packetdrill: Scriptable Network Stack Testing, from Sockets to Packets

Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan,
Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert
Google

Abstract

Testing today's increasingly complex network protocol implementations can be a painstaking process. To help meet this challenge, we developed packetdrill, a portable, open-source scripting tool that enables testing the correctness and performance of entire TCP/UDP/IP network stack implementations, from the system call layer to the hardware network interface, for both IPv4 and IPv6. We describe the design and implementation of the tool, and our experiences using it to execute 657 test cases. The tool was instrumental in our development of three new features for Linux TCP—Early Retransmit, Fast Open, and Loss Probes—and allowed us to find and fix 10 bugs in Linux. Our team uses packetdrill in all phases of the development process for the kernel used in one of the world's largest Linux installations.

1 Introduction

Despite their importance in modern computer systems, network protocols often undergo only ad hoc testing before their deployment, and thus they often disappoint us. In large part this is due to their complexity. For example, the TCP roadmap RFC [19] from 2006 lists 32 RFCs. Linux implements many of these, along with a few post-2006 Internet drafts, the sockets API, a dozen congestion control modules, SYN cookies, numerous software and hardware offload mechanisms, and socket buffer management. Furthermore, new algorithms have unforeseen interactions with other features, so testing has only become more daunting as TCP has evolved. In particular, we have made a number of changes to Linux TCP [14, 15, 17, 20–22, 30] and have faced significant difficulty in testing these features. The difficulties are exacerbated by the number of components interacting, including the application, kernel, driver, network interface, and network. We found we needed a testing tool for three reasons:

New feature development. Development testing of new TCP features has often relied either on testing patches on production machines or in emulated or simulated network scenarios. Both approaches are time-consuming. The former is risky and impossible to automate or reproduce; the latter is susceptible to unrealistic modeling.

Regression testing. While valuable for measuring overall performance, TCP regression testing with netperf, application load tests [16], or production workloads can fail to reveal significant functional bugs in congestion control, loss recovery, flow control, security, DoS hardening, and protocol state machines. Such approaches suffer from noise due to variations in site/network conditions or content, and a lack of precision and isolation; thus bugs in these areas can go unnoticed (e.g. the bugs discussed in Section 4.2 were only discovered with packetdrill tests).

Troubleshooting. Reproducing TCP bugs is often challenging, and can require developers to instrument a production kernel to collect clues and identify the culprit. But production changes risk regressions, and it can take many iterations to resolve the issue. Thus we need a tool to replay traces to reproduce problems on non-production machines.

To meet these challenges, we built packetdrill, a tool that enables developers to easily write precise, reproducible, automated test scripts for entire TCP/UDP/IP network stacks. We find that it meets our design goals:

Convenient. Developers can quickly learn the syntax of packetdrill and need not understand the internals of protocols or packetdrill itself. The syntax also makes it easy for the script writer to translate packet traces into test scripts. The tool runs in real time so tests often complete in under one second, enabling quick iteration.

Realistic. packetdrill works with packets and system calls, testing precise sequences of real events. packetdrill tests the exact kernel image used in production, running in real time on a physical machine. It

can run with real drivers and a physical network interface card (NIC), wire, and switch, or a TUN virtual NIC. It does not rely on virtual machines, user-mode Linux, emulated networks, or approximate models of TCP.

Reproducible. `packetdrill` can reliably reproduce test script timing with less than one spurious failure per 2500 test runs (see Section 4.4).

General. `packetdrill` allows a script to run in IPv4, IPv6, or IPv4-mapped IPv6 mode without modification. It runs on Linux, FreeBSD, OpenBSD, and NetBSD, and is portable across POSIX-compliant operating systems that support the `libpcap` packet capture/injection library. Since it is open source, it can be extended by protocol implementors to work with new algorithms, features, and packet formats, including TCP options.

We find `packetdrill` useful in feature development, regression testing, and production troubleshooting. During feature development, we use it to unit test implementations, thereby enabling test-driven development—we have found it vital for incrementally testing complex new TCP features on both the server and client side during development. Then we use it for easy regression testing. Finally, once code is in production, we use it to isolate and reproduce bugs. Throughout the process, `packetdrill` provides a succinct but precise language for discussing TCP scenarios in bug reports and email discussions.

In the rest of the paper, we discuss the design and implementation of `packetdrill`, our experiences using it, and related work.

2 Design

2.1 Scripting Language

`packetdrill` is entirely script-driven, to ease interactive use. `packetdrill` scripts use a language we designed to closely mirror two syntaxes familiar to networking engineers: `tcpdump` and `strace`. The language has four types of statements:

- Packets, using a `tcpdump`-like syntax, including TCP, UDP, and ICMP packets, and common TCP options: SACK, Timestamp, MSS, window scale, and Fast Open.
- System calls, using an `strace`-like syntax.
- Shell commands enclosed in `` `` `` backticks, which allow system configuration or assertions about network stack state using commands like `ss`.
- Python scripts enclosed in `%{}%` braces, which enable output or assertions about the `tcp_info` state that Linux and FreeBSD expose for TCP sockets.

2.2 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. For each system call line, `packetdrill` executes the system call and verifies that it returns the expected result. For each command line, `packetdrill` executes the shell command. For each incoming packet (denoted by a leading `<` on the line), `packetdrill` constructs a packet and injects it into the kernel. For each outgoing packet (denoted by a leading `>` on the line), `packetdrill` sniffs the next outgoing packet and verifies that the packet's timing and contents match the script.

Consider the example script in Figure 1, which shows a `packetdrill` script that tests TCP fast retransmit. This test passes as-is on Linux, FreeBSD, OpenBSD, and NetBSD, using a real NIC. As is typical, this script starts by setting up a socket (lines 1–4) and establishing a connection (lines 5–8). After writing data to a socket (line 9), the script expects the network stack under test to send a data packet (line 10) and then directs `packetdrill` to inject an acknowledgement (ACK) packet (line 11) that the stack will process. The script ultimately verifies that a fast retransmit occurs after three duplicate acknowledgements arrive.

2.3 Local and Remote Testing

`packetdrill` enables two modes of testing: *local* mode, using a TUN virtual network device, or in *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 since 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.

3 Implementation

`packetdrill` is a user-level application written entirely in C, adhering to Linux kernel code style to ease use in kernel testing environments. In this section we delve into the implementation of the tool.

```
0   socket(..., SOCK_STREAM, IPPROTO_TCP) = 3           // Create a socket.
+0  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0  // Avoid binding issues.
+0  bind(3, ..., ...) = 0                               // Bind the socket.
+0  listen(3, 1) = 0                                    // Start listening.

+0  < S 0:0(0) win 32792 <mss 1000,sackOK,nop,nop,nop,wscale 7>   // Inject a SYN.
+0  > S. 0:0(0) ack 1 <...>                                       // Expect a SYN/ACK.
+.1 < . 1:1(0) ack 1 win 257                                      // Inject an ACK.
+0  accept(3, ..., ...) = 4                                       // Accept connection.

+0  write(4, ..., 1000) = 1000   // Write 1 MSS of data.
+0  > P. 1:1001(1000) ack 1      // Expect it to be sent immediately.
+.1 < . 1:1(0) ack 1001 win 257  // Inject an ACK after 100ms.

+0  write(4, ..., 4000) = 4000   // Write 4 MSS of data.
+0  > .  1001:2001(1000) ack 1   // Expect immediate transmission.
+0  > .  2001:3001(1000) ack 1
+0  > .  3001:4001(1000) ack 1
+0  > P. 4001:5001(1000) ack 1

+.1 < . 1:1(0) ack 1001 win 257 <sack 2001:3001,nop,nop>  // Inject 3 ACKs with SACKs.
+0  < . 1:1(0) ack 1001 win 257 <sack 2001:4001,nop,nop>
+0  < . 1:1(0) ack 1001 win 257 <sack 2001:5001,nop,nop>

+0  > . 1001:2001(1000) ack 1    // Expect a fast retransmit.
+.1 < . 1:1(0) ack 6001 win 257  // Inject an ACK for all data.
```

Figure 1: A `packetdrill` script for TCP fast retransmit. Scripts use `...` to omit irrelevancies.

3.1 Components

3.1.1 Lexer and Parser

For generality and extensibility, we use `flex` and `bison` to generate `packetdrill`'s lexer and parser, respectively. The structure of the script language is fairly simple, and includes C/C++ style comments.

3.1.2 Interpreter

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()`).[1]

Packet events. For convenience, scripts use an abstracted notation for packets. Internally, `packetdrill` models aspects of TCP and UDP behavior; to do this, it 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.

For outbound packet events we start sniffing immediately, in order to detect any packets that go out earlier than the script specifies. When we sniff an outbound live packet we find the socket that sent it, and verify that the packet was sent at the expected time. Then we translate the live packet to its script equivalent and verify that the bits the kernel sent match what the script expected.

For inbound packet events we pause until the specified time, then translate the script values to their live equivalents so the network stack under test can process them, and then inject the packet into the kernel.

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 use `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, since it has no interface with the test's remote IP address.

In `packetdrill` scripts, several aspects of outgoing TCP packets are optional. This simplifies tests, allows them to focus on a single area of behavior, eases maintenance, and facilitates cross-platform testing by avoiding test failures due to irrelevant differences in protocol stack behavior over time or between different OSes. For example, scripts may omit the TCP receive window, or use a `<...>` notation for TCP options. If specified, they are checked; otherwise they are ignored. For example, the `<...>` on the SYN/ACK packet in Figure 1 ignores the only difference between the four OSes in this test.

System calls. For non-blocking system call events, we invoke the system call directly in the main thread. For

[1] Currently, for simplicity of both understanding and implementation, we support only one blocking system call at a time.

blocking calls, we enqueue the event on an event queue and signal the system call thread. The main thread then waits for the system call thread to block or finish the call.

When executing system calls we evaluate script symbolic expressions and translate to live equivalents to get inputs for the call. Then we invoke the system call; when it returns we verify that the actual output, including `errno`, matches the script's expected output.

Shell commands. `packetdrill` executes command strings with `system()`.

Python scripts. `packetdrill` runs Python snippets by recording the socket's `tcp_info` struct at the time of the script event, and then emitting Python code to export the data, followed by the Python snippet itself, for the Python interpreter to run after test execution completes.

3.2 Handling Variation

3.2.1 Network protocol features

`packetdrill` supports a wide array of protocol features. Developers can use the same script unmodified across IPv4, IPv6, and IPv4-mapped IPv6 modes by using command line flags to select the address mode and MTU size. Beyond IPv4, IPv6, TCP, and UDP, we support ECN and inbound ICMP (for path MTU discovery). It would be straightforward to add support for other IP-based protocols, such as DCCP or SCTP.

3.2.2 Machine configuration

We have found that most scripts share machine settings, and thus most scripts start by invoking a default shell script to configure the machine. Also, since script system calls do not specify aspects of the test machine's configuration, the interpreter substitutes these values in during test execution. For example, we select a default IP address that will be used for `bind` system calls based upon the choice of IPv4, IPv6, or IPv4-mapped IPv6.

3.2.3 Timing Models

Since many protocols are very 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.

3.2.4 Avoiding Spurious Failures

For over a year, we have used a `--tolerance_usecs` value of 4 ms, so a test will pass as long as events happen within 4 ms of the expected time. This allows the most common variation: a 1-ms deviation in RTT leads to a 3-ms deviation in retransmission timeout (RTO), initialized to $3 \cdot RTT$ per RFC 6298. We have found this to be a practical trade-off between precision and maintenance overhead, catching most significant timing bugs while usually allowing a full run of all `packetdrill` scenarios without a single spurious failure.

`packetdrill` also takes steps internally to reduce timing variation and spurious failures, including aligning the start of test execution at a fixed phase offset relative to the kernel scheduler tick, leveraging sleep wake-up events to obtain fresh tick values on "tick-less" Linux kernels lacking a regular scheduler tick, using a real-time scheduling priority, using `mlockall()` to attempt to pin its memory pages into RAM, precomputing data where possible, and automatically sending a TCP RST segment to all test connections at the end of a test to avoid interference from retransmissions.

4 Experiences and results

For over 18 months we have used `packetdrill` to test the Linux kernel used on Google production machines. Next we discuss how we've found it useful.

4.1 Features developed with `packetdrill`

Our team has used `packetdrill` to test the features that we have implemented in Linux and have published. We avoided pushing into production numerous bugs by using `packetdrill` during development to test TCP Early Retransmit [14], TCP Fast Open [30], TCP Loss Probe [20], and a complete rewrite of the Linux F-RTO implementation [15]; we also used it to test forward error correction for TCP [24]. The TCP features we developed before `packetdrill`, and thus for which we wrote `packetdrill` tests afterward, include increasing TCP's initial congestion window to ten packets [22], reducing TCP's initial retransmission timeout to 1 second [17], and Proportional Rate Reduction [21].

4.2 Linux bugs found with `packetdrill`

In the process of writing tests for the Linux TCP stack, our team found and fixed 10 bugs in the official version of Linux maintained by Linus Torvalds.

DSACK undo. Linux TCP can use duplicate selective acknowledgements, or DSACKs, to undo congestion window reductions. There was a bug where DSACKs were ignored if there were no outstanding unacknowledged packets at the time the sender receives the DSACK—this is actually the most common case [4]. Also, Linux was not allowing DSACK-based undo in some cases where ACK reordering occurred [5].

CUBIC and BIC RTO undo. CUBIC, the default TCP congestion control module for Linux, and the related BIC module had bugs preventing them from undoing a congestion window reduction that resulted from an RTO [6, 7]; RTOs are the most common form of loss recovery in web sites with short flows [21].

Model	Syntax	Description
Absolute	`0.750`	Specifies the specific time at which an event should occur.
Relative	`+0.2`	Specifies the interval after the last event at which an event should occur.
Wildcard	`*`	Allows an event to occur at any time.
Range	`0.750~0.9`	Requires the given event to occur within the time range.
Loose	`--tolerance_usecs=800`	Allows all events to happen within a range (from the command line).
Blocking	`0.750...0.9`	Specifies a blocking system call that starts/returns at the given times.

Table 1: Timing models supported in `packetdrill`.

TCP Fast Open server. We used `packetdrill` to find and fix several minor bugs in the TCP Fast Open server code: the RTT sample taken using the server's TCP SYN/ACK packet was incorrect [8, 9], TFO servers failed to process the TCP timestamp value on the incoming receiver ACK that completed the three-way handshake [10], and TFO servers failed to count retransmits that happened during the three-way handshake [11].

Receiver RTT estimation. We found and fixed a bug in which receiver-side RTT estimates were broken due to a path in which the code was directly comparing a raw RTT sample with one that had already been shifted into a fixed point representation [12].

Scheduler jiffies update. Jitter in `packetdrill` test RTT estimates hinted at a Linux kernel code path in which tick-less jiffies could be stale. Our audit of the jiffies code revealed such a bug, which we fixed [13].

4.3 Catching external behavior changes

`packetdrill` scripts brought to our team's attention external Linux kernel changes that were not bugs, but still had significant impacts in our environment, including timer slack [32] and recent fixes in packet size accounting [23]. For these changes we ended up adjusting our production kernel's behavior.

4.4 Test Suite

Coverage. Our team of nine developers has written 266 `packetdrill` scripts to test the Google production Linux kernel and 92 scripts to test `packetdrill` itself. Because `packetdrill` enables developers to run a given test script in IPv4, IPv6, or IPv4-mapped IPv6 modes, the number of total test case scenarios is even greater: 657. Table 2 summarizes the areas of TCP functionality covered by our `packetdrill` scripts.

Reproducibility. To quantify the reproducibility of our test results, we examined the spurious failure rate for two days of recent test runs on a 2.2GHz 64-bit multiprocessor PC running a recent Google production Linux kernel. We examined the most recent 54 complete runs of all 657 `packetdrill` test cases relevant for that release of the kernel, and found 14 test case failures, all of which were spurious. This implies an overall spurious test case failure rate of just under 0.0004, or 1 in 2500. Since fewer than a quarter of full test runs suffer from spurious failures, we find this to be an acceptable overhead on our kernel team. However, we continue to refine scripts to further reduce the spurious failure rate.

Feature	Description	Tests
Socket API	listen, connect, write, close, etc.	11
RFC 793	Core functionality	21
RFC 1122	Keep-alive	4
RFC 1191	Path MTU discovery	4
RFC 1323	Timestamps	1
RFC 2018	SACK (Selective Acknowledgement)	12
RFC 3168	Explicit Congestion Notification	3
RFC 3708	DSACK-based undo	10
RFC 5681	Congestion control	10
RFC 5827	Early retransmit	11
RFC 5682	F-RTO (Forward RTO-Recovery)	14
RFC 6298	Retransmission timer	13
RFC 6928	Initial congestion window	5
RFC 6937	Proportional rate reduction	10
IETF draft	Fast open	44
IETF draft	Loss probe	9
IETF draft	CUBIC congestion control	1
n/a	TSO (TCP segmentation offload)	3
n/a	Receive buffer autotuning	2
n/a	Linux `inet_diag` sockets	3
n/a	Miscellaneous	75
	Total test scripts	266

Table 2: Areas of TCP tested by `packetdrill` scripts.

Execution Time. `packetdrill` scripts execute quickly, so we run all `packetdrill` scripts before sending for review any commit that modifies the Google production TCP code. For the 54 test runs mentioned above, the total time to execute all 657 test cases was 25–26 minutes in all 54 test runs, an average of 2.4 seconds per test case.

5 Related Work

There are many tools to debug and test protocol implementations. RFC2398 [28] categorizes late-90s tools. The Packet Shell [27] seems to be the closest to `packetdrill` in design. It allowed scripts to send and receive packets to test a TCP peer's responses, but it was developed specifically for Solaris, is no longer available publicly, was more labor-intensive (e.g. it took 8

lines of `Tcl` commands to inject a single TCP SYN), and had no support for the sockets API, specifying packet arrival times, or handling timers. Orchestra [18] is a fault-injection library to check the conformance of TCP implementations to basic TCP RFCs. It places a layer below the X-kernel TCP stack and executes user-specified actions to delay, drop, reorder, and modify packets. Results require manual inspection, and the tests are not automated to check newer TCP stacks. While not developed for testing, TCPanaly [29] analyzes TCP traces to identify TCP implementations and diagnose RFC violations or performance issues. In `packetdrill` such domain knowledge is constructed through scripts; in TCPanaly its built directly into the software itself, which is harder to revise and expand.

The tools above were developed in the late 1990s, and to our knowledge none of them is being actively used to test modern TCP stacks. By contrast, IxANVL [1] is a modern commercial protocol conformance testing tool that covers core TCP RFCs and a few other networking protocols, but unlike `packetdrill` it can not be easily extended or scripted to troubleshoot bugs or test new changes, and is not open source.

Other research efforts test protocols by manually writing a model in a formal language, and then using automated tools to check for bugs [2, 3, 26, 31]. While these models are rigorous, their high maintenance cost is unsustainable, since they diverge from the rapidly-evolving code they try to model. Other tools automatically find bugs, but only within narrow classes, or in user-level code [25]. These approaches are complementary to ours.

6 Conclusion

`packetdrill` enables quick, precise, reproducible scripts for testing entire TCP/UDP/IP network stacks. We find `packetdrill` indispensable in verifying protocol correctness, performance, and security during development, regression testing, and troubleshooting. We have released `packetdrill` as open source in the hope that sharing it with the community will make the process of improving Internet protocols an easier one.

The source code and test scripts for `packetdrill` are available at: http://code.google.com/p/packetdrill/.

Acknowledgements

We would like to thank Bill Sommerfeld, Mahesh Bandewar, Chema Gonzalez, Laurent Chavey, Willem de Bruijn, Eric Dumazet, Abhijit Vaidya, Cosmos Nicolaou, and Michael F. Nowlan for their help and feedback.

References

[1] IxANVL. http://goo.gl/SV6ia.

[2] BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Proc. of SIGCOMM* (2005), ACM.

[3] BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proc. of ACM POPL* (2006), ACM.

[4] CARDWELL, N. *Linux commit 5628adf*. http://goo.gl/Fnj46.

[5] CARDWELL, N. *Linux commit e95ae2f*. http://goo.gl/uyRUp.

[6] CARDWELL, N. *Linux commit fc16dcd*. http://goo.gl/xv6xB.

[7] CARDWELL, N. *Linux commit 5a45f00*. http://goo.gl/cHYiw.

[8] CARDWELL, N. *Linux commit 0725398*. http://goo.gl/jWu0S.

[9] CARDWELL, N. *Linux commit 016818d*. http://goo.gl/axR97.

[10] CARDWELL, N. *Linux commit e69bebd*. http://goo.gl/Rh2J1.

[11] CARDWELL, N. *Linux commit 30099b2*. http://goo.gl/BKZWH.

[12] CARDWELL, N. *Linux commit 18a223e*. http://goo.gl/BL4O5.

[13] CARDWELL, N. *Linux commit 6f10392*. http://goo.gl/IFQ4D.

[14] CHENG, Y. *Linux commit eed530b*. http://goo.gl/MPmF0.

[15] CHENG, Y. *Linux commit e33099f*. http://goo.gl/hhIfU.

[16] CHENG, Y., HÖLZLE, U., CARDWELL, N., SAVAGE, S., AND VOELKER, G. Monkey see, monkey do: A tool for TCP tracing and replaying. In *Proc. of USENIX ATC* (2004).

[17] CHU, J. *Linux commit 9ad7c04*. http://goo.gl/gxiFT.

[18] DAWSON, S., JAHANIAN, F., AND MITTON, T. Experiments on six commercial TCP implementations using a software fault injection tool. *Software Practice and Experience 27*, 12 (1997), 1385–1410.

[19] DUKE, M., BRADEN, R., EDDY, W., AND BLANTON, E. A Roadmap for Transmission Control Protocol (TCP) Specification Documents, September 2006. RFC 4614.

[20] DUKKIPATI, N., CARDWELL, N., CHENG, Y., AND MATHIS, M. Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses, Feb. 2013. IETF Draft, draft-dukkipati-tcpm-tcp-loss-probe-01.

[21] DUKKIPATI, N., MATHIS, M., CHENG, Y., AND GHOBADI, M. Proportional rate reduction for TCP. In *Proc. of IMC* (2011).

[22] DUKKIPATI, N., REFICE, T., CHENG, Y., CHU, J., HERBERT, T., AGARWAL, A., JAIN, A., AND SUTIN, N. An Argument for Increasing TCP's Initial Congestion Window. *ACM Comput. Commun. Rev. 40* (2010).

[23] DUMAZET, E. *Linux commit 87fb4b7*. http://goo.gl/MgRWi.

[24] FLACH, T., DUKKIPATI, N., TERZIS, A., RAGHAVAN, B., CARDWELL, N., CHENG, Y., JAIN, A., HAO, S., KATZ-BASSETT, E., AND GOVINDAN, R. Reducing Web Latency: the Virtue of Gentle Aggression. In *SIGCOMM* (2013).

[25] KOTHARI, N., MAHAJAN, R., MILLSTEIN, T. D., GOVINDAN, R., AND MUSUVATHI, M. Finding protocol manipulation attacks. In *SIGCOMM* (2011), pp. 26–37.

[26] MUSUVATHI, M., ENGLER, D., ET AL. Model checking large network protocol implementations. In *Proc. of NSDI* (2004).

[27] PARKER, S., AND SCHMECHEL, C. The packet shell protocol testing tool. http://goo.gl/CS4kf.

[28] PARKER, S., AND SCHMECHEL, C. RFC2398: Some testing tools for TCP implementors, August 1998.

[29] PAXSON, V. Automated packet trace analysis of TCP implementations. In *Proc. of ACM SIGCOMM* (1997), ACM.

[30] RADHAKRISHNAN, S., CHENG, Y., CHU, J., JAIN, A., AND RAGHAVAN, B. TCP Fast Open. In *Proc. of CoNEXT* (2011).

[31] SMITH, M., AND RAMAKRISHNAN, K. Formal specification and verification of safety and performance of TCP selective acknowledgment. *IEEE/ACM ToN 10*, 2 (2002).

[32] VAN DE VEN, A. *Linux commit 3bbb9ec*. http://goo.gl/w18r6.