

Securing the Soft Underbelly of a Supercomputer with BPF Probes

Author: Billy Wilson

Wednesday, March 17, 2021



PREPARED BY:

CSIAAC Cyber Security & Information Systems
Information Analysis Center
Operated by Quanterion Solutions Incorporated

266 Genesee Street
Utica, NY 13502

<https://www.quanterion.com>
qinfo@quanterion.com

Abstract

With the evolution of modern technology, many organizations are now embracing big data solutions to drive decision-making by uncovering patterns, trends, and correlations in massive amount of raw data. Subsequently, the adoption of big data analytic technologies presents multi-faceted challenges to manage voluminous data while mitigating data security risks.

Computer scientists and engineers across DoD laboratories are finding different means to collect, synthesize, process, and compare data in order to make the most of scientific observations. The capabilities of grid computing to connect large-scale computers to share resources is generating a surplus of unstructured data to analyze. Big data and high-performance computing (HPC) are hot button subjects amongst academic, industrial, and government organizations. Scientists and engineers believe that high performance computing resources can significantly advance scientific research and discovery.

In 2015, the White House published Executive Order 13702 to create a National Strategic Computing Initiative (NSCI). The NSCI was established to promote U.S. leadership in High Performance Computing (HPC) to maximize benefits of high-performance computing (HPC) research, advancement of economic competitiveness, development, and scientific discovery. Most recently, U.S. computing leaders, including Department of Energy Laboratories, have partnered with government, universities, and private sector to launch the COVID-19 High Performance Computing Consortium. The consortium will allow researchers worldwide to access to the world's most powerful HPC resources in support of COVID-19 research.

The primary objective of HPC systems is to ensure the most resourceful execution of large-scale data analytics, which dictates lightweight security measures with the intention of reducing the overhead coupled with security requirements. Cybersecurity for HPC is a critical mission aspect that presents unique challenges in providing non-repudiation, thus providing a high level of data protection and confidentiality for scientific observations. In this special report, we delineate methods for closing HPC Security Gaps by using Berkeley Packet Filter (BPF) as part of a network load balancer. Berkeley Packet Filter (BFP) was designed in the 1990s as a virtual machine for efficient packet filters. This report will discuss how BFP is used for monitoring, debugging, and collecting statistics from the kernel. This special report is geared toward developers and users who want to understand HPC and BPF broader functionality as part of the Kernel Runtime Security to assist with improving detection of security threats.

Table of Contents

1. Introduction	4
2. High-Performance Computing (HPC) Ecosystem	4
3. HPC Cluster Architecture	5
4. BPF Introduction.....	7
4.1. BPF Compiler Collection	8
4.2. bpftrace	10
5. Performance Analysis of BPF in HPC	13
5.1. Test Environment	14
5.2. Benchmarking Software	15
5.3. BPF probes.....	16
5.4. Simulating Low-Profile Attacks.....	20
6. Results.....	20
6.1. Performance Results	20
6.2. Detection Results	23
7. Conclusions and Future Work	26
8. References.....	28

1. Introduction

Today, research organizations are the principal operators of high-performance computing (HPC) systems. Researchers utilize HPC to fast-track scientific discovery while adhering to security control standards to protect data files which could degrade the undertaking of the high-volume calculations. One standard solution is to secure a set of login nodes that mediate access to an enclave of lightly monitored compute nodes, referred to as “the soft underbelly of a supercomputer” by one DoD representative (National, 2016). Recent advances in the BPF subsystem, a Linux tracing technology, have provided a new means to monitor compute nodes with minimal performance degradation. Well-crafted BPF traces can detect malicious activity on an HPC cluster without slowing down systems or the researchers that depend on them. In this paper, a series of low-profile attacks are conducted against a compute cluster under heavy computational load, and BPF probes are attached to detect the attacks. The probes successfully log all attacks, and performance loss is less than one percent for all benchmarks save for one inconclusive set.

2. High-Performance Computing (HPC) Ecosystem

In high-performance computing (HPC), many organizations that facilitate research provide a remote shell for writing, compiling, and executing arbitrary code. The code runs on a networked cluster of servers with hundreds of thousands of processor cores and has access to petabytes of storage. Information security practitioners must secure these environments for government research contracts, but the solutions they architect cannot reduce bare-metal cluster performance by more than a defined percentage, possibly as low as 1%. These limitations impact the security of HPC sites in government agencies, academia, and the private sector.

Colloquially known as “supercomputers,” HPC clusters utilize numerous machines to deliver a more powerful computing environment to deal with computational problems that are too massive for conventional computers. Cluster sizes range from dozens to tens of thousands of “nodes” (HPC parlance for servers). Today, the Summit supercomputer at the Department of Energy’s Oak Ridge National Laboratory ranks number one on the Top500 Supercomputing list, touting over 2,400,000 processor cores and peaking at 200 petaflops (i.e., two hundred quadrillion floating-point operations per second) (TOP500, 2019).

In practice, large clusters share their resources among many users, including those not employed by the host institution. For example, the XSEDE Federation is a cyberinfrastructure ecosystem composed of 36 different institutions across the United States, providing HPC resources to the science and engineering community as a single coordinated effort (XSEDE, n.d.). This author administers an HPC cluster at an academic institution, which serves not only the campus community but also collaborators from other organizations across the world.

A current approach to HPC security is to lock down a few login nodes with required security controls and only lightly monitor the army of isolated compute nodes behind them. At a NIST Workshop on HPC Security in 2016, a DoD representative described these compute nodes as the “soft underbelly” of

supercomputing (National, 2016). Detecting malicious activity on the compute nodes themselves while maintaining performance requirements was considered an unsolved problem.

Three years ago, Brendan Gregg announced that “superpowers have finally come to Linux” in the form of Berkeley Packet Filter (BPF) tracing tools (Gregg, 2017). Although systems administrators and analysts had used BPF to filter network packets for decades, Linux kernel developers had both improved its performance and opened it up to general usage through a new `bpf()` syscall. Thus, BPF was no longer a network tracing tool but a system-wide tracing tool. Gregg has since demonstrated the value of BPF tracing to security practitioners at subsequent conferences (Gregg & Maestretti, 2017).

This paper has two primary purposes. The first is to introduce BPF as a general tracing tool for detecting malicious activity on Linux systems. A summary of recent developments in BPF and an explanation of its usage is provided. Example scripts are also included that demonstrate tracing open TTYS, network activity, filesystem activity, and Bash commands.

The second purpose is to evaluate BPF as a security tool for production HPC clusters, both from a performance perspective and a detection perspective. A security monitoring agent that affects performance by even one or two percent has a low chance of adoption on HPC clusters that prioritize fast research results. Should it be adopted, there must be an assurance that the agent will not slow down compute nodes and will detect the attacks it purports to defend against.

To validate BPF, a series of low-profile attacks are conducted against eight compute nodes running a series of benchmarks, both without and with BPF probes attached. Benchmarks without BPF probes are compared to benchmarks with BPF probes to determine the performance loss. The logs of the BPF trace scripts are compared with attack script logs to determine the attack detection rate.

3. HPC Cluster Architecture

System monitoring of a large-scale High-Performance Community (HPC) cluster architecture is a difficult task which could easily become further challenging as the scale and complexity of the platforms increase.

The complexity of an HPC cluster can range from elementary to mind-boggling. At one end of the spectrum, students can interconnect a stack of Raspberry Pis to make a Beowulf cluster for educational purposes (Kiepert, 2013). At the other end is the NASA Advanced Supercomputing Division, whose Pleiades cluster interconnects eleven thousand compute nodes in an 11-dimensional hypercube topology for performance purposes (Chang, Jin & Bauer, 2016).

An HPC node provides a multi-compiler and multi-version environment intended to support scientific software from many different disciplines. For example, the author administers nodes that have eight versions of `gcc`, two versions of Intel compilers, five versions of CUDA libraries, and three versions of Boost C++ libraries, not including additional variations when compiled with MPI support.

Researchers initially authenticate to a “login” node session. From here, they can write, compile, and debug arbitrary code. Once a researcher is ready to launch their software on the compute nodes, they submit a “job” to the scheduler. The job specifies the resources needed, the time required, and the commands to run. The scheduler maintains a queue of all jobs, dispatching them to the compute nodes as time, resources, and fair share permit.

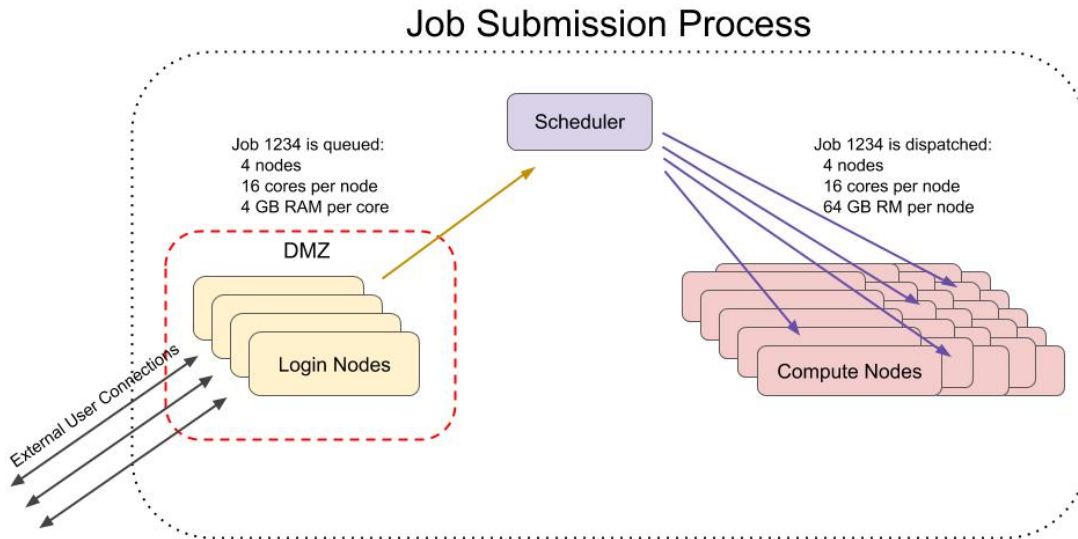


Figure 1. Job Submission Process (Source: Author, Billy Wilson)

Compute node operating systems are installed using scalable provisioning technologies such as PXE booting, Kickstart for thick provisioning, and read-only root NFS for thin provisioning, among others. The nodes are also configured to mount central storage to make data available for processing across a large set of nodes, with performance tiers ranging from archival tape storage to high-performance parallel filesystems.

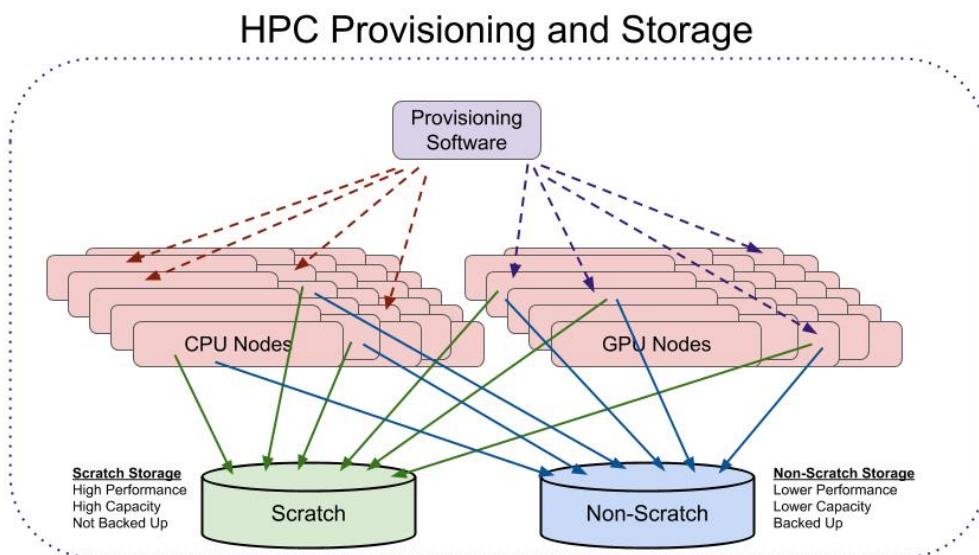


Figure 2. HPC Provisioning and Storage (Source: Author, Billy Wilson)

Firewalls and network monitors are typically not deployed for compute nodes. Generally, it is due to the thought that the head node would have the overall ability to interact with the compute nodes so keeping the security controls at that gateway could be sufficient if that is the only entry/exit point for media and traffic. With network speeds reaching tens of gigabits per second per node, or terabytes per second in aggregate, host-based and network-based products can degrade performance and cause job failures. Also, the network traffic itself is highly variable. Software may use traditional Ethernet or high-bandwidth, low-latency fabrics such as InfiniBand and Omni-Path. The characteristics of network traffic differ from software to software and even across the lifetime of a single piece of software as it is developed on HPC systems.

These details highlight the following difficulties for the security practitioner:

- Compute nodes run arbitrary code;
- Compute nodes can access centralized research storage;
- Compute nodes produce highly variable network traffic;
- Compute nodes have fewer security controls for performance reasons; and
- Compute nodes produce terabytes of network traffic per second in aggregate.

4. BPF Introduction

Many security practitioners identify the filter expression of tcpdump as BPF, but this is somewhat inaccurate. Tcpdump transparently compiles the expression into BPF code. The actual BPF bytecode can be dumped using the `-d` option. This bytecode is fed into a register-based virtual machine that runs in the Linux kernel.

```
[root@m7-10-16 tools]# tcpdump -d 'host 192.168.10.1'
(000) ldh      [14]
(001) jeq     #0x800          jt 2    jf 6
(002) ld      [28]
(003) jeq     #0xc0a80a01     jt 12   jf 4
(004) ld      [32]
(005) jeq     #0xc0a80a01     jt 12   jf 13
(006) jeq     #0x806          jt 8    jf 7
(007) jeq     #0x8035         jt 8    jf 13
(008) ld      [30]
(009) jeq     #0xc0a80a01     jt 12   jf 10
(010) ld      [40]
(011) jeq     #0xc0a80a01     jt 12   jf 13
(012) ret     #262144
(013) ret     #0
```

Figure 3. Dump of BPF bytecode from tcpdump (Source: Author, Billy Wilson)

Originally implemented in 1992, the two-register virtual machine approach of “BSD Packet Filter” was twenty to one hundred times faster than its competing packet filters, partly because the implementation matched how the underlying RISC CPU operated, and partly because of its improved buffer model (McCanne, 1992).

Support for BPF in Linux was added in the 2.5 development kernel and stayed largely untouched for roughly a decade. In the last eight years, however, BPF has changed dramatically, burgeoning into its own Linux subsystem. Many new terms have evolved over the years, the following section provides a brief review of these developments.

In 2012, Will Drewry struggled to have code accepted into the Linux kernel. He wrote a patch to allow seccomp to filter arbitrary syscalls, but his work was in limbo between a `prctl()` maintainer who suggested using the perf subsystem for filtering, and a perf maintainer who suggested using `prctl()` for filtering, with neither gatekeeper budging (Edge, 2011). In a stroke of brilliance, Will found the BPF virtual machine and used it to filter allowed syscalls instead of network traffic (Corbet, 2012).

Two years later, Alexei Starovoitov posted a patch set that greatly improved BPF performance. He increased the number of registers from two to ten, added to its instruction set to better resemble modern processors, and upgraded its registers to 64 bits (Corbet, 2014 May). His work yielded a four-fold increase in speed (Starovoitov, 2014 March), and importantly, he also posted a patch that demonstrated using BPF for tracing filters (Starovoitov, 2014 May).

A month later, Alexei extended BPF further. He moved BPF out of the network subsystem into its own directory, signaling the intention for its general use. He also implemented a new `bpf()` syscall. This allowed users with `CAP_SYS_ADMIN` privileges (i.e., root) to load BPF programs into the kernel to respond to specific events that they defined. An in-kernel verifier ensured the safety of the program before loading it (Corbet, 2014 July).

This improved BPF implementation went through many names. It was first known as “internal BPF” (as opposed to “classic BPF”) but was later called extended BPF, or eBPF. Today, system maintainers have chosen to simply call the execution engine BPF, without any reference to what the acronym originally represented (Gregg, 2020 January).

4.1. BPF Compiler Collection

While valuable to kernel developers, the `bpf()` syscall was impractical to those who didn’t keep a copy of the kernel source code lying around. The BPF Compiler Collection (BCC) was created in April of 2015 to address this issue. It greatly simplified the process of writing tracing tools that could leverage BPF (Fleming, 2017).

Over the course of a few years, this collection grew into a mature suite of tools that were easy for systems administrators to use. There are currently over 100 BCC tools readily available for monitoring system calls, language function calls (including php, perl, ruby, and python), network events, filesystem performance, database performance, and more. Four basic examples of these tools are included below. These examples are not intended to detect sophisticated attackers, but rather to demonstrate the potential of the tools.

The opensnoop tool traces `open()` and `openat()` syscalls. In this example, the tool detected a user’s failed attempts to list the `/root` directory and view `/etc/shadow`:


```
# /usr/share/bcc/tools/opensnoop -u 1000 -x
PID   COMM          FD ERR PATH
41892  cat            -1  2  /etc/shadow
41905  ls             -1  2  /root
```

Figure 4. Example of “opensnoop” (Source: Author, Billy Wilson)

The execsnoop tool traces new processes via exec() syscalls. This example shows a user attempting to run nc, download ncat, and create and run a suspicious Python script.

```
# /usr/share/bcc/tools/execsnoop
PCOMM      PID   PPID   RET  ARGS
nc          27530 16339   0  /usr/bin/nc evil.org 4444
wget        27540 16339   0  /usr/bin/wget https://github.com/andrew-
d/static-binaries/raw/master/binaries/linux/x86_64/ncat
vim         27642 16339   0  /usr/bin/vim tunnel.py
chmod       27646 16339   0  /usr/bin/chmod u+x tunnel.py
tunnel.py   27648 16339   0  ./tunnel.py
```

Figure 5. Example of “execsnoop” (Source: Author, Billy Wilson)

The ttysnoop tool displays the output of a TTY as if the administrator is sitting at the same terminal. The following example shows an administrator snooping /dev/pts/1 and observing a user named “billy” exploring the system:

```
# /usr/share/bcc/tools/ttysnoop 1
which nmap
/usr/bin/which: no nmap in
(/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin)
billy@testnode /tmp$ which nc
/usr/bin/nc
billy@testnode /tmp$ which ncat
/usr/bin/ncat
billy@testnode /tmp$ route -n
Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric Ref    Use
Iface
0.0.0.0          192.168.10.1   0.0.0.0         UG    100    0      0 em1
billy@testnode /tmp$ for i in 192.168.10.{1..3}; do ping -c1 -w1 $i && echo
$i is up; sleep 5; done
ping: socket: Operation not permitted
ping: socket: Operation not permitted
ping: socket: Operation not permitted
billy@testnode /tmp$
```

Figure 6. Example of “ttysnoop” (Source: Author, Billy Wilson)

Last is the tcpstates tool, used here for tracing any TCP state changes involving remote ports 22, 80, or 443. While the trace was running, a user connected over SSH to a neighboring compute node for 10 seconds and then closed the connection. Next, the user attempted to access a website with wget and then sent a keyboard-interrupt after three failed connection attempts.

```
# /usr/share/bcc/tools/tcpstates -D 22,80,443
SKADDR          C-PID C-COMM      LADDR          LPORT RADDR
RPORT OLDSTATE  -> NEWSTATE   MS
ffff9ac37e622f80 42979 ssh          192.168.10.179 0      192.168.10.178 22
CLOSE          -> SYN_SENT    0.000
ffff9ac37e622f80 0      swapper/12 192.168.10.179 0      192.168.10.178 22
SYN_SENT       -> ESTABLISHED 0.218
ffff9ac37e622f80 42979 ssh          192.168.10.179 0      192.168.10.178 22
ESTABLISHED    -> FIN_WAIT1   10899.358
ffff9ac37e622f80 42979 ssh          192.168.10.179 0      192.168.10.178 22
FIN_WAIT1      -> FIN_WAIT2   0.066
ffff9ac37e622f80 42979 ssh          192.168.10.179 0      192.168.10.178 22
FIN_WAIT2      -> CLOSE       0.003
ffff9ac3874c8000 42988 wget         192.168.10.179 0      52.2.61.110    80
CLOSE          -> SYN_SENT    0.000
ffff9ac3874c8000 0      swapper/12 192.168.10.179 0      52.2.61.110    80
SYN_SENT       -> CLOSE       131001.368
ffff9ac3874c8000 42988 wget         192.168.10.179 0      52.2.61.110    80
CLOSE          -> SYN_SENT    1000.312
ffff9ac3874c8000 0      swapper/12 192.168.10.179 0      52.2.61.110    80
SYN_SENT       -> CLOSE       130071.675
ffff9ac3874c8000 42988 wget         192.168.10.179 0      52.2.61.110    80
CLOSE          -> SYN_SENT    2000.533
ffff9ac3874c8000 0      swapper/12 192.168.10.179 0      52.2.61.110    80
SYN_SENT       -> CLOSE       129071.439
ffff9ac3874c8000 42988 wget         192.168.10.179 0      52.2.61.110    80
CLOSE          -> SYN_SENT    3001.378
ffff9ac3874c8000 42988 wget         192.168.10.179 0      52.2.61.110    80
SYN_SENT       -> CLOSE       8582.302
```

Figure 7. Example of “tcpstates” (Source: Author, Billy Wilson)

While easy to use, BCC tools are not necessarily easy to write or maintain. They are python scripts with embedded BPF programs written in C. Tools may break when the traced code changes, requiring continual maintenance from version to version of the traced software.

4.2. bpftrace

In December 2016, an even more intuitive tool came to fruition as a result of Alastair Robertson’s spare-time hobby. Robertson started a project built on BCC and BPF called bpftrace, and it offered an AWK-like syntax that was already familiar to many systems administrators and security practitioners. The project attracted prominent BCC contributors and completed its first set of major features in 2018 (Gregg, 2020 January). Today, bpftrace is a full-fledged tracing utility that can use a stupendous variety of sources and trigger many types of actions.

The main downside of the tool is that it requires a minimum Linux kernel version of 4.1 and recommends version 4.9 to take full advantage of its features. This means that the tool is only available on later versions of Linux distributions such as Red Hat Enterprise Linux 8, Debian 9, and Ubuntu 19.04. Even then, the version of bpftrace on these distributions does not have all the features available in the latest version.

For those exploring bpftrace for the first time, two helpful starting points are running `bpftrace -l` for a list of static and dynamic probes available for use and `bpftrace -lv [tracepoint_name]` for the arguments available to retrieve values from when a probe fires.

The basic syntax of bpftrace and a few instructive examples are provided. A full walkthrough of writing bpftrace scripts is outside the scope of this paper, but readers who wish to familiarize themselves with using the tool can review Brendan Gregg's bpftrace tutorial¹.

bpftrace scripts follow a basic syntax familiar to AWK users:

```
#!/usr/bin/bpftrace

probe1 /filter/ { action }
probe2, probe3 /filter/ { action }
```

Figure 8. Basic syntax of bpftrace (Source: Author, Billy Wilson)

The following example traces `openat()` calls by UID 1000:

```
#!/usr/bin/bpftrace

BEGIN
{
    printf ("%s\t%s\t%s\n", "COMM", "FILE", "RETVAL");
}

tracepoint:syscalls:sys_enter_openat
/uid == 1000/
{
    @filename[tid] = str(args->filename);
}

tracepoint:syscalls:sys_exit_openat
/uid == 1000 && args->ret < 0/
{
    printf("%s\t%s\t%d\n", comm, @filename[tid], args->ret);
    delete(@filename[tid]);
}
```

Figure 9. Example Script of Tracing `openat()` Syscall (Source: Author, Billy Wilson)

The script prints a header; saves the target filename when UID 1000 enters `openat()`; and prints the command, file, and errno when `openat()` returns an error. It produced the following output when UID 1000 attempted to open `/etc/shadow`.

```
# ./detect_failed_openat.bt
Attaching 3 probes...
COMM      FILE      RETVAL
cat       /etc/shadow  -13
```

Figure 10. Example Output of Tracing `openat()` Syscall (Source: Author, Billy Wilson)

¹ https://github.com/iovisor/bpftrace/blob/master/docs/tutorial_one_liners.md

Userspace functions can also be traced. The following example script from Brendan Gregg (2020 January) traces the `readline()` function in `/bin/bash`. Once started, it will trace `readline()` for all current and future invocations of `/bin/bash`.

```
#!/usr/bin/bpftrace

BEGIN
{
    printf("Tracing bash commands... Hit Ctrl-C to end.\n");
    printf("%-9s %-6s %s\n", "TIME", "PID", "COMMAND");
}

uretprobe:/bin/bash:readline
{
    time("%H:%M:%S ");
    printf("%-6d %s\n", pid, str(retval));
}
```

Figure 11. Example Script of Tracing Bash `readline()` (Source: Author, Billy Wilson)

The script produced the following output, revealing an attempt by a bash session with PID 28853 to invoke a cryptocurrency miner:

```
# /usr/share/bpftrace/tools/bashreadline.bt
Attaching 2 probes...
Tracing bash commands... Hit Ctrl-C to end.
TIME      PID      COMMAND
10:08:30  16339    chmod u+x trace_bash_readline.bt
10:08:31  16339    ./trace_bash_readline.bt
10:08:54  16339    id
10:08:59  16339    cd ~
10:15:25  28853    ./cgminer -o stratum+tcp://mmpool.org:3333 -u jexotic -p
tigercoins4LYFE
10:15:38  16339    vim job.sbatch
```

Figure 12. Example Output of Tracing Bash `readline()` (Source: Author, Billy Wilson)

Shared libraries can also be traced. This is especially valuable because it allows an administrator to place probes that are difficult for an attacker to avoid. The following example script places probes in the `gethost*()` and `getaddrinfo()` functions of the GNU C library to trace DNS queries. It is modified from Brendan Gregg's `gethostlatency.bt` script (Gregg, 2018).

```
#!/usr/bin/bpftrace

BEGIN
{
    printf("%-8s %-6s %-6s %-16s %s\n", "TIME", "UID", "PID", "COMM", "HOST");
}

uprobe:/lib64/libc.so.6:getaddrinfo,
uprobe:/lib64/libc.so.6:gethostbyname,
uprobe:/lib64/libc.so.6:gethostbyname2
{
    time("%H:%M:%S ");
    printf("%-6d %-6d %-16s %s\n", uid, pid, comm, str(arg0));
}
```

Figure 13. Example Script of Tracing GNU C Library (Source: Author, Billy Wilson)

The output shows DNS queries from a user invoking curl and wget on questionable websites:

```
# ./dnsqueries.bt
Attaching 4 probes...
TIME      UID      PID      COMM      HOST
11:15:59 1000     31854    curl      questionablewebsite.cn
11:16:00 1000     31857    wget      a2ng98eh2k0c94782hdo.com
```

Figure 14. Example Output of Tracing GNU C Library (Source: Author, Billy Wilson)

These examples demonstrated the ability of bpftrace to monitor filesystems, processes, user sessions, and network activity. Once installed, the software includes over thirty high-quality scripts that cover dozens of system activities. As Brendan Gregg put it, gaining this depth and breadth of visibility on a Linux system “can feel like having X-ray vision” (Gregg, 2020 January). This level of vision is available to any Linux systems administrator who becomes adept at using the tools.

5. Performance Analysis of BPF in HPC

The remainder of this paper is dedicated to measuring the performance impact of BPF when monitoring compute nodes under heavy load. As crucial as it is to demonstrate the effectiveness of a security solution, HPC administrators likewise need assurance that security tools will not degrade performance beyond a defined threshold.

Brendan Gregg targeted a performance loss of less than 1% when using BPF tools and scripts in production at Netflix (Gregg & Maestretti, 2017). The expectations in this paper’s performance analysis were as follows:

- Performance loss <1%: BPF probes are widely recommended in HPC
- Performance loss 1%-3%: BPF probes are recommended in qualified circumstances
- Performance loss >3%: BPF probes should be revised until performance is acceptable

5.1. Test Environment

Eight compute nodes with identical hardware were reserved for testing. They were connected to an InfiniBand fabric composed of FDR and EDR switches in a CLOS network topology (i.e., a fat-tree topology with multiple roots). The nodes' hardware characteristics were as follows:

Node	
Model	Dell PowerEdge C6320
Sockets	2
Processor	
Model	Intel Xeon E5-2680 v4
Codename	Broadwell
Cores	14
Base Frequency	2.4 GHz
Turbo Frequency	3.3 GHz
L2 Cache	14 x 256 KiB
L3 Cache	35 MiB
Memory	
Size	128 GB (8 x 16 GB DIMMs)
Type	DDR4
Speed	2400 MT/s
Local Disk	
Disk count	1
Disk Size	1 TB
Disk Type	SATA
Disk Speed	7200 RPM
Ethernet	
NIC Model	Intel 82599ES Dual-port SFI/SFP+
Speed	10GbE
InfiniBand	
NIC Model	Mellanox ConnectX-3
Speed	56 Gb/sec (4X FDR)

Figure 15. Hardware Specifications for Compute Nodes (Source: Author, Billy Wilson)

A new operating system image was built that supported BPF tools, benchmarking software, HPC scheduling, centralized storage, and the InfiniBand fabric. A provisioning server presented this image to the compute nodes, which mounted the image as a read-only root to ensure it was identical and unchangeable across all compute nodes. The provisioning server also provided writable partitions that were bind-mounted onto key locations using `/etc/rwtab` and `/etc/statetab`.

The operating system included the following software of interest:

Software	Description	Version
bcc-tools	BPF Compiler Collection and Tools	0.8.0-4.el8
bpftool	BPF Tracer	0.9-3.el8
gcc	GNU Compiler Collection	8.3.1-4.5.el8
kernel	Linux Kernel	4.18.0-147.5.1.el8_1
libibverbs	Libraries for InfiniBand verbs support	22.3
openblas	Linear algebra library	0.3.3-2.el8
openmpi	Message passing library	4.0.1-3.el8
rdma-core	Drivers for InfiniBand support	22.3
slurmd	HPC scheduling client	19.05.0
HPL	High-Performance Linpack Benchmark	2.3 (netlib)

Figure 16. List of Key Software and Versions on Compute Nodes (Source: Author, Billy Wilson)

The Intel cores in these compute nodes were of the Broadwell generation. These were touted to have up to 16 floating-point operations per clock cycle because of the new fuse-multiply-add (FMA) instruction, but real-world runs have shown lower results because the instruction wasn't as generally applicable as other instructions like AVX2. For this analysis, the cores were estimated to provide 12 floating-point operations per cycle.

Each compute node's theoretical max "flops," or floating-point operations per second, is the product of its total processor cores, clock speed (GHz), and floating-point operations per cycle. When estimating 12 operations per cycle, the compute nodes for this analysis had an estimated theoretical max of 806 gigaflops per node.

5.2. Benchmarking Software

A series of High-Performance LINPACK (HPL) Benchmarks were executed on a set of compute nodes; these comparative studies were accomplished with and without Berkeley Packet Filter (BPF) probes, measuring to what degree the BPF probes affected performance.

This HPL Benchmark is used as a reference benchmark to calculate the top-performing supercomputers in the world. In essence, HPL Benchmark is performing numerical linear algebra techniques to solve a series of polynomial equations. It is inherent for the administrator to scale the extent of the problem and to optimize the software in order to attain best outcomes. A HPL Linpack's compiler and parameters are configured to set the problem size, block size, and process geometry. A broad examination of optimization performances is beyond the scope of this paper; however, a reasonable HPL baseline should obtain 75% to 85% of the theoretical max flops of a compute node.

The HPL parameters and characteristics for each grouping of nodes were as follows:

Count	Prob. Size	Blocks	P	Q	Cores	Memory Used	Theoretical Max GFlops
1 node	98000	248	4	7	28	76 GB	806.4
2 nodes	140000	248	7	8	56	156 GB	1612.8
4 nodes	200000	248	8	14	112	320 GB	3225.6
8 nodes	280000	248	14	16	224	627 GB	6451.2

Figure 17. HPL Parameters and Characteristics Per Node Grouping (Source: Author, Billy Wilson)

Using the eight compute nodes, the author ran a total of 32 HPL benchmark tests. First, each individual compute node ran the benchmark. Next, the nodes were grouped into pairs to run the benchmark together. Then, they were grouped into fours. Finally, all eight nodes ran the benchmark as a single cluster, twofold. These tests were all repeated using the BPF probes. For the repetitive tests, a script was executed to simulate low-profile attacks that the BPF probes were projected to detect.

5.3. BPF probes

The BPF execution engine is fast, but it cannot make up for BPF probes that are frequently fired or inherently slow. If a probe is attached to an event that fires millions of times per second, the overhead will add up. In some cases, tracing `malloc()` or `free()` will slow the target application tenfold or more (Gregg, 2020 January). In contrast, an ideal BPF probe will fire infrequently and provide high-value data.

Before writing a BPF probe, it is important to determine the question that needs to be answered. These are the questions that the probes of this performance analysis were written to answer:

- Are compute nodes attempting to send beacons to external systems?
- Are compute nodes running cryptocurrency miners?
- Are compute nodes the source of any suspicious lateral movements?
- Are compute node processes attempting to escalate privileges?
- Are compute nodes using an SSH proxy to connect to external systems?

To this end, four `bpfttrace` scripts were written: `dnssnoop.bt`, `pamsnoop.bt`, `sshtunnel.bt`, and `tcpconnect_filter.sh`. These scripts produced logs in a key-value format for easy parsing. All scripts output the timestamp, script type (`dns`, `pam`, `sshproxy`, `tcp`) and the PID, UID, and command of the process that caused the probe to fire. Each script also output additional data for its unique type.

The first script, `dnssnoop.bt`, logged DNS queries by tracing the relevant function calls in the GNU C library. It took a UID as its first argument on the command line to log only the DNS queries of a given user.


```

dnssnoop.bt
#!/usr/bin/bpftrace -Bline

// Trace glibc for any dns queries by the user

uprobe:/lib64/libc.so.6:getaddrinfo,
uprobe:/lib64/libc.so.6:gethostbyname,
uprobe:/lib64/libc.so.6:gethostbyname2
/uid == $1 /
{
    time("%Y-%m-%d %H:%M:%S ");
    printf("type=dns pid=%d uid=%d comm=%s query=%s\n", pid, uid, comm,
str(arg0));
}

```

Figure 18. Content of dnssnoop.bt Script (Source: Author, Billy Wilson)

The second script, pamsnoop.bt, detected processes changing from one user to another by tracing Linux PAM, the library responsible for handling authentication tasks. Its first argument on the command line specified the UID to monitor. It logged both the original user and the new user (target) associated with the process. It also logged the return value of the traced function.

```

pamsnoop.bt
#!/usr/bin/bpftrace -Bline

// Trace PAM library to detect when a user changes to another user

uprobe:/lib64/libpam.so.0:pam_modutil_getpwnam
/uid == $1/
{
    @seen[tid] = 1;
    @uid[tid] = uid;
    @user[tid] = str(arg1);
}

uprobe:/lib64/libpam.so.0:pam_modutil_getpwnam
/@seen[tid] && uid != @uid[tid]/
{
    time("%Y-%m-%d %H:%M:%S ");
    printf("type=pam pid=%d uid=%d comm=%s ", pid, uid, comm);
    printf("user=%s target=%s retval=%d\n", @user[tid], str(arg1), retval);
}

END
{
    clear(@seen);
    clear(@uid);
    clear(@user);
}

```

Figure 19. Content of pamsnoop.bt Script (Source: Author, Billy Wilson)

The third script, sstunnel.bt, detected when SSH was used to forward TCP ports. TCP port forwarding is a built-in SSH feature that allows someone to use an SSH server as a proxy to reach external resources.

This feature can be disabled on SSH servers with the AllowTCPForwarding family of SSH options, but it is on by default and is often left that way.

The script detected port forwarding by tracing the `inet_sock_set_state()` syscall and logging whenever an SSH client changed a socket to a LISTEN state. Note that while SSH servers regularly open listening ports, a client opening a listening port is a tell-tale sign of port forwarding.

Specifically, the script detected local, or dynamic, port forwarding. Local port forwarding specifies an SSH server, a remote host and port to connect to, and a local port to open. Any network connections to the local port will be forwarded through the SSH server to the remote host and port. Dynamic port forwarding turns the SSH client into a SOCKS proxy, allowing software to connect to the local port and forward all traffic through an SSH server.

```
sshtunnel.bt
#!/usr/bin/bpftrace -Bline

// Trace inet_sock_set_state() to detect SSH clients attempting local and
dynamic port forwarding

tracepoint:sock:inet_sock_set_state
/comm == "ssh"/
{
    if (args->newstate == 10 && args->protocol == 6)
    {
        time("%Y-%m-%d %H:%M:%S ");
        printf("type=sshproxy pid=%d uid=%d comm=%s listening_port=%d \n", pid,
uid, comm, args->sport);
    }
}
```

Figure 20. Content of `sshtunnel.bt` Script (Source: Author, Billy Wilson)

The final script, `tcpconnect_filter.sh`, was the most complex of the four. This was because the version of `bpftrace` available on RHEL 8.1 was still missing key functionality for network tracing. It lacked features such as integer casting, `strncmp()`, and an `array[]` operator, making it impossible to retrieve data from some of the most valuable networking data structures.

Dale Hamel wrote the original `tcpconnect.bt`, which traced the `tcp_connect()` kernel function to detect all TCP connects (Hamel, 2018). The script was modified for this research, which included wrapping it in a Bash script to enable the whitelisting of a subnet. This allowed the compute cluster's subnet to be whitelisted so that only TCP connections to external resources would cause probes to fire.

Also note that this script only traced TCP traffic which is protocol; unlike the more traditional `traceroute` which sends UDP connectionless Internet protocol. Since UDP is a stateless protocol, it will cause the probe to fire on every sent message.

```

tcpconnect_filter.sh
#!/usr/bin/bash

if [ "$#" -ne 2 ]; then
    echo "Usage: $0 [whitelist-network-address] [whitelist-subnet-mask]"
    exit 1
fi

IFS=. read -s n1 n2 n3 n4 <<< $(echo $1)
IFS=. read -s s1 s2 s3 s4 <<< $(echo $2)

/usr/bin/bpftrace -Bline -e'

#include <net/sock.h>

kprobe:tcp_connect
{
    $sk = ((struct sock *) arg0);
    $inet_family = $sk->__sk_common.skc_family;
    $af_inet = 2;

    if ($inet_family == $af_inet) {
        $daddr = $sk->__sk_common.skc_daddr;
        $saddr = $sk->__sk_common.skc_rcv_saddr;
        $sport = $sk->__sk_common.skc_num;
        $dport = $sk->__sk_common.skc_dport;

        // Destination port is big endian, it must be flipped
        $dport = ($dport >> 8) | (($dport << 8) & 0x00FF00);

        // Filter Network Address
        $filter_addr = ($1) + ($2 << 8) + ($3 << 16) + ($4 << 24);

        // Filter Netmask
        $filter_netmask = ($5) + ($6 << 8) + ($7 << 16) + ($8 << 24);

        if (($daddr & $filter_netmask) != ($filter_addr & $filter_netmask)) {

            time("%Y-%m-%d %H:%M:%S ");
            printf("type=tcp_connect pid=%d uid=%d comm=%s ", pid, uid, comm);
            printf("saddr=%s sport=%d daddr=%s dport=%d\n", ntop($af_inet,
$saddr), $sport, ntop($af_inet, $daddr), $dport);
        }
    }
}' $n1 $n2 $n3 $n4 $s1 $s2 $s3 $s4

```

Figure 21. Content of tcpconnect_filter.sh Script (Source: Author, Billy Wilson)

These scripts were copied to each compute node and executed at the beginning of benchmarks that measured BPF performance impact. Their logs were redirected to files on a shared storage system.

5.4. Simulating Low-Profile Attacks

For tests with BPF probes attached, a Bash script was launched on participating nodes that simulated low-profile attacks. The script produced a timestamped log for each action. Every 1 to 15 seconds, it performed one of the following actions as an unprivileged user:

- Triggered a DNS query of an external domain
- Escalated to a privileged user
- Opened an SSH tunnel
- Attempted a TCP connection to a random private IP

For DNS queries, the script randomly chose a domain from a list, many of them representing common bitcoin mining sites. It then chose from one of five command-line tools to trigger the query: curl, wget, python, dig, or host.

For privilege escalation, the unprivileged user was temporarily given privileges to use sudo to escalate to the root user on the compute nodes. The script ran a basic command as root after escalation. This simulated an adversary who had obtained control over an account with sudo privileges and subsequently escalated to root via regular administrative techniques. It did not represent privilege escalation via software flaw exploitations.

For SSH tunneling, the script randomly chose between local port forwarding and dynamic port forwarding. Local port forwards connected to an SSH server in the DMZ and opened a tunnel to <https://ubuntu.com> using a random local port. Dynamic port forwards connected to that same server and opened a random local port for SOCKS proxy use.

For TCP connection attempts, the script used Bash's built-in `/dev/tcp` feature. This is not an actual device on the filesystem, but a device emulated by Bash for easy interaction with TCP sockets. Any I/O to `/dev/tcp/[host]/[port]` triggers a TCP connection attempt to that host and port. The script chose a random host in the 192.168.0.0/16 subnet and a random port, attempting to connect to it over TCP.

The full body of the low-profile attack script can be found in Appendix A.

6. Results

The results of the benchmarks were analyzed both from a performance perspective and a detection perspective to show whether BPF tracing scripts can sufficiently detect attacks on compute nodes without degrading performance for researchers.

6.1. Performance Results

HPL results indicated that the BPF probes had less than 1% impact on compute node performance. In many cases, HPL benchmarks with BPF enabled recorded higher gigaflops than the non-BPF benchmarks. The author did not interpret these gains to mean that BPF probes improve performance, as such a claim for a tracing tool cannot be concluded. At best, these discrepancies suggested that BPF had nearly zero performance impact on compute nodes. Perhaps more realistically, the gains may have suggested that

other factors besides BPF also influenced compute node performance. Possibilities include thermal fluctuations that impact Intel Turbo Mode, congestion on the InfiniBand fabric, and normal jitter from system processes.

Each chart below was scaled based on the theoretical max gigaflops for a node count. Thus, the vertical axis for the single-node chart has a maximum of 806.4 gigaflops, while the vertical axis for the eight-node chart has a maximum of 6451.2 gigaflops, which is eight times greater. Performance percentages are calculated against these maximums.

For single-node runs, Node 1 suffered a performance loss of 0.41%, the greatest loss of all tests apart from a discrepancy with eight-node runs. Node 8 was the only node with no performance loss, instead recording a gain of 0.02%.

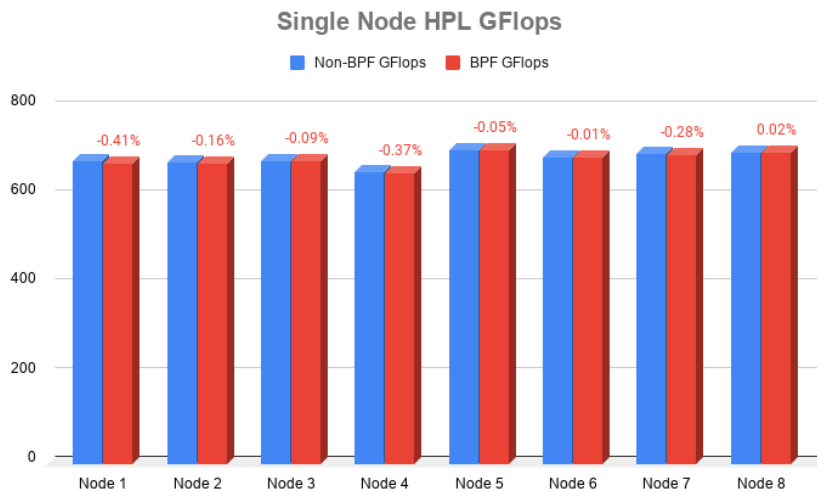


Figure 22. Chart of Single Node HPL Gigaflops (Source: Author, Billy Wilson)

For two-node runs, there were no instances of BPF causing performance loss. Node pairs [1-2] and [7-8] tied for the smallest gain of 0.12%, while node pair [3-4] had the largest gain of 0.65%.

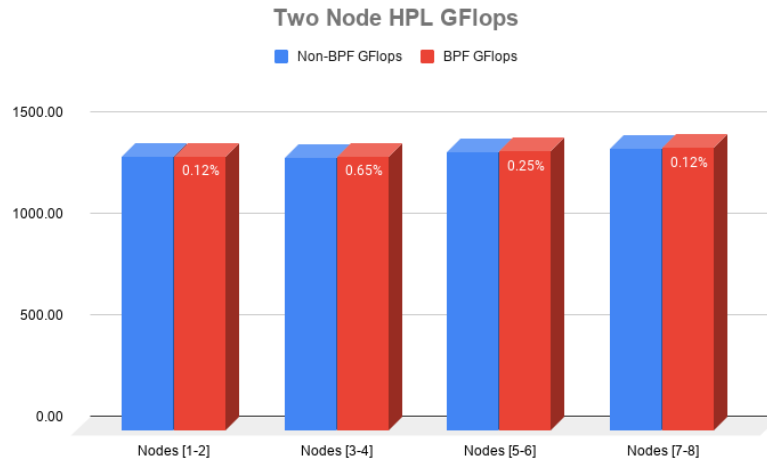


Figure 23. Chart of Two-Node HPL Gigaflops (Source: Author, Billy Wilson)

The four-node runs likewise recorded small gains for BPF-enabled runs, with nodes [1-4] gaining 0.09% and nodes [5-8] gaining 0.13%.

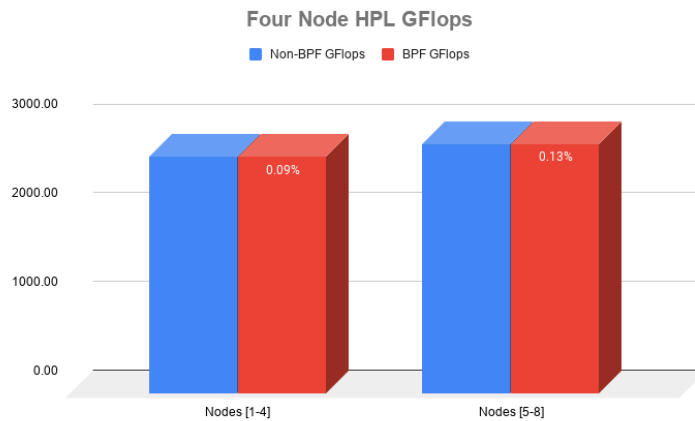


Figure 24. Chart of Four-Node HPL Gigaflops (Source: Author, Billy Wilson)

The eight-node HPL results were unusual enough to warrant discussion. These benchmarks were run twice, meaning that there were two non-BPF results and two BPF results. Depending on how they were paired, the benchmarks either supported that BPF had low performance impact or painted a picture of unexplainable performance differences from run to run.

The four eight-node benchmarks can be paired in two ways. The chart on the left below in Figure 25 shows the results when the low-performing non-BPF and BPF runs are paired together and the high-performing non-BPF and BPF runs are paired together. The chart on the right show the results when the non-BPF and BPF runs that ran first are paired together and the non-BPF and BPF runs that ran second are paired together. Depending on how the results are paired, very different outcomes are seen.

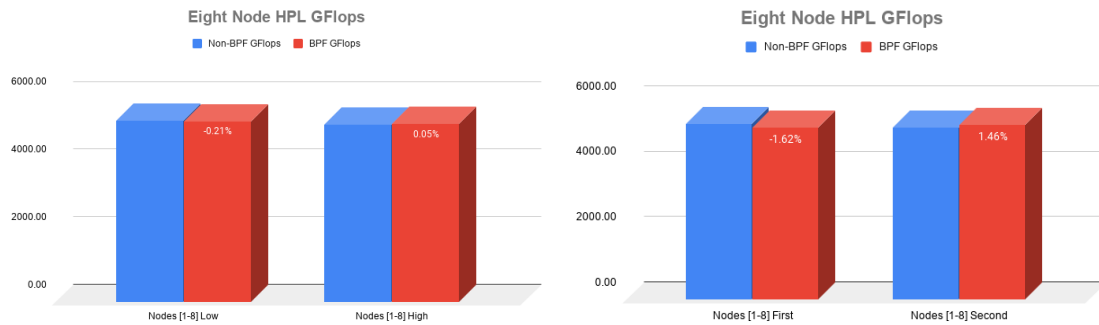


Figure 25. Eight-Node HPL Gigaflops, Paired by Performance and Execution Order (Source: Author, Billy Wilson)

When paired by lows and highs, enabling BPF caused a performance loss of 0.21% in the “low” benchmarks and a gain of 0.05% in the “high” benchmarks. However, when paired chronologically, enabling BPF caused a 1.62% loss in the first benchmarks and a 1.46% gain in the second benchmarks.

Put differently, the results of the two non-BPF runs that used identical hardware and software had a delta of 100 gigaflops, and likewise for the two runs with BPF enabled. Such a delta would only make sense if outside factors such as equipment temperatures or network congestion influenced the results.

Because of these discrepancies, the author found the eight-node results to be inconclusive. Perhaps the larger conclusion to be drawn is that BPF traces caused a performance change somewhere between a 1.62% loss or 1.46% gain for eight-node runs. Taking the averages of the non-BPF and BPF benchmarks, performance loss was only 0.08%. Ultimately, the discrepancies are best resolved with further benchmark testing.

A table of all performance results can be found in Appendix B.

6.2. Detection Results

The logs of the four bpfftrace scripts were cross-checked against logs from the low-profile attack script to determine whether the BPF probes were adequate in detecting unwanted behavior.

Overall, the author found that while the *performance* of the bpfftrace scripts was exemplary, the *fidelity* of the scripts was hampered by excess noise. Scripts often produced multiple logs for a single action of the attack script. They also produced logs that were triggered by the benchmark software itself.

The dnssnoop.bt script created logs not only for domain-based host lookups but also for IP-based host lookups, including those handled by the /etc/hosts file. This was especially apparent as HPL began IPC communications with itself and other cluster members. Logged queries included all participating nodes, as well as domains and IPs that pointed to localhost.

dnssnoop.bt Log Excerpt						
2020-04-27	13:55:13	type=dns	pid=24567	uid=12345	comm=mpirun	query=node-1
2020-04-27	13:55:13	type=dns	pid=24567	uid=12345	comm=mpirun	query=node-2
2020-04-27	13:55:13	type=dns	pid=24567	uid=12345	comm=mpirun	query=node-2
2020-04-27	13:55:13	type=dns	pid=24567	uid=12345	comm=mpirun	query=node-3
2020-04-27	13:55:13	type=dns	pid=24567	uid=12345	comm=mpirun	query=node-3
2020-04-27	13:55:13	type=dns	pid=24567	uid=12345	comm=mpirun	query=node-4
2020-04-27	13:55:13	type=dns	pid=24567	uid=12345	comm=mpirun	query=node-4
...						
2020-04-27	13:55:13	type=dns	pid=24685	uid=12345	comm=xhpl	query=node-1
2020-04-27	13:55:13	type=dns	pid=24584	uid=12345	comm=xhpl	query=localhost
2020-04-27	13:55:13	type=dns	pid=24584	uid=12345	comm=xhpl	query=192.168.10.11
2020-04-27	13:55:13	type=dns	pid=24584	uid=12345	comm=xhpl	query=fe80::dead:beef:dead:beef
2020-04-27	13:55:13	type=dns	pid=24584	uid=12345	comm=xhpl	query=192.168.10.11
2020-04-27	13:55:13	type=dns	pid=24584	uid=12345	comm=xhpl	query=fe80::dead:beef:dead:beef
2020-04-27	13:55:13	type=dns	pid=24584	uid=12345	comm=xhpl	query=127.0.0.1
2020-04-27	13:55:13	type=dns	pid=24584	uid=12345	comm=xhpl	query>:::1

Figure 26. Log Excerpt of dnssnoop.bt (Source: Author, Billy Wilson)

When DNS queries involved dig or host commands, the query was obfuscated. It could be concluded that this was due to the query being routed through systemd-resolved but this was unconfirmed.

Low Profile Attack Log Excerpt						
2020-04-27	15:47:28	type=dns	comm=dig	url=reuters.com		
dnssnoop.bt Log Excerpt						
2020-04-27	15:47:29	type=dns	pid=5997	uid=12345	comm=isc-worker0000	query=127.0.0.1
2020-04-27	15:47:29	type=dns	pid=5997	uid=12345	comm=isc-worker0000	query>:::1

Figure 27. Correlation of dig Execution with dnssnoop.bt Log (Source: Author, Billy Wilson)

However, all queries using wget, curl, and python produced one accurate log per action.

Low Profile Attack Log Excerpt						
2020-04-27	14:33:11	type=dns	comm=wget	url=btc.top		
2020-04-27	14:34:28	type=dns	comm=python	url=nanopool.org		
2020-04-27	14:36:17	type=dns	comm=curl	url=nanopool.org		
dnssnoop.bt Log Excerpt						
2020-04-27	14:33:13	type=dns	pid=1401	uid=12345	comm=wget	query=btc.top
2020-04-27	14:34:30	type=dns	pid=1491	uid=12345	comm=python	query=nanopool.org
2020-04-27	14:36:19	type=dns	pid=1752	uid=12345	comm=curl	query=nanopool.org

Figure 28. Correlation of wget, python, and curl with dnssnoop.bt Log (Source: Author, Billy Wilson)

The pamsnoop.bt script successfully detected sudo attempts, with the caveat that three logs were produced per sudo attempt. For every set of triplets, one log had a non-zero return value, so it should

be possible to reduce log output to one line per successful sudo attempt when filtered by the return value.

Low Profile Attack Log Excerpt
2020-04-27 16:23:16 type=sudo comm=sudo user=billy
pamnoop.bt Log Excerpt
2020-04-27 16:23:18 type=pam pid=8928 uid=0 comm=sudo user=billy target=root retval=0
2020-04-27 16:23:18 type=pam pid=8928 uid=0 comm=sudo user=billy target=root retval=0
2020-04-27 16:23:18 type=pam pid=8928 uid=0 comm=sudo user=billy target=root retval=-810366576

Figure 29. Correlation of sudo Attempt with pamnoop.bt Log (Source: Author, Billy Wilson)

The sstunnel.bt successfully detected all SSH port forwarding connections, but it produced two logs per connection.

Low Profile Attack Log Excerpt
2020-04-27 14:02:47 type=ssh_proxy comm=ssh port_fwd_options=-D 43323
2020-04-27 14:02:59 type=ssh_proxy comm=ssh port_fwd_options=-L 32337:ubuntu.com:443
sshtunnel.bt Log Excerpt
2020-04-27 14:02:49 type=sshproxy pid=31319 uid=12345 comm=ssh listening_port=43323
2020-04-27 14:02:49 type=sshproxy pid=31319 uid=12345 comm=ssh listening_port=43323
2020-04-27 14:03:01 type=sshproxy pid=31340 uid=12345 comm=ssh listening_port=32337
2020-04-27 14:03:01 type=sshproxy pid=31340 uid=12345 comm=ssh listening_port=32337

Figure 30. Correlation of SSH Port Forwarding Attempt with sstunnel.bt Log (Source: Author, Billy Wilson)

Finally, the tcpconnect_filter.sh script successfully detected all TCP connection attempts to resources outside of the compute subnet. The fact that no compute nodes were included in the logs suggested that the subnet whitelist worked properly. One oversight was that the script did not exclude localhost TCP connections.

tcpconnect filter.sh Log Excerpt
2020-04-27 13:55:12 type=tcp_connect pid=30335 uid=12345 comm=xhpl saddr=127.0.0.1 sport=40400 daddr=127.0.0.1 dport=46757
2020-04-27 13:55:26 type=tcp_connect pid=30712 uid=12345 comm=wget saddr=192.168.10.11 sport=48552 daddr=47.254.4.118 dport=80
2020-04-27 13:55:42 type=tcp_connect pid=30726 uid=12345 comm=curl saddr=192.168.10.11 sport=47822 daddr=47.52.122.155 dport=80
2020-04-27 13:55:57 type=tcp_connect pid=30740 uid=12345 comm=bash saddr=192.168.10.11 sport=39408 daddr=192.168.144.84 dport=18044
2020-04-27 13:56:31 type=tcp_connect pid=30782 uid=12345 comm=bash saddr=192.168.10.11 sport=38216 daddr=192.168.228.27 dport=31399
2020-04-27 13:57:29 type=tcp_connect pid=30894 uid=12345 comm=ssh saddr=192.168.10.11 sport=42908 daddr=10.10.10.15 dport=22

Figure 31. Log Excerpt of tcpconnect_filter.sh (Source: Author, Billy Wilson)

The table below provides an aggregated count of the logs across all runs. Each row pairs an attack log type with its associated bpftrace script log type.

The dnssnoop.bt script proved the noisiest. It produced logs for /etc/hosts lookups, domain lookups, and IP lookups, as well as DNS queries performed by the attack script. The pamsnoop.bt script produced exactly three times as many logs as sudo attempts. The sstunnel.bt script produced twice as many logs as SSH proxy attempts, minus one. There was a single time when an ssh_proxy attack log produced only one bpftrace log. The tcpconnect_filter.sh script detected all TCP connections outside the subnet, including connections to the SSH server in the DMZ, but the script also errantly included the many localhost communications by HPL.

Attack Script Type	Attack Log Count	bpftrace Script	Bpftrace Log Count
dns	1703	dnssnoop.bt	19627
sudo	1720	pamsnoop.bt	5160
ssh_proxy	1713	ssstunnel.bt	3425
tcpconnect	1766	tcpconnect_filter.sh	4902

Figure 32. Count Comparisons of Low-Profile Attack Logs and bpftrace Logs (Source: Author, Billy Wilson)

7. Conclusions and Future Work

BPF probes, more specifically the bpftrace tool are recommended on HPC compute nodes for detecting malicious behavior. This recommendation is based on performance comparisons of single-node, two-node, and four-node HPL runs, both without and with BPF probes attached.

In future performance analyses, researchers could control additional factors that cause variation in HPL results. One example is disabling Turbo Mode on Intel processors. Researchers could also perform HPL runs above four nodes, as the results of the eight-node HPL runs in this study were inconclusive.

Moreover, future work should focus on the improvement of bpftrace scripts, especially as new features become available in future Linux distributions. Upcoming features include integer casting, the strncmp() function, and the array[] operator² (Gregg, 2020 April). Best case for an auditor is to have a single (or just a few logs to review). Would recommend noting that an audit reduction script could be done to further filter the numerous logs that are generated between needed audit review cycles.

This will be especially true for using the inet_sock_set_state() syscall. Although this syscall was traced for the sstunnel.bt script, some of its most valuable data in its arguments remained unusable due to the lack of an array[] operator. Once the operator is available, for example, the tcpconnect_filter.bt script can be entirely rewritten to use this syscall instead of a less stable dynamic kernel probe. The sstunnel.bt will also be able to log remote port forwarding in addition to local and dynamic port forwarding.

² Array operator functionality was merged into the master branch of bpftrace on 21 April 2019 and will hopefully be available in RHEL 8.2.

The bpftrace scripts in the analysis were limited to TCP. It would be valuable to write tracing scripts for other protocols such as ICMP, UDP, and InfiniBand if it can be done without producing excessive noise.

Security practitioners familiar with kernel code can use their knowledge to produce new bpftrace scripts catered to detect attack scenarios they see in the wild. As an example, Brendan Gregg used bpftrace to detect attempts to exploit a zero-day Docker vulnerability by tracing the uncommonly used `renameat2()` syscall (Gregg, 2020 January). Using the `signal()` function of bpftrace, an administrator today could write tracing scripts that proactively kill processes entering syscalls or functions known to be associated with bad behavior.

The latest bpftrace versions also support `cgroups`, making it easier to integrate tracing tools with HPC scheduler jobs. Using `cgroups`, a script can potentially filter processes associated with specific jobs dispatched by users.

Intuitively, with the accelerating digital transformation coupled with the growing complexity of ecosystems, it has become increasingly problematic for organizations to manage their collective digital footprint (i.e. attack surface). It's important to leverage and examine other emerging technologies for real-time traffic monitoring and the latest event-based analytic tools to detect and manage anomalies. One promising patch set is the Kernel Runtime Security Instrumentation (KRSI), which was developed by Google. This Linux Security Module (LSM) framework provides a mechanism for various security checks with much greater configuration auditing tools compared to Linux Auditing System (AuditD) (Corbet, 2019).

Within this analysis support, the BPF tracing tools' performance and detection results are accessible for classifying cyber threat actors within the HPC environments. The implementation of the BPF trace scripts successfully recorded the simulation attack activities without degrading the performance beyond 1% of compute nodes, exclusive of the discrepancies of the eight-node processors. With a few supplementary enhancements to the trace scripts, it could eliminate duplication and innocuous classification, while freeing up newly authorized trace scripts to exploit broader categories of attacks. These newly imposed developments will contribute to a high-fidelity detection and response solution built into the Linux kernel to protect against Linux kernel to protect both the security and performance of supercomputers.

8. References

- Chang, Y. T. S., Jin, H., & Bauer, J. (2016, November). Methodology and Application of HPC I/O Characterization with MPIProf and IOT. In *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*. IEEE, 2016.
- Corbet, J. (2012, January). Yet another new approach to seccomp. *LWN*. Retrieved 10 April 2020 from <https://lwn.net/Articles/475043/>
- Corbet, J. (2014, May). BPF: the universal in-kernel virtual machine. *LWN*. Retrieved 2 December 2019 from <https://lwn.net/Articles/599755/>
- Corbet, J. (2014, July). Extending extended BPF. *LWN*. Retrieved 2 December 2019 from <https://lwn.net/Articles/603983/>
- Corbet, J. (2019, December). KRSI — the other BPF security module. *LWN*. Retrieved 2 May 2020 from <https://lwn.net/Articles/808048/>
- Edge, J. (2011, July). Seccomp filters: No clear path. *LWN*. Retrieved 10 April 2020 from <https://lwn.net/Articles/450291/>
- Fleming, M. (2017, December). A thorough introduction to eBPF. *LWN*. Retrieved 2 December 2019 from <https://lwn.net/Articles/742082/>
- Gregg, B. (2017, January). BPF: Tracing and More. Presented at *linux conf au*, Hobart, Australia. Retrieved 28 March 2020 from <https://www.youtube.com/watch?v=JRFNIKUROPE>
- Gregg, B. (2018, September). *gethostlatency.bt* [Computer software]. Retrieved 28 March 2020 from <https://github.com/iovisor/bpftrace/blob/master/tools/gethostlatency.bt>
- Gregg, B. (2020, January). *BPF Performance Tools: Linux system and Application Observability*. United States: Addison-Wesley.
- Gregg, B., et al. (2020, April). *bpftrace Reference Guide*. *GitHub*. Retrieved 1 May 2020 from https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md
- Gregg, B., & Maestretti, A. (2017, February). Security Monitoring with eBPF. In *BSidesSF 2017*, San Francisco, CA. Retrieved 28 March 2020 from <https://www.youtube.com/watch?v=44nV6Mj11uw>
- Hamel, D. (2018, November). *tcpconnect.bt* [Compute software]. Retrieved 28 March 2020 from <https://github.com/iovisor/bpftrace/blob/master/tools/tcpconnect.bt>
- Kiepert, J. (2013, May). Creating a Raspberry Pi-based Beowulf Cluster. Department of Electrical and Computer Engineering, Boise State University, Boise, ID.
- McCanne, S., & Jacobson, V. (1992, December). The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *1993 Winter USENIX Conference*. San Diego, 1993. Retrieved 10 April 2020 from <https://www.tcpdump.org/papers/bpf-usenix93.pdf>
- National Institute for Standards and Technology. (2016). HPC Security Best Practices: Strengths and Weaknesses. In *NSCI: High-Performance Computing Security Workshop*, Gaithersburg, MD.
- Starovoitov, A. (2014, March). *net: filter: rework/optimize internal BPF interpreter's instruction set*. In *kernel/git/torvalds/linux.git* [software]. Retrieved 10 April 2020 from <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dff551b8>

Starovoitov, A. (2014, May). Tracing: accelerate tracing filters with BPF. In *net-next* [mailing list]. Retrieved 10 April 2020 from <https://lwn.net/Articles/598545/>

TOP500. (2019, November). TOP500 List - November 2019. Retrieved 21 March 2020 from <https://www.top500.org/list/2019/11/>

XSEDE. (n.d.). XSEDE Federation. Retrieved March 21, 2020, from <https://www.xsede.org/web/xsede-old/xsede-federation>

Appendix A: Low-Profile Attack Script

```

low_profile.sh
#!/usr/bin/bash

MIN_SLEEP=1
MAX_SLEEP=15
NONROOT_USER=someuser
DMZ_IP=10.1.2.3
URLPOOL_BGN=$(awk '$1 == "###URLPOOL_BGN###" { print NR + 1}' $0)
URLPOOL_END=$(awk '$1 == "###URLPOOL_END###" { print NR - 1}' $0)

dns_request() {

    local urlpool_line=$(shuf -i $URLPOOL_BGN-$URLPOOL_END -n 1)
    local dns_query=$(awk "NR == $urlpool_line" $0)
    local choice=$(shuf -i 0-4 -n 1)

    case $choice in
        0)
            date +"%Y-%m-%d %H:%M:%S type=dns comm=wget url=$dns_query"
            sudo -u $NONROOT_USER wget -T1 --tries 1 -O /dev/null $dns_query
        >/dev/null 2>&1
            ;;
        1)
            date +"%Y-%m-%d %H:%M:%S type=dns comm=curl url=$dns_query"
            sudo -u $NONROOT_USER curl -m1 $dns_query >/dev/null 2>&1
            ;;
        2)
            date +"%Y-%m-%d %H:%M:%S type=dns comm=dig url=$dns_query"
            sudo -u $NONROOT_USER dig +tries=1 +time=1 $dns_query >/dev/null 2>&1
            ;;
        3)
            date +"%Y-%m-%d %H:%M:%S type=dns comm=host url=$dns_query"
            sudo -u $NONROOT_USER host -W1 $dns_query >/dev/null 2>&1
            ;;
        4)
            date +"%Y-%m-%d %H:%M:%S type=dns comm=python url=$dns_query"
            sudo -u $NONROOT_USER python -c "import socket;
            socket.setdefaulttimeout(1); socket.gethostbyname('$dns_query')" >/dev/null
            2>&1
            ;;
    esac
}

run_sudo() {
    # This test depends on the $NONROOT_USER being
    # able to sudo without a password

    date +"%Y-%m-%d %H:%M:%S type=sudo comm=sudo user=$NONROOT_USER"
    sudo -u $NONROOT_USER sudo id >/dev/null 2>&1
}

```

```
tcp_connect_attempt() {

    local o1=192
    local o2=168
    local o3=$(shuf -i 0-255 -n 1)
    local o4=$(shuf -i 0-255 -n 1)
    local port=$(shuf -i 1-61000 -n 1)

    date +"%Y-%m-%d %H:%M:%S type=tcp_connect comm=bash ip=$o1.$o2.$o3.$o4
port=$port"
    timeout 1 sudo -u $NONROOT_USER bash -c "echo
>/dev/tcp/$o1.$o2.$o3.$o4/$port"

}

ssh_proxy() {

    local choice=$(shuf -i 0-1 -n 1)
    local port=$(shuf -i 10000-60000 -n 1)

    case $choice in
    0)
        local port_fwd_opt="-L $port:ubuntu.com:443"
        ;;
    1)
        local port_fwd_opt="-D $port"
        ;;
    esac

    date +"%Y-%m-%d %H:%M:%S type=ssh_proxy comm=ssh
port_fwd_options=$port_fwd_opt"
    sudo -u $NONROOT_USER ssh -n -o ConnectTimeout=5 -o ConnectionAttempts=1
$port_fwd_opt $DMZ_IP 'id' >/dev/null 2>&1

}

while :
do

    interval=$(shuf -i $MIN_SLEEP-$MAX_SLEEP -n 1)
    sleep $interval || exit 1

    choice=$(shuf -i 0-3 -n 1)
    case $choice in
    0)
        dns_request
        ;;
    1)
        run_sudo
        ;;
    2)
        tcp_connect_attempt
        ;;
    3)

```

```
ssh_proxy
;;
esac

done

exit 0

# This script selects a url from the list below.
# You can add urls, but don't change the tags

###URLPOOL_BGN###
poolin.com
f2pool.com
btc.com
antpool.com
viabtc.com
1thash.top
slushpool.com
btc.top
bitfury.com
minexmr.com
nanopool.org
prohashing.com
reuters.com
baidu.com
google.com
reddit.com
###URLPOOL_END###
```


Appendix B: HPL Results

Nodes	Non-BPF Run			BPF Run			Comparison of Non-BPF and BPF Run Performance
	Seconds	GFlops	Pct Max	Seconds	GFlops	Pct Max	
1	924.45	678.75	84.17%	928.95	675.47	83.76%	-0.41%
2	927.19	676.75	83.92%	928.91	675.49	83.77%	-0.16%
3	921.79	680.71	84.41%	922.81	679.96	84.32%	-0.09%
4	956.43	656.06	81.36%	960.79	653.09	80.99%	-0.37%
5	889.83	705.16	87.45%	890.34	704.76	87.40%	-0.05%
6	911.6	688.32	85.36%	911.69	688.26	85.35%	-0.01%
7	901.95	695.69	86.27%	904.9	693.42	85.99%	-0.28%
8	897.25	699.33	86.72%	897.05	699.49	86.74%	0.02%
[1-2]	1357.69	1347.40	83.54%	1355.80	1349.30	83.66%	0.12%
[3-4]	1366.23	1339.00	83.02%	1355.58	1349.50	83.67%	0.65%
[5-6]	1335.42	1369.90	84.94%	1330.99	1374.00	85.19%	0.25%
[7-8]	1317.00	1389.00	86.12%	1315.13	1391.00	86.25%	0.12%
[1-4]	2006.03	2658.70	82.42%	2003.93	2661.50	82.51%	0.09%
[5-8]	1902.89	2802.80	86.89%	1900.05	2807.00	87.02%	0.13%
[1-8]	2735.62	5349.70	82.93%	NA	NA	NA	NA
[1-8]	2791.96	5241.80	81.25%	NA	NA	NA	NA
[1-8]	NA	NA	NA	2790.09	5336.10	82.71%	NA
[1-8]	NA	NA	NA	2742.62	5245.30	81.31%	NA