로세스의 최소 duration 시간을 설정 -d=1000 은 1초(1000ms)
    errno = 0;
    env.min_duration_ms = strtol(arg, NULL, 10);
    if (errno || env.min_duration_ms <= 0) {
      fprintf(stderr, "Invalid duration: %s\n", arg);
      argp_usage(state);
    }
    break;
  case ARGP_KEY_ARG:
    argp_usage(state);
    break;
  default:
    return ARGP_ERR_UNKNOWN;
  }
  return 0;
}

static const struct argp argp = {
  .options = opts,
  .parser = parse_arg,
  .doc = argp_program_doc,
};

static int libbpf_print_fn(enum libbpf_print_level level, const char *format, va_list args) {
  if (level == LIBBPF_DEBUG && !env.verbose)
    return 0;
  return vfprintf(stderr, format, args);
}

static volatile bool exiting = false;

static void sig_handler(int sig) {
  exiting = true;
}

static int handle_event(void *ctx, void *data, size_t data_sz) {
  const struct event *e = data;
  struct tm *tm;
  char ts[32];
  time_t t;

  time(&t);
  tm = localtime(&t);
  strftime(ts, sizeof(ts), "%H:%M:%S", tm);

  if (e->exit_event) {  // 프로세스가 종료될때
```

```
    printf("%-8s %-5s %-16s %-7d %-7d [%u]", ts, "EXIT", e->comm, e->pid,e->ppid,e->exit_code);
    // duration_ns 값이 있으면
    if (e->duration_ns) printf(" (%llums)", e->duration_ns / 1000000);
    printf("\n");
  } else {  // 프로세스가 실행될때
   printf("%-8s %-5s %-16s %-7d %-7d %s\n", ts, "EXEC", e->comm, e->pid, e->ppid, e->filename);
  }
  return 0;
}

int main(int argc, char **argv) {
  struct ring_buffer *rb = NULL;
  struct bootstrap_bpf *skel;
  int err;

  /* Parse command line arguments */
  err = argp_parse(&argp, argc, argv, 0, NULL, NULL);
  if (err) return err;

  /* Set up libbpf errors and debug info callback */
  libbpf_set_print(libbpf_print_fn);

  /* Cleaner handling of Ctrl-C */
  signal(SIGINT, sig_handler);   // 시그널 핸들러를 등록
  signal(SIGTERM, sig_handler);

  /* Load and verify BPF application */
  skel = bootstrap_bpf__open();
  if (!skel) { fprintf(stderr, "Failed to open and load BPF skeleton\n"); return 1; }

  // BPF 코드의 min_duration_ns 값을 설정
  skel->rodata->min_duration_ns = env.min_duration_ms * 1000000ULL;

  err = bootstrap_bpf__load(skel);
  if (err) { fprintf(stderr, "Failed to load and verify BPF skeleton\n"); goto cleanup; }

  err = bootstrap_bpf__attach(skel);
  if (err) { fprintf(stderr, "Failed to attach BPF skeleton\n"); goto cleanup; }

  // eBPF 프로그램과 사용자 공간 사이의 데이터 전송을 위한 링 버퍼를 설정.
     handle_event 는 링퍼에서 이벤트를 처리할 콜백 함수
  rb = ring_buffer__new(bpf_map__fd(skel->maps.rb), handle_event, NULL, NULL);
  if (!rb) { err = -1; fprintf(stderr, "Failed to create ring buffer\n"); goto cleanup; }

  printf("%-8s %-5s %-16s %-7s %-7s %s\n", "TIME", "EVENT", "COMM", "PID", "PPID", "FILENAME/EX
IT CODE");
  while (!exiting) {
    err = ring_buffer__poll(rb, 100 /* timeout, ms */); // 링버퍼를 폴링하여 새로운 이벤트를 처
리. 새로운 데이터가 없을 경우 타임아웃(100ms) 시간동안 대기
    /* Ctrl-C will cause -EINTR */
    if (err == -EINTR) { err = 0; break; }
    if (err < 0) { printf("Error polling perf buffer: %d\n", err); break; }
  }
```

```
cleanup:
  /* Clean up */
  ring_buffer__free(rb);
  bootstrap_bpf__destroy(skel);

  return err < 0 ? -err : 0;
}
```

```
  clang -target bpf 를 이용하여 BPF 커널 코드를 컴파일
$ clang -g -O2 -target bpf   -D__TARGET_ARCH_x86                  \
      -I$BS/include  -I.  -idirafter /usr/include/x86_64-linux-gnu \
      -c bootstrap.bpf.c    -o bootstrap.bpf.o

  bpftool gen skeleton 명령을 이용하여 헤더 파일 스켈레톤을 생성
$ $BS/bpftool/src/bpftool gen skeleton bootstrap.bpf.o > bootstrap.skel.h

  사용자 공간 프로그램을 컴파일 하고 링크함
$ cc -g -Wall -I$BS/include    -c bootstrap.c -o bootstrap.o
$ cc -g -Wall bootstrap.o  $BS/libbpf/src/libbpf.a   -lelf -lz -o bootstrap
```

```
$ sudo strace -fe bpf ./bootstrap
TIME     EVENT COMM             PID     PPID    FILENAME/EXIT CODE
04:35:53 EXIT  kworker/dying    33652   2       [0]
04:36:02 EXIT  bash             33290   33289   [0]
04:36:02 EXIT  sudo             33289   33288   [1]
04:36:02 EXIT  sudo             33288   32611   [0]
04:36:03 EXEC  ls               33749   32611   /usr/bin/ls
[...]    "sleep 1"을 실행 시작과 종료시 트리거됨
05:23:04 EXEC  sleep            34204   34193   /usr/bin/sleep
05:23:05 EXIT  sleep            34204   34193   [0] (1000ms)
```

[참고자료]

- Visualizing System Performance with RHEL, Kernel Metric Graphing with Performance Co-Pilot, Grafana, and Bpftrace https://www.redhat.com/en/blog/visualizing-system-performance-rhel-8-part-3-kernel-metric-graphing-performance-co-pilot-grafana-and-bpftrace
- Bpftrace Recipes: 5 Real Problems Solved https://www.youtube.com/watch?v=wMtArNjRYXU
- Demystifying eBPF Tracing: A Beginner's Guide to Performance Optimization https://www.groundcover.com/ebpf/ebpf-tracing
- eBPF in CPU Scheduler https://lpc.events/event/11/contributions/954/ https://www.youtube.com/watch?v=CgB7JpSL5cs
- FreeBSD DTrace Tutorial https://wiki.freebsd.org/DTrace/Tutorial
- DTrace QuickStart http://www.tablespace.net/quicksheet/dtrace-quickstart.html
- Dynamic Tracing Guide https://illumos.org/books/dtrace/bookinfo.html#bookinfo
- eBPF: Unlocking the Kernel https://www.youtube.com/watch?v=Wb_vD3XZYOA
- LISA21 - BPF Internal, Brendan Gregg, https://www.brendangregg.com/Slides/LISA2021_BPF_Internals/ https://www.youtube.com/watch?v=_5Z2AU7QTH4
- OpenDTrace Specification 1.0, Aug. 2018. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-924.pdf
- DTrace-on-Windows https://github.com/microsoft/DTrace-on-Windows?tab=readme-ov-file
- BPF Performance Tools, Brendan Gregg, https://www.brendangregg.com/bpf-performance-tools-book.html
- bpftrace Cheat Sheet https://www.brendangregg.com/BPF/bpftrace-cheat-sheet.html
- BPF/eBPF, 최승혁, https://velog.io/@choiish98/BPF
- eBPF assembly with LLVM https://qmonnet.github.io/whirl-offload/2020/04/12/llvm-ebpf-asm/
- eBPF Tutorial by Example https://eunomia.dev/tutorials/
- BPF Portability and CO-RE https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html
- Eunomia BPF https://www.youtube.com/watch?v=pIh4rvtMtTk
- libbpf Overview https://libbpf.readthedocs.io/en/latest/libbpf_overview.html

이 보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24L2M1C6
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

ISBN 978-89-294-1799-4 (PDF)