Drilling Network Stacks with packetdrill

NEAL CARDWELL AND BARATH RAGHAVAN

Neal Cardwell received an M.S. in Computer Science from the University of Washington, with research focused on TCP and Web performance. He joined Google in 2002. Since then he has worked on networking software for google.com, the Googlebot web crawler, the network stack in the Linux kernel, and TCP performance and testing. neal@google.com

Barath Raghavan received a Ph.D. in Computer Science from UC San Diego and a B.S. from UC Berkeley. He joined Google in 2012 and was previously a Senior Researcher at ICSI in Berkeley, CA. His work has focused on network protocol design, applied cryptography, and sustainable computing. barath@google.com

Testing and troubleshooting network protocols and stacks can be painstaking. To ease this process, our team built packetdrill, a tool that lets you write precise scripts to test entire network stacks, from the system call layer down to the NIC hardware. packetdrill scripts use a familiar syntax and run in seconds, making them easy to use during development, debugging, and regression testing, and for learning and investigation.

Have you ever had the experience of staring at a long network trace, trying to figure out what on earth went wrong? When a network protocol is not working right, how might you find the problem and fix it? Although tools like tcpdump allow us to peek under the hood, and tools like netperf help measure networks end-to-end, reproducing behavior is still hard, and knowing when an issue has been fixed is even harder.

These are the exact problems that our team used to encounter on a regular basis during kernel network stack development. Here we describe packetdrill, which we built to enable scriptable network stack testing. packetdrill allows a user to specify a sequence of interactions with the network stack in a short script and then execute the script to verify the network stack's behavior.

packetdrill has a range of applications that we have been using it for on a daily basis:

- Regression testing a network stack: we have a suite of hundreds of packetdrill scripts that are run by all developers on our team before submitting a patch for review.
- Test-driven development of network protocols: we have developed several new features for Linux TCP using packetdrill.
- Reproduction of bugs seen in production network traces: we have used packetdrill to isolate hard-to-reproduce bugs seen in complex real traces.

We also believe that packetdrill can have significant value for

- self-directed learning of a network protocol, by writing scripts to elicit various behaviors from the network protocol in question;
- as a tool for teaching about network protocols in a university setting; and
- with minor extensions, scriptable testing of network applications that live above core network protocols.

packetdrill currently enables the user to test the correctness, performance, security, and general behavior of core network protocols—TCP, UDP, and ICMP—running on IPv4 and IPv6, and runs on Linux, FreeBSD, NetBSD, and OpenBSD. The tool is primarily for black-box testing, though it provides some support for examining internal network protocol state when supported by the OS.

packetdrill is released under version 2 of the GNU Public License (just like the Linux kernel), and we encourage patches, which you can send to the packetdrill email list (packetdrill@googlegroups.com), to extend the tool. For example, adding support for other IP-based protocols, such as DCCP or SCTP, would be straightforward, and we welcome patches to support these and other protocols.

The packetdrill Scripting Language

The packetdrill scripting language provides all the basic building blocks needed to set up a detailed, reproducible scenario for black-box testing of a network stack. The tool supports four types of statements: packets, system calls, shell commands, and Python scripts. Each statement is timestamped and is executed by the interpreter in real time, verifying that events proceed as the script expects. We discuss each type of statement in turn.

Packets

Arguably the most essential building block of any networking scenario is the packet. packetdrill allows the user to specify both inbound packets to inject into the system under test and outbound packets to expect the system to send. To keep the tests succinct and easy to both write and read, we use a syntax like that of tcpdump, which is familiar to most developers and system administrators who troubleshoot networking issues on UNIX systems. Modeled after UNIX shell input/output redirection operators, < denotes an input packet to construct and inject and > denotes an output packet to sniff and verify.

Here's an example of a TCP SYN packet, which packetdrill creates and injects into the network stack under test 100 ms after the start of the test:

```
0.100 < S 0:0(0) win 32792 <mss 1000,nop,nop,sackOK,nop,wscale 6>
```

Here's an example of an outbound UDP packet expected to be sent immediately after a prior event (denoted by +0), which packetdrill sniffs for and then verifies for matching specification (e.g., length, headers, etc.):

```
+0 > udp (1472)
```

System Calls

System calls are the other essential building block of a black-box network stack test scenario, since they express the application's intent and the work the kernel is supposed to perform. To specify a system call in packetdrill, the user only needs to provide the call's salient inputs, the duration for which the call is expected to block (if at all), and the expected outputs. The syntax mirrors that of strace, which we chose because it is familiar to most Linux users and is clear to any C programmer. In addition, in most cases it provides a quick one-line summary of both the inputs and outputs of a system call.

Here's an example of a bind() system call invocation in packetdrill notation:

```
+0 bind(3, ..., ...) = 0
```

In this example, 3 denotes the file descriptor number to pass in, and the = 0 denotes the expected return value (i.e., the user expects the system call to succeed). The ellipsis (...) here in place of the traditional addr and addrlen parameters is not to simplify the presentation in this article; rather, packetdrill supports this notation, again borrowed from strace, to allow scripts to omit irrelevant details. Under the hood, packetdrill fills in a sockaddr for bind and connect using an IP address and port number from command line options (with defaults for those options chosen to be appropriate for the address family involved—e.g., RFC 1918 private IPv4 address spaces). Hiding these details simplifies scripts and makes them quicker and easier to write and read. Just as important, it allows most scripts to be run without modification using IPv4, IPv6, or dual-mode (AF_INET6 socket with IPv4 traffic), depending on the command line arguments to packetdrill.

Shell Commands

packetdrill also allows scripts to specify arbitrary shell command sequences to execute, typically to configure the machine under test (e.g., with sysctl) or to assess the state of the machine (e.g., with netstat or ss). packetdrill implements this, as you would imagine, using a simple invocation of the C library's system() call. To enclose the commands, packetdrill borrows the backtick syntax used in shells and Perl.

Here's a typical example, which disables TCP timestamps in order to test TCP behavior without them:

```
+0 `sysctl -q net.ipv4.tcp_timestamps=0`
```

Python Commands

Finally, packetdrill allows inline Python code snippets to print information and to make assertions about the internal state of a TCP socket using the TCP_INFO getsockopt() option supported by Linux and FreeBSD. Users can enclose such snippets between %{ and }% tokens, a nod to lex/flex and yacc/bison syntax for embedding inline C snippets.

The following Linux-based example asserts that the sender's congestion window is 10 packets:

```
+0 %{ assert tcpi_snd_cwnd == 10 }%
```

In this example, under the hood packetdrill will make a TCP_INFO getsockopt() call for the socket under test and then stash the output tcp_info struct in memory. Then, when the test finishes execution, packetdrill emits a Python script encoding the contents of the tcp_info struct, followed by the Python code snippet that can print or make assertions about any interesting values.

An Example packetdrill Script

Next we give a short example. Suppose that you want to verify that your TCP stack correctly validates incoming TCP RST packets (see RFC 5961, Improving TCP's Robustness to Blind In-Window Attacks). Listing 1 shows a script (targeted at Linux)

that verifies that a TCP endpoint ignores a RST whose sequence number is just beyond the offered window.

```
// Create a listening TCP socket.
0   socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
+0  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
+0  bind(3, ..., ...) = 0
+0  listen(3, 1) = 0

// Establish a new connection.
+0  < S 0:0(0) win 32792 <mss 1000,sackOK,nop,nop,nop,wscale 7>
+0  > S. 0:0(0) ack 1 win 29200 <mss
      1460,nop,nop,sackOK,nop,wscale 6>
+.1 < . 1:1(0) ack 1 win 257
+0  accept(3, ..., ...) = 4

// sequence number out of window!
+.010 < R. 29202:29202(0) ack 1 win 257
// verify that the connection is OK
+.010  write(4, ..., 1000) = 1000
+0  > P. 1:1001(1000) ack 1
```

Listing 1: Validating handling of out-of-window RSTs

packetdrill's Design

Execution Model

packetdrill parses an entire test script, and then executes each timestamped line in real time—at the pace described by the timestamps—to replay and verify the scenario. The packetdrill interpreter has one thread for the main flow of events and another for executing any system calls that the script expects to block (e.g., poll()).

For convenience, scripts use an abstracted notation for packets. Internally, packetdrill models aspects of TCP and UDP behavior; to do this, packetdrill maintains mappings to translate between the values in the script and those in the live packet. The translation includes IP, UDP, and TCP header fields, including TCP options such as SACK and timestamps. Thus we track each socket and its IP addresses, port numbers, TCP sequence numbers, and TCP timestamps.

Local and Remote Testing

packetdrill enables two modes of testing: local mode, using a TUN virtual network device, or remote mode, using a physical NIC.

In local mode, packetdrill uses a single machine and a TUN virtual network device as a source and sink for packets. This tests the system call, sockets, TCP, and IP layers, and is easier to use because there is less timing variation, and users need not coordinate access to multiple machines.

In remote mode, users run two packetdrill processes, one of which is on a remote machine and speaks to the system under test over a LAN. This approach tests the full networking system: system calls, sockets, TCP, IP, software and hardware offload mechanisms, the NIC driver, NIC hardware, wire, and switch; however, due to the inherent variability in the many components under test, remote mode can result in larger timing variations, which can cause spurious test failures.

The packet plumbing is, naturally, a bit different in local and remote modes. To capture outgoing packets we use a packet socket (on Linux) or libpcap (on BSD-derived OSes). To inject packets locally we use a TUN device; to inject packets over the physical network in remote mode we again use a packet socket or libpcap. To consume test packets in local mode we use a TUN device; remotely, packets go over the physical network and the remote kernel drops them, because it has no interface with the test's remote IP address.

Local Mode

Local mode is the default, so to use it you need no special command line flags; you only need to provide the path of the script to execute:

```
./packetdrill foo.pkt
```

Remote Mode

To use remote mode, on the machine under test (the "client" machine), you must specify one command line option to enable remote mode (acting as a client) and then a second option to specify the IP address of the remote server machine to which the client packetdrill instance will connect. Only the client instance takes a packetdrill script argument, which can be the path of any ordinary packetdrill test script:

```
client# ./packetdrill --wire_client --wire_server_ip=<server_ip>
foo.pkt
```

On the remote machine, on the same layer 2 broadcast domain (e.g., same Ethernet switch), run the following to have a packetdrill process act as a "wire server" daemon to inject and sniff packets remotely on the wire:

```
server# ./packetdrill --wire_server
```

How does this work? First, the client instance connects to the server (using TCP), and sends the command line options and the contents of the script file. Then the two packetdrill instances work in concert to execute the script and test the client machine's network stack.

Model	Syntax Example	Description
Absolute	0.75	The specific time at which an event should occur.
Relative	+0.2	The interval after the last event at which an event should occur.
Wildcard	*	Allows an event to occur at any time.
Range	0.750~0.900	The absolute time range in which the event should occur.
Relative Range	+0.1~+0.2	The relative time range after the last event in which the event should occur.
Loose	--tolerance_usecs=800	Allows all events to happen within a range (from the command line).
Blocking	0.750...0.900	Specifies a blocking system call that starts/returns at the given times.

Table 1: Timing models supported by packetdrill

Timing Models

Because many protocols are sensitive to timing, we added support for significant timing flexibility in scripts. Each statement has a timestamp, enforced by packetdrill: if an event does not occur at the specified time, packetdrill flags an error and reports the actual time. Table 1 shows the packetdrill timing models.

Protocol Features

IPv4 and IPv6

packetdrill supports IPv4, IPv6, and dual-stack modes. The user specifies which mode to use when executing a test by using the `--ip_version` command line flag: AF_INET sockets with IPv4 traffic (`--ip_version=ipv4`), AF_INET6 sockets with IPv6 traffic (`--ip_version=ipv6`), and AF_INET6 sockets with IPv4 traffic (`--ip_version=ipv4-mapped-ipv6`).

To enable running the same script unmodified in any of the three modes, scripts omit IP-version-specific aspects of packets and system calls. For example, scripts do not specify the local and remote IP addresses of packets inside the script itself. Likewise, scripts do not specify a domain (AF_INET or AF_INET6) in a socket() call, nor do they specify the address and address length in a bind() call. As a result, getting a local test originally used for AF_INET sockets and IPv4 to work in other addressing modes is easy.

To run the test using AF_INET6 sockets with IPv4 traffic, use:

```
./packetdrill --ip_version=ipv4-mapped-ipv6 foo.pkt
```

To run the test using AF_INET6 sockets with IPv6 traffic, you'll need to specify both `--ip_version` and an MTU that is 20 bytes larger than the typical 1500-byte MTU, to accommodate the IPv6 header, which is 20 bytes larger than the IPv4 header:

```
./packetdrill --ip_version=ipv6 --mtu=1520 foo.pkt
```

With these small adjustments to the packetdrill command line, you can test all three addressing modes with a single script, with no extra development work.

Note that to get FreeBSD and NetBSD to allow using ipv4-mapped-ipv6 mode you must first tell the kernel you want to enable this mode of operation with:

```
sysctl -w net.inet6.ip6.v6only = 0
```

Also note that OpenBSD does not support ipv4-mapped-ipv6 mode because it explicitly disallows AF_INET6 sockets from handling IPv4 traffic.

Path MTU Discovery

packetdrill allows testing of Path MTU Discovery, which most TCP senders use to dynamically find an Internet path's maximum transmission unit (MTU), the biggest packet size that can safely traverse the path without suffering a performance hit due to IP-layer fragmentation and reassembly. Path MTU Discovery is described in RFC 1191 for IPv4 and RFC 1981 for IPv6. The basic idea is that senders mark the "Don't Fragment" (DF) bit in all outgoing IP headers. If a router along the path sees that it needs to fragment the packet but the DF bit is set, then the router sends an ICMP message saying "unreachable - fragmentation needed and DF set," with the MTU that the sender should use. When the sender receives this ICMP message, it retransmits any outstanding data and uses smaller packets in the future.

Listing 2 shows a simple Path MTU scenario (this script passes on Linux):

```
// Send a data segment.
+0 write(4, ..., 1460) = 1460
+0 > P. 1:1461(1460) ack 1

// ICMP says that segment was too big.
+0.100 < [1:1461(1460)] icmp unreachable frag_needed mtu 1200

// TCP retransmits with smaller packet size.
+0 > . 1:1161(1160) ack 1
+0 > P. 1161:1461(300) ack 1
```

Listing 2: TCP Path MTU Discovery example

Explicit Congestion Notification

packetdrill supports Explicit Congestion Notification, or ECN (see RFC 3168), a standard protocol that allows routers to explicitly signal to Internet transports (typically TCP) that there is congestion in the network by setting bits in the IP header. The ECN approach has several advantages over the traditional congestion signaling mechanism of dropping packets, but it is not yet widely deployed.

Any packet can have an ECN clause following the direction (< or >) field. Tests that do not care about ECN (and most tests do not) can simply omit the ECN clause. The supported ECN clauses allow tests to directly specify the injected or expected values of the two ECN bits; they are:

- [noecn] The IP ECN field is 00; sender transport (e.g., TCP) does not support ECN
- [ect1] The IP ECN field is 01, ECT(1), indicating "ECN-Capable Transport"
- [ect0] The IP ECN field is 10, ECT(0), indicating "ECN-Capable Transport"
- [ce] The IP ECN field is 11, set by a router to say "Congestion Experienced"

One interesting aspect of ECN is that ECN-capable senders (such as ECN-savvy TCP stacks) can set the ECN bits to either the ECT(0) or ECT(1) codepoints to indicate that they "speak ECN." This allows the sender and receiver to collaborate to detect whether some network element or receiver is corrupting or lying about the ECN bits, which would disrupt congestion signaling and potentially allow senders to grab an unfair share of bandwidth (see RFC 3540, Robust Explicit Congestion Notification (ECN) Signaling with Nonces). To cope with this potential variation, packetdrill also allows outgoing packets to use a fourth type of ECN clause, which specifies that an outgoing packet should have either the ECT(0) or ECT(1) codepoint:

- [ect01] The (outgoing) IP ECN field should be 10 or 01

Future Work

packetdrill can be used at present for testing not only fundamental network protocols that it supports natively (TCP, UDP, and ICMP on IPv4/IPv6) but also applications that use these protocols (e.g., a Web application that runs over TCP); however, because packetdrill has no knowledge of application-level datagrams, its ability to mimic, in script form, specific higher-layer protocols and application interactions is limited. We hope to make it easier for users to specify application-level payloads to be sent or received.

Also, packetdrill currently only supports testing a single connection at a time. We hope to extend it to support testing multiple concurrent connections. Furthermore, although packetdrill currently supports local (stand-alone) and on-the-wire (two-host) operations, it does not yet support multi-host operation or testing a remote machine that is not itself running packetdrill. These may be useful in some cases, and they should be straightforward to add to the current framework.

We welcome patches from the community, both for bug fixes and new features.

References

[1] Neal Cardwell, et al, "packetdrill: Scriptable Network Stack Testing, from Sockets to Packets," USENIX ATC 2013: https://www.usenix.org/conference/atc13/packetdrill-scriptable-network-stack-testing-sockets-packets.

[2] packetdrill open source project home and git repository: https://code.google.com/p/packetdrill/.

[3] packetdrill email list, for questions, discussion, and patches: http://groups.google.com/group/packetdrill.