Why DTrace Should Be In Your Toolkit

Ruud van der Pas

Director, OpenMP Architecture Review Board

Multicore World 2026

Christchurch, New Zealand, February 16-20, 2026

Outline

- ***Two Questions***
- ***What is DTrace?***
- ***Some DTrace Concepts***
- ***Scratching the DTrace Surface***
- ***Q & (some) A***

Two Questions

What's Going On?

- ***Serious infrastructure outages***
  - ***Large scale, e.g. Crowdstrike***
  - ***Very high impact, e.g. an airline check-in system that is down***
- ***In all such cases, the outage is very disruptive and expensive***
  - ***How much attention is given to prevent these outages***
  - ***Are there any tools to help?***

Conventional Troubleshooting Tools

- ***Quite a number of troubleshooting tools exist***
    - ***strace - trace system calls***
    - ***ftrace - kernel tracing***
    - ***netstat - network diagnostics***
    - ***iostat - I/O diagnostics***
    - ***…***
- ***Great tools, but a restricted focus and no customization***

What is DTrace?

The Origin of DTrace

"If a complicated system is doing something unexpected, how do you determine what it is doing and why, without taking the system down?"

This lead to the development of DTrace

My One Line Description of DTrace

"DTrace is your microscope into the Operating System"

What is DTrace?

- ***A dynamic tracing tool***
  - ✳ ***But really more than anything else, a programming language***
- ***Traces activities in both userland and the kernel***
  - ✳ ***And both at the same time***
  - ✳ ***Provides a truly holistic view***
- ***Designed to be safe to use on a production system***

How does it Work?

- The user writes a program/script in the D language
  - ✳ Don't worry, "D" is like a shell language with C/awk flavors
- Full control over what needs to be traced
  - ✳ Can trace multiple different events simultaneously
  - ✳ Also the reporting of events is under user control
- Provides a high degree of flexibility and customization

Some DTrace Concepts

DTrace Probes

- ***Probes are events that are available to trace***
    - ***For example, the entry/return to a specific (kernel) function***
- ***Every system with DTrace installed has many probes***
    - ***Typically in the order of 100,000 or more***
- ***Bind actions to any of the probes being traced***
    - ***The D language is used to define the action(s) to be taken***

The Structure of a Probe

```
provider:module:function:name
[/ predicate /]
{
  D statements
}
```

Examples of Probe Definitions

An example

```
syscall:vmlinux:bind:entry
  {printf("I am in function %s\n",probefunc);}
```

Use a blank field, a wildcard, and a predicate:

```
syscall::bind*:entry
/ my_flag == 1 /
{
  printf("I am in function %s\n",probefunc);
}
```

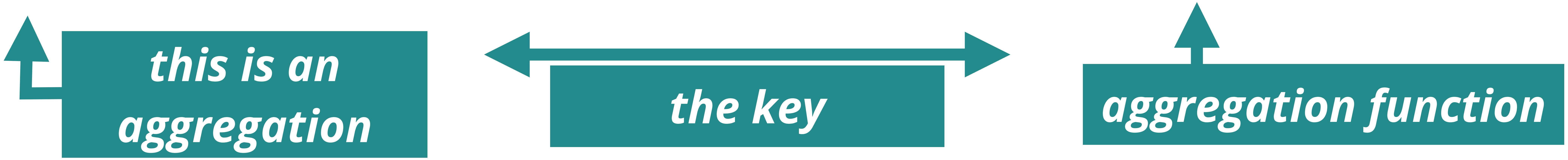
The pid provider: pid$target

- The examples later on, use the pid provider: pid$target
- Variable $target expands to the process id of the target
- This means that all the tracing is restricted to the target
    - Without this, all system activities are traced
    - That is totally fine, of course, but not what we'll do today

Aggregations

- A very powerful data structure in DTrace
  - ✳ Stores the result of an aggregation function (e.g. min/max)
  - ✳ Indexed through an arbitrary key (e.g. integer, string, etc.)

```
@nicolas-headcount[2026,"christchurch"] = count();
```

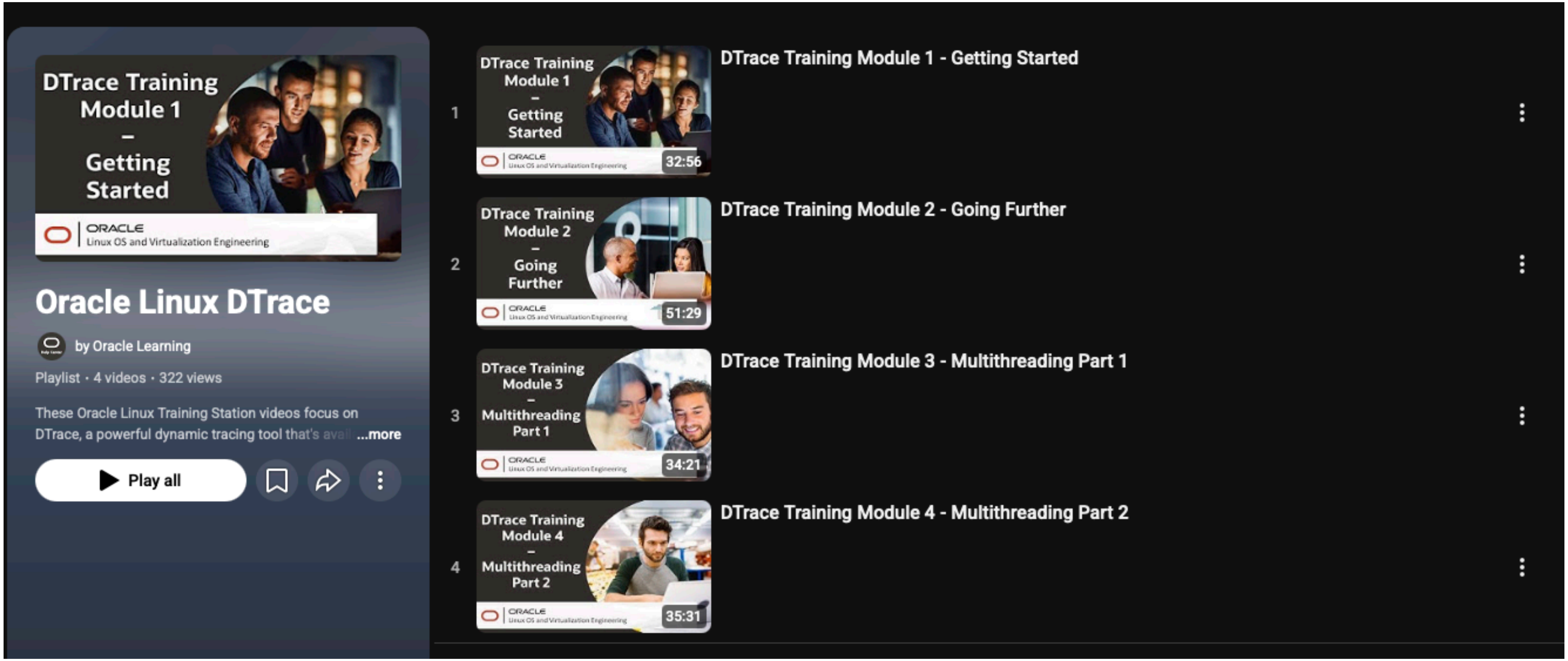


this is an aggregation

the key

aggregation function

A Starting Point to Learn More



Oracle Linux DTrace

by Oracle Learning

Playlist · 4 videos · 322 views

These Oracle Linux Training Station videos focus on DTrace, a powerful dynamic tracing tool that's avai ...more

1. DTrace Training Module 1 - Getting Started (32:56)
2. DTrace Training Module 2 - Going Further (51:29)
3. DTrace Training Module 3 - Multithreading Part 1 (34:21)
4. DTrace Training Module 4 - Multithreading Part 2 (35:31)

DTrace Training Module 1 – Getting Started
ORACLE Linux OS and Virtualization Engineering
Play all

https://www.youtube.com/@OracleLearning => Oracle Linux DTrace

Scratching the DTrace Surface

Monday Morning 9am - Why Me?

Hey Rudd,

Tomorrow we are launching the world leading BlackBox application.

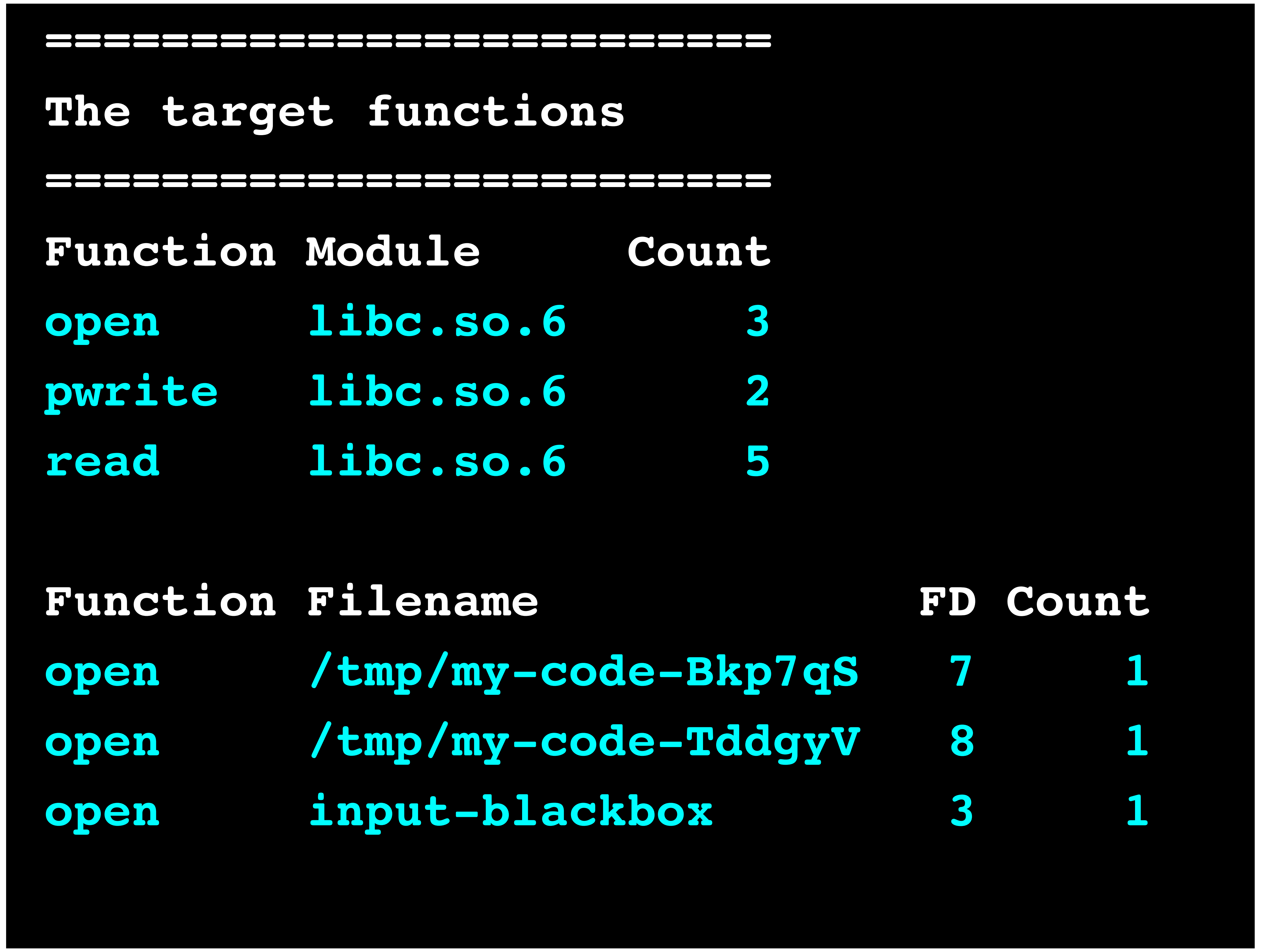
For the all important "go/no go" WhatAreWeDoingHere meeting that starts today at 10am, can you give me the I/O characteristics for this application?

We need to know the filenames used, their file descriptors and how many bytes are read and written on a per filename basis.

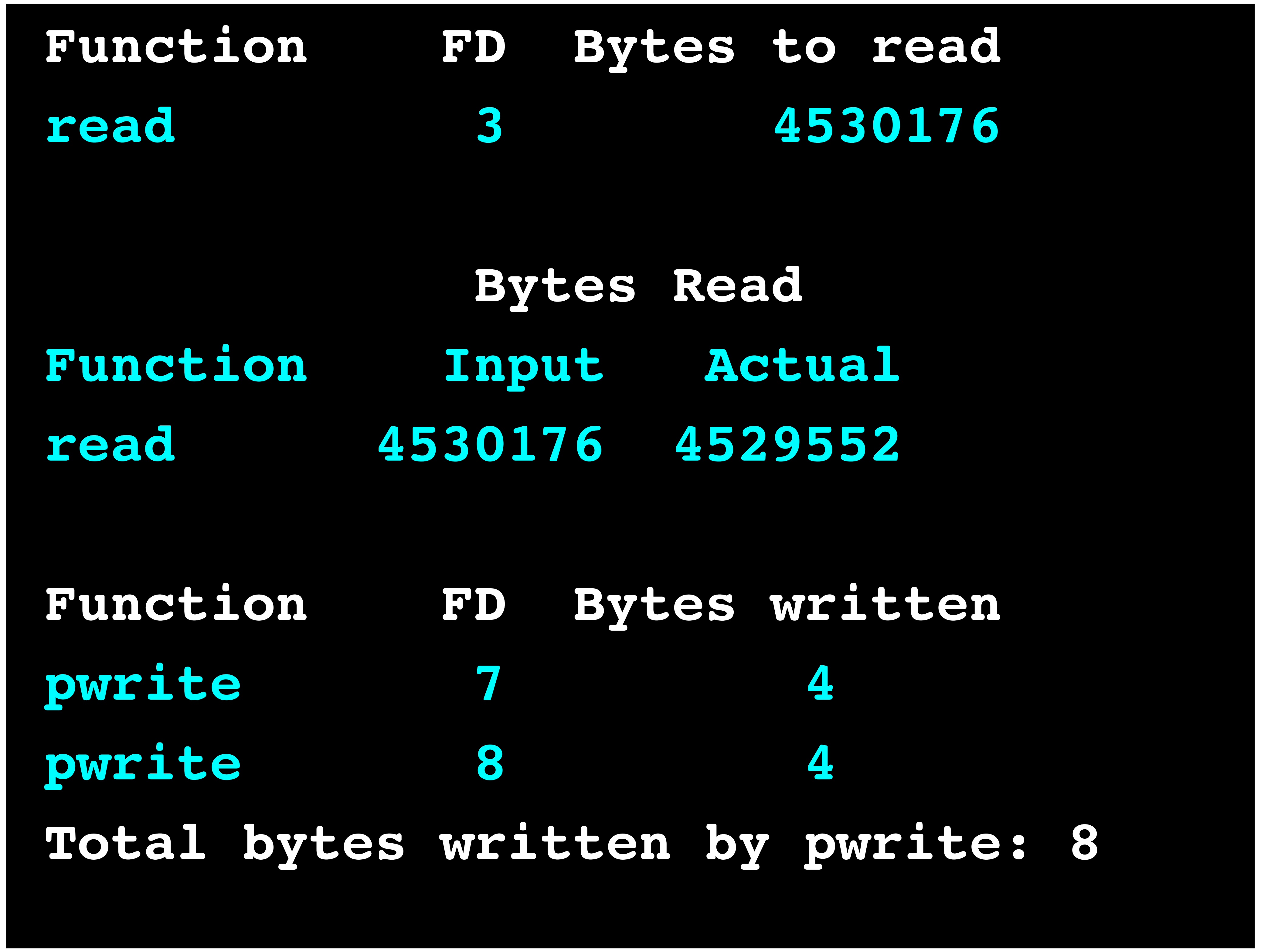
Thanks.

A. Hole, VP of AGT (Advanced Generic Technology)

For the Impatient - The Results



```
=========================
The target functions
=========================
Function Module    Count
open     libc.so.6     3
pwrite   libc.so.6     2
read     libc.so.6     5

Function Filename             FD Count
open     /tmp/my-code-Bkp7qS   7     1
open     /tmp/my-code-TddgyV   8     1
open     input-blackbox        3     1
```



```
Function    FD  Bytes to read
read         3        4530176

             Bytes Read
Function    Input   Actual
read      4530176  4529552

Function    FD  Bytes written
pwrite       7            4
pwrite       8            4
Total bytes written by pwrite: 8
```

The Target Application

```
$ ./blackbox -f input-blackbox
$
```

That's correct. We have no idea what is going on in this code.

Let's find out!

The Strategy

- *Cast a wide net first: find all potentially relevant functions*
  - *Leverage the support for empty fields and wildcards*
- *Write and test probes for each piece of information needed*
  - *It is easy to do this and focus on one thing at a time*
- *Merge the scripts into one overall script*

Cast a Wide Net - Script scan-wide-net.d

```
pid$target::*read*:entry,
pid$target::*write*:entry,
pid$target::*open*:entry
{
   @target_calls[probefunc,probemod] = count();
}
```

The probes (use a blank field for the module and leverage wildcards)

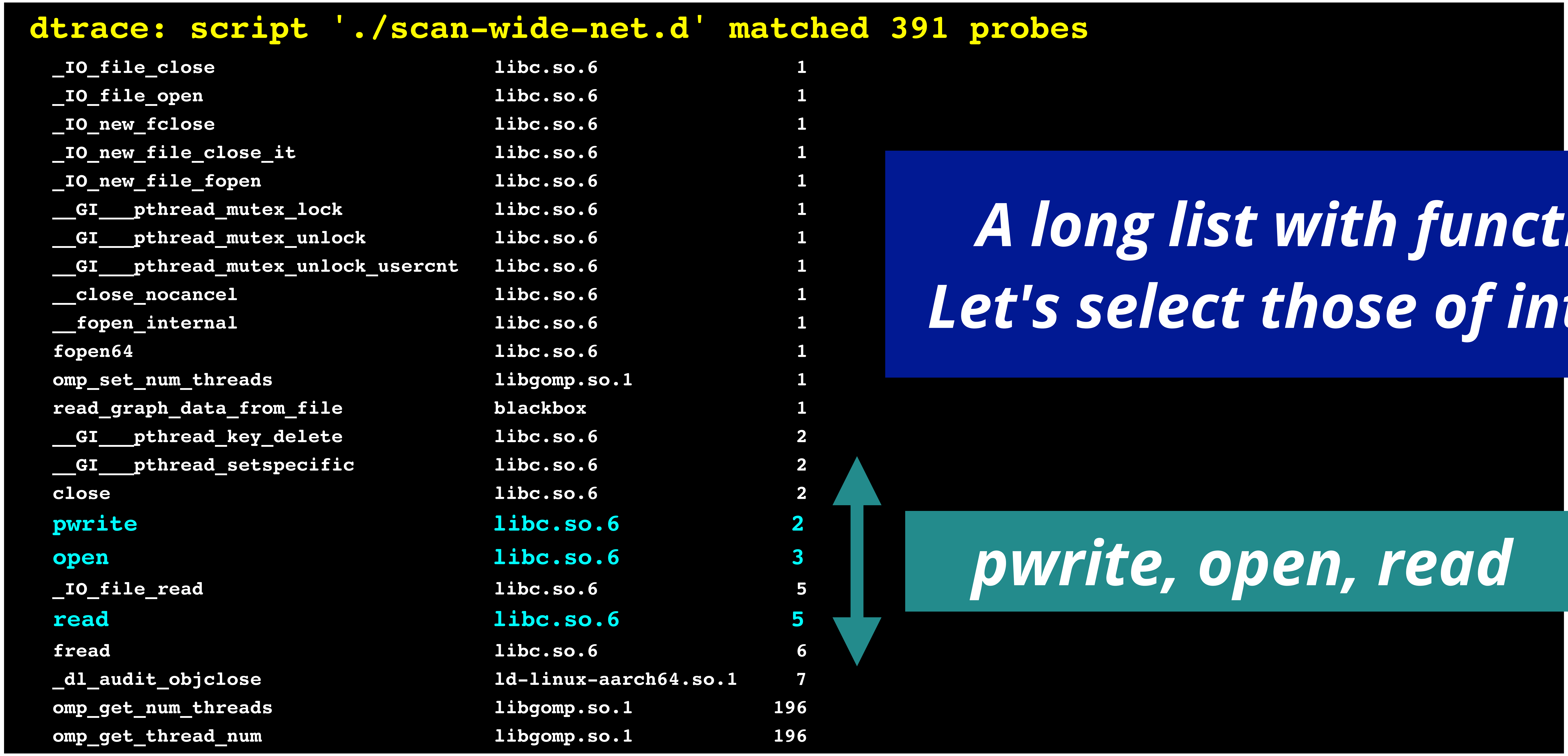
function name

module name

Trace all functions with read, write, open, or close in the name

DTrace in Action

```
$ sudo dtrace -s scan-wide-net.d -c "./blackbox -f input-blackbox"
```



```
dtrace: script './scan-wide-net.d' matched 391 probes
    _IO_file_close                     libc.so.6                1
    _IO_file_open                      libc.so.6                1
    _IO_new_fclose                     libc.so.6                1
    _IO_new_file_close_it              libc.so.6                1
    _IO_new_file_fopen                 libc.so.6                1
    __GI___pthread_mutex_lock          libc.so.6                1
    __GI___pthread_mutex_unlock        libc.so.6                1
    __GI___pthread_mutex_unlock_usercnt libc.so.6               1
    __close_nocancel                   libc.so.6                1
    __fopen_internal                   libc.so.6                1
    fopen64                            libc.so.6                1
    omp_set_num_threads                libgomp.so.1             1
    read_graph_data_from_file          blackbox                 1
    __GI___pthread_key_delete          libc.so.6                2
    __GI___pthread_setspecific         libc.so.6                2
    close                              libc.so.6                2
    pwrite                             libc.so.6                2
    open                               libc.so.6                3
    _IO_file_read                      libc.so.6                5
    read                               libc.so.6                5
    fread                              libc.so.6                6
    _dl_audit_objclose                 ld-linux-aarch64.so.1    7
    omp_get_num_threads                libgomp.so.1           196
    omp_get_thread_num                 libgomp.so.1           196
```

A long list with functions
Let's select those of interest

pwrite, open, read

The Selected Functions

From the man pages of open, read, and pwrite:

```
fd = open(const char *pathname, int flags,/* mode_t mode */ );

bytes_read = read(int fd, void buf[.count], size_t count);

bytes_written = pwrite(int fd, const void buf[.count],

                size_t count, off_t offset);
```

The First Six Lines

Every script shown next, starts with these lines:

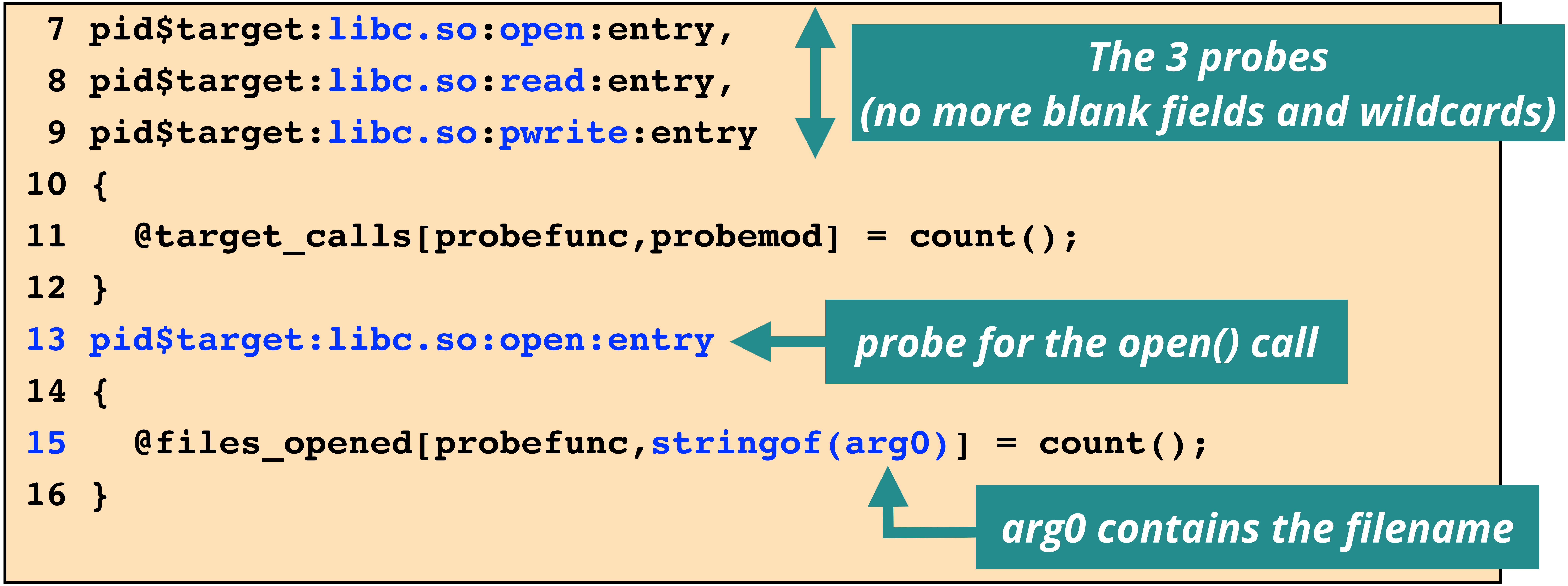
```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option aggsortkey=1
#pragma D option aggsortkeypos=0

```

execute script under control of dtrace

suppress default output

Print aggregations sorted by the first colum

Zoom in on Functions of Interest - Script scan.d/1



```
pid$target:libc.so:open:entry,
pid$target:libc.so:read:entry,
pid$target:libc.so:pwrite:entry
{
  @target_calls[probefunc,probemod] = count();
}
pid$target:libc.so:open:entry
{
  @files_opened[probefunc,stringof(arg0)] = count();
}
```

The 3 probes
(no more blank fields and wildcards)

probe for the open() call

arg0 contains the filename

There are 3 functions that we are interested in

Zoom in on Functions of Interest - Script scan.d/3

```
END
{
  printf("%s\n","=========================");
  printf("The target functions\n");
  printf("%s\n","=========================");
  printf("%-8s %-10s %5s\n","Function","Module","Count");
  printa("%-8s %-10s %@5d\n",@target_calls);

  printf("\n%-8s %20s %6s\n","Function","Filename","Count");
  printa("%-8s %20s %@6d\n",@files_opened);
}
```

Print the results in the END probe (this is almost half of the code)

Get the Results!

```
$ sudo scan.d -c "./blackbox -f input-blackbox"
```

```
=========================
The target functions
=========================
Function Module      Count
open     libc.so.6       3
pwrite   libc.so.6       2
read     libc.so.6       5

Function             Filename  Count
open      /tmp/my-code-Famb8X      1
open      /tmp/my-code-fgUczk      1
open           input-blackbox      1
```

Now we know the filenames in use

How about the file descriptors?

New Approach - Connect Filenames and Descriptors

```
pid$target:libc.so:open:entry
{
  self->fname_open = copyinstr(arg0);
}
pid$target:libc.so:open:return
/ self->fname_open != 0 /
{
  @files_and_fd_opened[probefunc,
                       self->fname_open,
                       arg1] = count();
  self->fname_open = 0;
}
```

capture the filename
(copy from user space to a DTrace buffer)

reference the filename

arg1 contains the file descriptor

In the return probe we link the filename and descriptor

Print the Results

```
END
{
  printf("\nThe mapping between filenames and descriptorss\n");
  printf("%-8s %-20s %2s %5s\n","Function","Filename","FD","Count");
  printa("%-8s %-20s %2d %@5d\n",@files_and_fd_opened);
}
```

Print the header and the aggregation

Get the Results!

```
$ sudo file-mapping.d -c "./blackbox -f input-blackbox"
```

```
The mapping between filenames and descriptorss
Function Filename                 FD Count
open     /tmp/my-code-6mMFMC       8     1
open     /tmp/my-code-y89cUY       7     1
open     input-blackbox            3     1
```

Now we know the filenames in use and their file descriptors

Number of Bytes Read - 2 Probes

```
pid$target:libc.so:read:entry
{
  @total_bytes_read[probefunc,arg0] = sum(arg2);
  @total_bytes_input[probefunc]     = sum(arg2);
}
pid$target:libc.so:read:return
{
  @total_bytes_actually_read[probefunc] = sum(arg1);
}
```

arg0 contains the file descriptor

arg2 contains the number of bytes to be read

arg1: the number of bytes actually read

This tells us how many bytes have been read

Print the Results in the END Probe

```
END
{
  printf("\n%-8s %5s  %13s\n","Function","FD","Bytes to read");
  printa("%-8s %5d  %@13d\n",@total_bytes_read);

  printf("\n%23s\n","Bytes Read");
  printf("%-8s %8s %8s\n","Function","Input","Actual");
  printa("%-8s %@8d %@8d\n",@total_bytes_input,
                            @total_bytes_actually_read);
}
```

Both the number of bytes to be read, and delivered, are shown

Get the Results!

```
$ sudo read.d -c "./blackbox -f input-blackbox"
```

```
Function      FD  Bytes to read
read           3        4530176


               Bytes Read
Function      Input    Actual
read        4530176   4529552
```

We know how many bytes have been read

But why a difference between the 2 numbers?

Number of Bytes Written

```
pid$target:libc.so:pwrite:entry
{
  @pwrite_bytes[probefunc,arg0]  = sum(arg2);
  @pwrite_total_bytes[probefunc] = sum(arg2);
}
END
{
  printf("\n%-8s %5s  %s\n","Function","FD","Bytes written");
  printa("%-8s %5d  %@8d\n",@pwrite_bytes);
  printa("Total bytes written by %s: %@d\n",@pwrite_total_bytes);
}
```

arg0 contains the file descriptor

arg2 contains the number of bytes to be written

This probe is very similar to the one for the read() function

Get the Results!

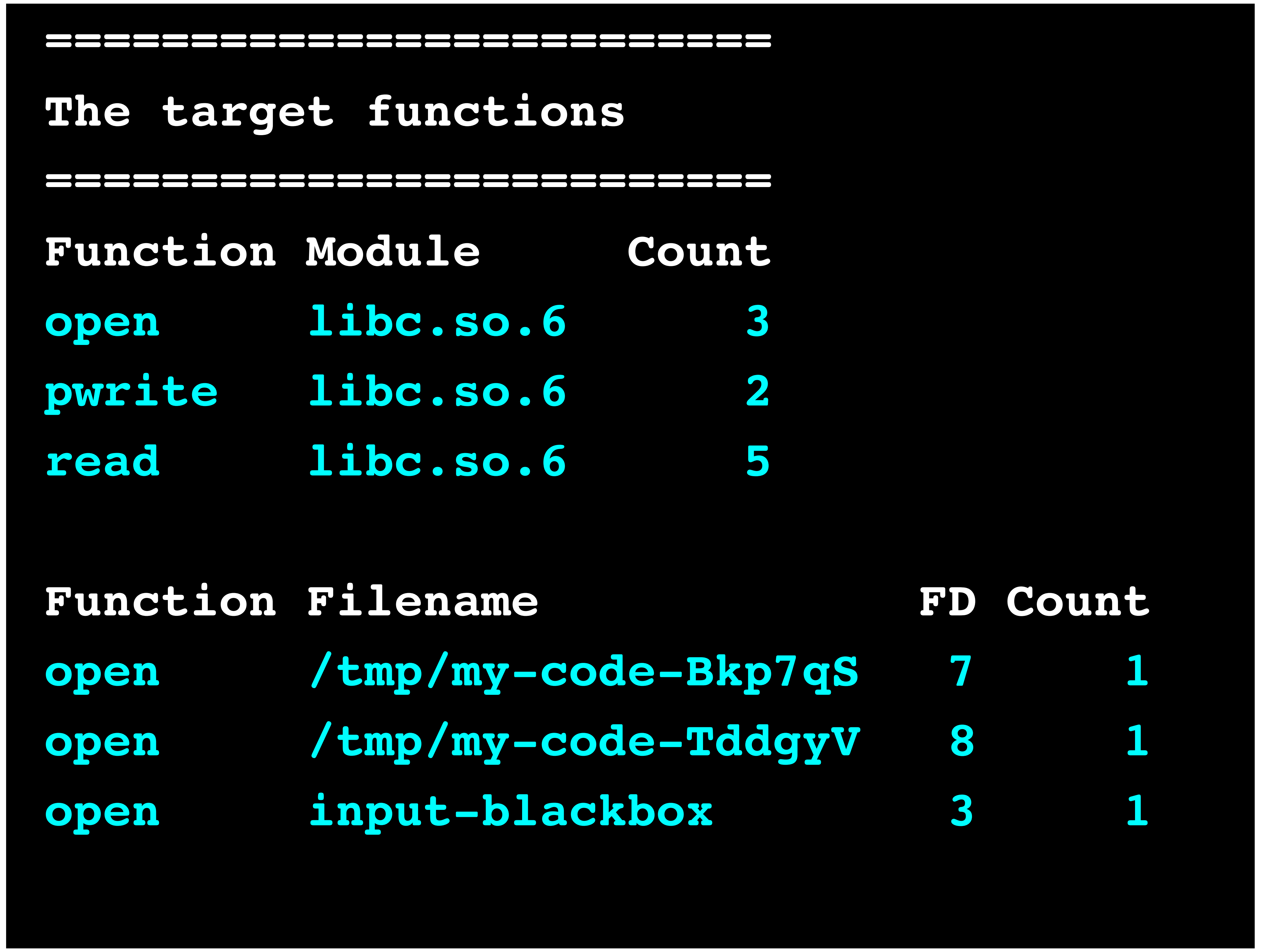
```
$ sudo write.d "./blackbox -f input-blackbox"
```

```
Function     FD  Bytes written
pwrite        7            4
pwrite        8            4
Total bytes written by pwrite: 8
```

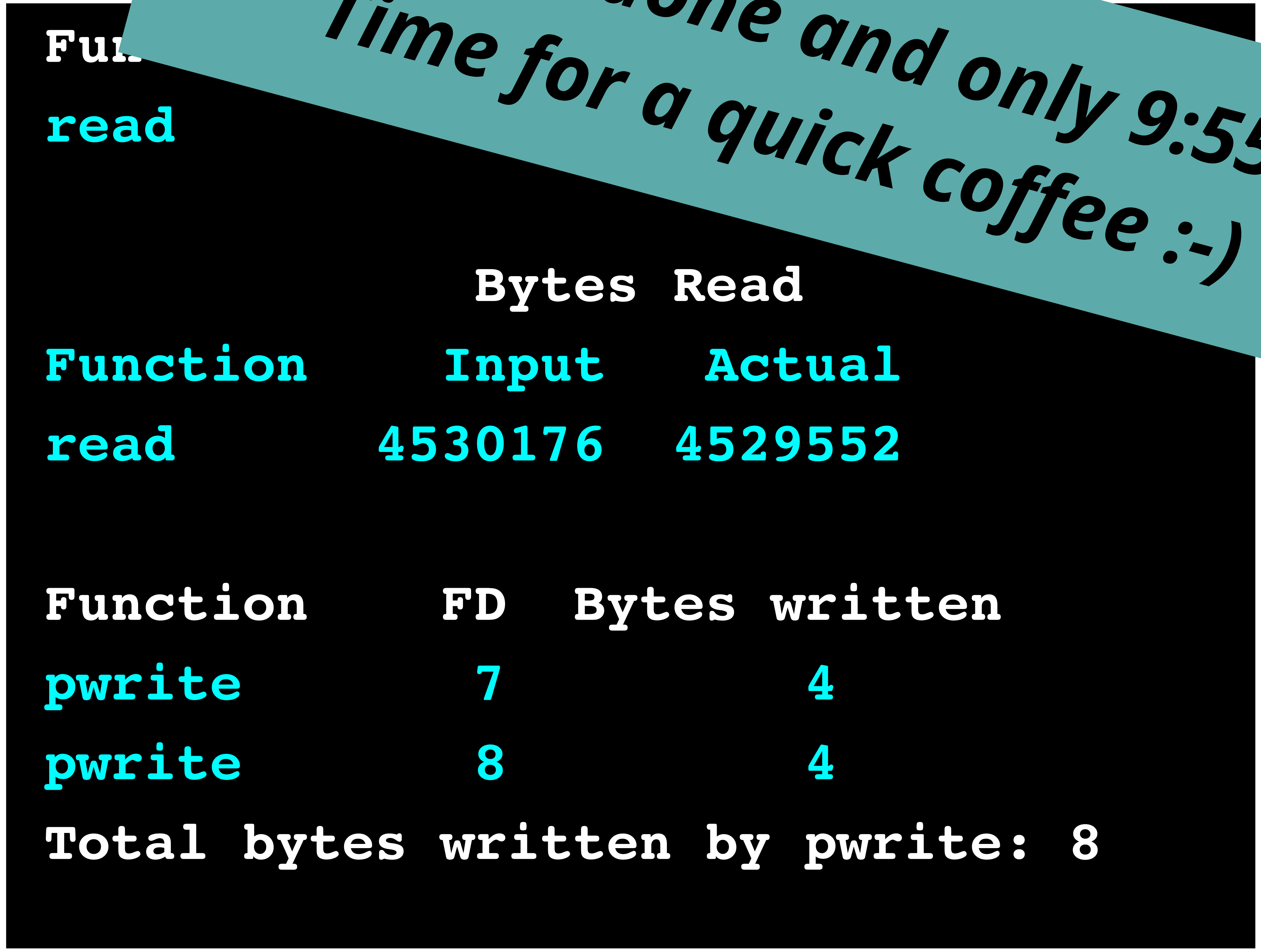
Only 4 bytes written per file

Putting It All Together (plus se leanup)

Phew! It's done and only 9:55am.
Time for a quick coffee :-)



```
=============================
The target functions
=============================
Function Module      Count
open     libc.so.6       3
pwrite   libc.so.6       2
read     libc.so.6       5

Function Filename           FD  Count
open     /tmp/my-code-Bkp7qS  7      1
open     /tmp/my-code-TddgyV  8      1
open     input-blackbox       3      1
```



```
Fu
read


               Bytes Read
Function     Input    Actual
read       4530176   4529552


Function     FD   Bytes written
pwrite        7             4
pwrite        8             4
Total bytes written by pwrite: 8
```

Q & (some) A

Thank You And ... Stay Tuned!