Testing NetBSD Automagically

Martin Husemann <martin@NetBSD.org>

Abstract

A huge effort is made by the NetBSD project to test “current” – the bleeding edge version – systematically. While the setup is still developing, diverse, and somewhat chaotic, this has already proven to be an extremely valuable early notice alarm system during times of massive churn all over the tree, as well when hunting down concrete bugs already introduced by apparently innocent changes.

The introduction of tests changes developers mind and approaches to a problem. At the same time it splits the developer community – into the ones that believe in bugs in code and tests that find them, and the ones that believe in bugs in test cases (or the test environment). Unfortunately there is no bug free code nor are there bug free test cases, so it is always necessary to look into failure details before blaming the one or the other.

Testing a full operating system, covering kernel and user land (all of it!) is practically impossible. This paper will try to show the stony way, the problems met on the social side, the ongoing quest to improve the testing framework. It will show examples of the quickly increasing number of test cases, and discuss in detail and categorize examples from the various types of failures/bugs encountered and solved (or not yet solved).

The author is running the NetBSD/sparc64 instance of the regular tests and between two runs busy keeping the number of failures down.

What are we testing?

The automatic test runs (see http://releng.netbsd.org/test-results.html) cover about half of the cpu families supported by NetBSD currently. They are run both on real hardware and on emulators. On i386, amd64 and sparc (tested under QEMU) we test from scratch – each test run starts with booting the NetBSD installer, installing on a fresh (virtual) disk, rebooting into the installed system and running the full test suite there. On other architectures we test a prebuild system on real hardware, running only the test suite.
Currently the tests are only in parts run centralized on machines owned by The NetBSD Foundation, others are done by individual developers, on their local machines. The testsuite covers a wide variety of random things, many of them converted from example code found in bug reports. Creating a systematic test suite covering all of an operating system including user land tools is a herculean task - we are slowly improving, but not getting anywhere near full coverage anytime soon.
There will be more details about test cases in the chapter about bugs we found below.

The building blocks

An overview of the building blocks used in our test setups is given:

- ATF – the Automatic Testing Framework
  developed by Julio Merino as a Google Summer of Code project
- RUMP – term abused as an umbrella here for multiple technologies developed mostly by Antti Kantee to run kernel code in user land
- QEMU – the well-known open source machine emulator
- Anita, an Automated NetBSD Installation and Test Application
  A python script interacting with QEMU to drive a full NetBSD installation from scratch, reboot into the fresh install and run the test there. Writen by Andreas Gustafsson

Note that emulators/virtualization (like QEMU used in the anita setups) are not a necessary or central part of the test setup, but only a very handy way to automate a full scriptable test run. As we will see below, results from emulators are always suspicious and need to be double checked on real hardware.

What is ATF?

ATF is the Automated Testing Framework (http://www.netbsd.org/~jmmv/atf/) developed initially by Julio Merino during Summer of Code 2007. The main features of ATF that are used in the NetBSD automatic test setups are:

- Easily scriptable, unattended automatic runs
- Easily extractable reports in various formats (xml used internally, plain text as user feedback during the test run, html as overall result)
- Test programs can be pre-(cross)-compiled and later run on the target hardware

We will look at test program example code later in more detail.

There are some different features that are more important to (test program) developers:

- A test program can be coded in the language most suitable for it, ATF comes with various bindings, for example /bin/sh, C and C++. Most test programs in the NetBSD test suite are coded in C currently, with a few ones (mostly testing userland stuff) in /bin/sh.
- A test program can be debugged outside of ATF “easily”. For easy cases, ATF automatically invokes GDB and gets a back trace if a test program crashes unexpectedly.

- You do not have to run the full test suite. The tests are organized in a hierarchical directory structure; running all tests below a random directory is easy.

Furthermore, testing is pretty fast – but I fear (and at the same time hope) that additional tests will change this soon. Currently a full ATF run of the NetBSD testsuite on the sparc64 setup takes about 30 minutes.

One of the main "uncommon" ideas behind ATF and the NetBSD testsuite is that binary only users can easily run it. Most other testing "frameworks" treat testing as a post-build step. NetBSD runs on a variety of hardware not easily available in a test lab, so we offer a "test.tgz" set during standard installation allowing all users to easily install and run the tests on their own machines. No further tools required, everything else (besides the test binaries) comes with NetBSD base.

The ATF site has a few comparisons to other testing enviroments (http://www.netbsd.org/~jmmv/atf/about.html) and there is an intended successor to ATF called kyua (pronounced: Q.A., see http://code.google.com/p/kyua/).

What is RUMP?

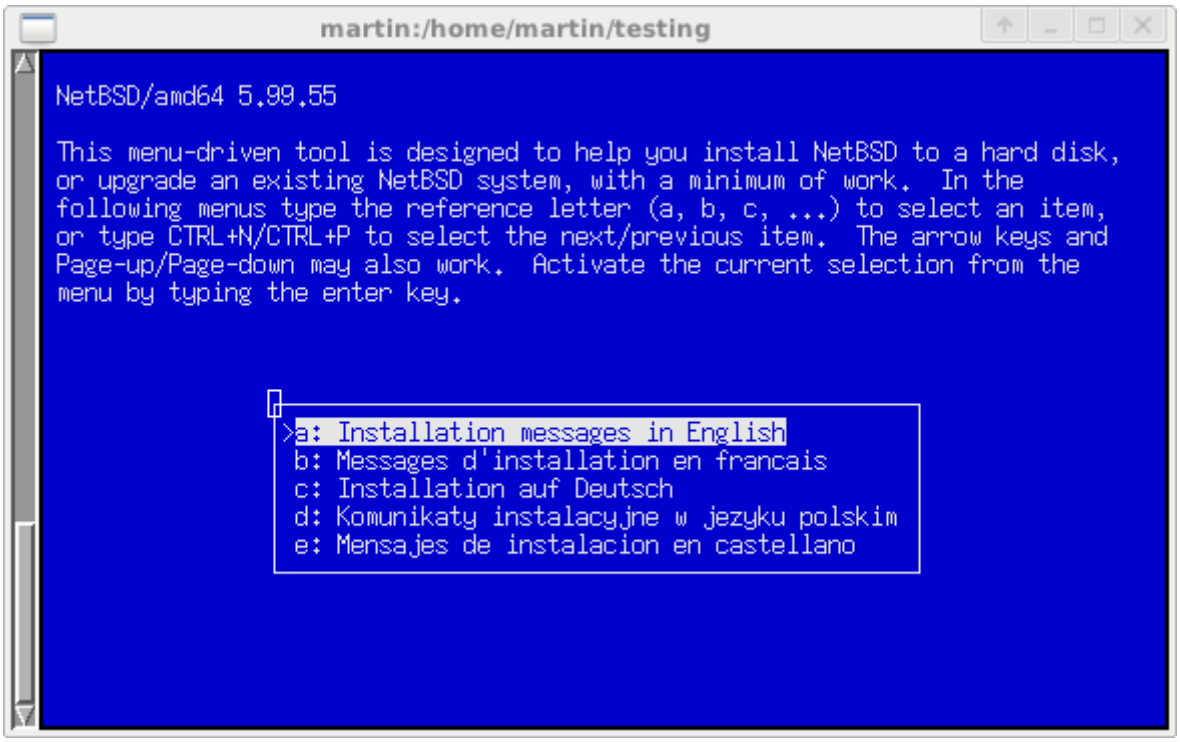
In this paper I use RUMP as an umbrella term for various technologies used to run kernel code in userland, mostly developed by NetBSD developer Antti Kantee. The basic idea is to compile and link standard kernel code against a set of headers and libraries that arrange for standard syscalls to end up as calls to the rump kernel (instead of the host kernel), which the rump kernel (in the end) services via the host kernel, or deals with them itself. This is very different to what, for example, Usermode Linux does: we are dealing with kernel sources, but not stock userland binaries. Rump also does not target to be a full grown system running on a host as some kind of virtualization; only selected slices of the kernel are loaded into the rump server process. For example when dealing with networking, we will require librump, librumpuser and librumpnet in our rump client. If used as a development tool to create a USB driver, our rump client would require librumpdev instead of librumpnet. Rump can be used to easily create userland programs understanding disk images (using librumpvfs).

A special case is librumphijack, which is preloaded via ld.elf_so's LD_PRELOAD mechanism to hijack stock system calls from "native" binaries and redirect them to a rump server. This way you can run, for example, firefox using an experimental TCP/IP stack in a local rump server. Contrary to what I stated before, this could be driven to the limit, creating something close to Usermode Linux, however, it would be pretty inconvenient in the current form.

RUMP is used in various test programs in the NetBSD testsuite, to avoid crashing the host kernel, or "virtualizing" only small parts of the kernel to keep the test case simple, or simply because it is very easy to operate on disk images freshly created for the test case (avoiding vnd configuration trouble and bookkeeping). We will see an example of such a test case later.

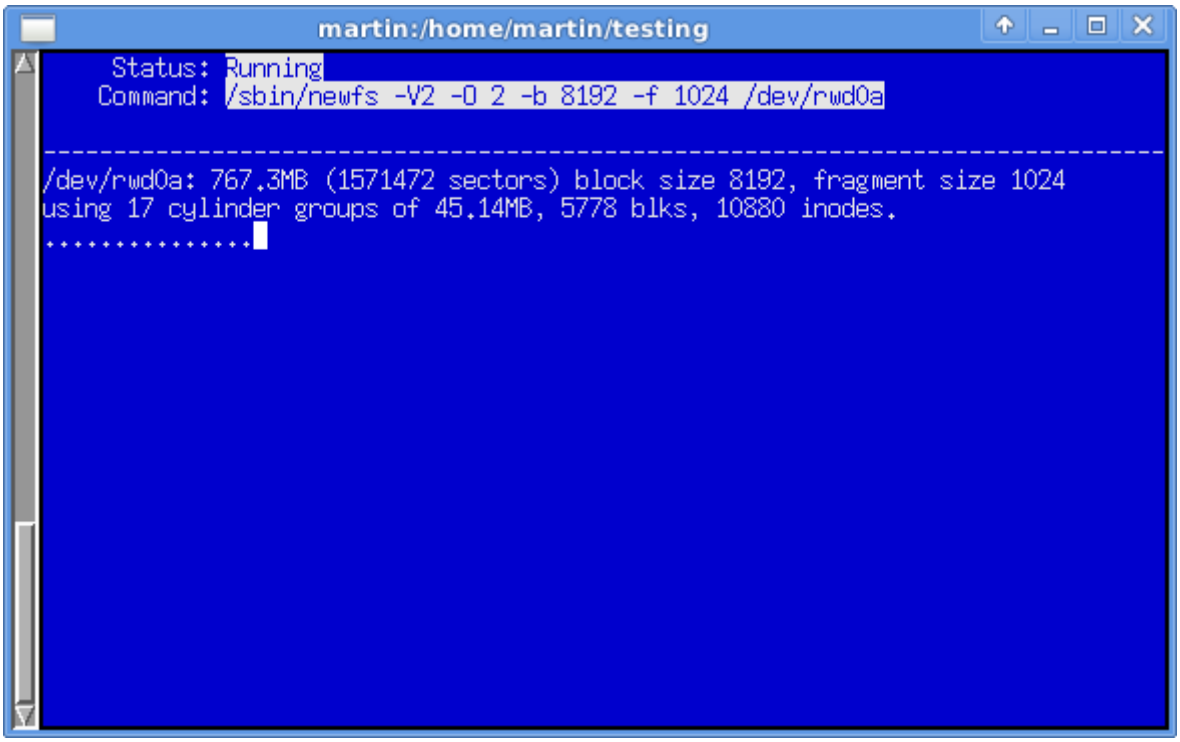
What is Anita?

Anita is a python script written by NetBSD developer Andreas Gustafsson. It fully automates a from-scratch installation of NetBSD on a virtual machine after (optionally) downloading the to-be-installed test version (or an official release) via internet. It starts qemu with an emulated serial console and matches strings in the console output. It then sends commands to sysinst (the NetBSD installer):



Currently it can do this on (emulated) i386, amd64 and sparc machines – other architectures could be added easily, if emulation in qemu is stable and sysinst support is available – it is just a matter of time/doing it.

Anita automatically calculates the size required for the virtual hard disk and creates the image for us. It then guides sysinst through all the partitioning and disk setup needed:



It then installs the binary sets needed for a test setup. After a short while the virtual machine is ready to reboot. Anita then logs into the freshly installed NetBSD and does a full ATF test

run, collecting the ATF generated log files and html output. If the test run finishes (without crashing the virtual machine) the result is an ATF summary report, which we will examine in more details shortly.

Using QEMU and from scratch installations for this kind of testing has both advantages and disadvantages. On the obvious plus side, we test code that is not updated often, like the boot sectors and secondary boot loaders, and of course we test the installer itself. On the downside, emulation is not always exact, especially floating point corner cases seem to be problematic, so all failures have to be verified on real hardware. The emulation of course is slower than the real hardware, so timing differs from "native" test runs, which is why we do both emulated and native runs systematically. If a test case fails, or even crashes the kernel we are lucky, because it did not crash the testing machine itself, but on the other hand due to the way Anita works, it is not easy to intercept at this point take over control (at the kernel debugger level) manually. Instead you will have to reproduce the failure again manually – which is not that hard and could even be done using the virtual setup created by Anita.

What does a test program look like?

There are different kinds of test programs, from very simple shell scripts to simple C programs, and even more complex C programs using a rump kernel process or file system server. Examples will be given and described in short. The overview should demonstrate that while tests may be awfully complex, the test code itself is always very short, all the complexity is hidden in the framework (some call it magic, some do not like it).

Test Example 1

Of course we have to start with a "hello world" example. Here is an excerpt of the test program that tries to verify the compiler is working and produces executable object files (The full test case is available here: toolchain/cc/t_hello):

```
 28 atf_test_case hello
 29 hello_head() {
 30     atf_set "descr" "compile and run \"hello world\""
 31     atf_set "require.progs" "cc"
 32 }
 33
 46 hello_body() {
 47     cat > test.c << EOF
 48 #include <stdio.h>
 49 #include <stdlib.h>
 50 int main(void) {printf("hello world\n");exit(0);}
 51 EOF
 52     atf_check -s exit:0 -o ignore -e ignore cc -o hello
  test.c
 53     atf_check -s exit:0 -o inline:"hello world\n" ./hello
 54 }
 55
117 atf_init_test_cases()
118 {
119
120     atf_add_test_case hello
123 }
```

The excerpt skips a few lines (but we will look at a few of them next) that add a bit of complexity by repeating the test but testing statically linked binaries and 32 bit binaries on 64 bit architectures that have a 32 bit cousin.

Obviously this test case is written in simple /bin/sh syntax, it is executed by the atf-shell, which just is /bin/sh with the atf bindings predefined. The test program contains the typical fragments for all ATF test programs:

- atf_init_test_cases: simply calls atf_add_test_case for all test cases in this test program
- hello_head: defines the ATF meta variables describing this test case, here it is just the description used in the final report to describe the test case, and a prerequisite – the test case can not run on installations where the C compiler is missing.
- hello_body: the main body of the test case. Here a short C program source code is written to a temporary file, then the C compiler is invoked and it's exit code checked to be zero. Next, the created binary is run and it's stdout compared with the "golden output", here listed literarily inline in the test code.

Very simple and straight forward. ATF will take care to execute the whole test case (or actually every single test case in a test program) all by itself, fresh, inside an empty temporary directory, and will also clean up after the test.
As mentioned above, this example is slightly stripped. The second test case, testing for static binaries, is boring – mostly a copy of the standard test case. The third test case is only used on architectures like amd64, mips64 and sparc64 that have 32 bit equivalents. The main magic added is this fragment:

```
 93     atf_check -s exit:0 -o ignore -e ignore cc -o hello32 -
   m32 test.c
 94     atf_check -s exit:0 -o ignore -e ignore cc -o hello64
   test.c
 95     file -b ./hello32 > ./ftype32
 96     file -b ./hello64 > ./ftype64
 97     if diff ./ftype32 ./ftype64 >/dev/null; then
 98           atf_fail "generated binaries do not differ"
 99     fi
100     echo "32bit binaries on this platform are:"
101     cat ./ftype32
102     echo "While native (64bit) binaries are:"
103     cat ./ftype64
104     atf_check -s exit:0 -o inline:"hello world\n" ./hello32
```

This tries (verbatim, by trial and error) weather the C compiler supports the –m32 flag, and if the resulting binary differs from the standard one. The full test code then proceeds to test 32bit static binaries as well – just because NetBSD once had a bug that made this combination fail.

Test Example 2

This is a very simple test program written in C (full source at: lib/libc/time/t_mktime.c):

```
36 ATF_TC(mktime);
37
38 ATF_TC_HEAD(mktime, tc)
39 {
40
41     atf_tc_set_md_var(tc, "descr", "Test mktime(3) with
   negative year");
42 }
43
44 ATF_TC_BODY(mktime, tc)
45 {
46     struct tm tms;
47
```

```
48      (void)memset(&tms, 0, sizeof(tms));
49      tms.tm_year = ~0;
50
51      errno = 0;
52
53      ATF_REQUIRE_ERRNO(errno, mktime(&tms) != (time_t)-1);
54  }
55
56  ATF_TP_ADD_TCS(tp)
57  {
58
59      ATF_TP_ADD_TC(tp, mktime);
60
61      return atf_no_error();
62  }
```

It follows the same structure as we saw before, only this time using the C binding:

- ATF_TP_ADD_TCS adds all test cases of the test program (here: only one)
- ATF_TC_HEAD(mktime) sets up meta variables (here: only the description of the test case)
- And finally: ATF_TC_BODY(mktime) is the body of the test case.

Again ATF takes care to run the test case in a new process and empty, temporary directory. The only other ATF specific instruction used here is the ATF_REQUIRE_ERRNO() macro, which is much like an ASSERT(), but additionally prints a proper error string from the errno value passed, if the assertion fails.
A typical, slightly more complex test program dealing with floating point, thus needing special handling for non IEEE754 floating point hardware like found on VAX, is this (see tests/lib/libm/t_round.c):

```
32  /*
33   * This tests for a bug in the initial implementation where
34   * precision was lost in an internal substraction, leading to
35   * rounding into the wrong direction.
36   */
37
38  /* 0.5 - EPSILON */
39  #define VAL     0x0.7fffffffffffcp0
40  #define VALF    0x0.7fffff8p0
41
42  #ifdef __vax__
43  #define SMALL_NUM     1.0e-38
44  #else
45  #define SMALL_NUM     1.0e-40
46  #endif
54  ATF_TC_BODY(round_dir, tc)
55  {
56      double a = VAL, b, c;
57      float af = VALF, bf, cf;
58
59      b = round(a);
60      bf = roundf(af);
61
62      ATF_CHECK(fabs(b) < SMALL_NUM);
63      ATF_CHECK(fabsf(bf) < SMALL_NUM);
64
65      c = round(-a);
66      cf = roundf(-af);
67
68      ATF_CHECK(fabs(c) < SMALL_NUM);
```

```
69      ATF_CHECK(fabsf(cf) < SMALL_NUM);
70  }
```

(boring, administrative parts removed for brevity). It is common to add a test case like this after errors have been analyzed and fixed – often based on a (non ATF, "free style") test program provided in a problem report, to verify the commited fix, catch all similar failures eventually happening on other architectures and avoid future regressions.

Test Example 3

Some tests require more complex setups. One example for this are the tests for libcurses. We will not look at source in details here, but give a general overview. For curses testing it is not enough to run some test input through (say) xterm and take a screenshot, comparing it pixel per pixel with a golden "how it should look like" image. The image would not catch timing errors, or suboptimal output sequences that, in the end, result in the same display.
Instead of this approach, a more complex setup was chosen by Brett Lymn, who did tests:

- Two helper programs are run, "director" which sends commands to "slave", which then creates the output it has been told to.
- A special terminfo entry is used, which displays typically hidden control sequences (e.g. <ESC>[…) as clear text, abbreviations of the terminfo name describing them. For example: `flash=flash, home=home` or `cud=cud%p1%dX, cud1=^J`.
- The director process reads test instructions from special "scripts" in its own special-purpose language

This way the tests do not only provide a binary "equal to golden output" result, but also can show diffs, that help tracking down underlying bug.
Besides simple output formatting, the tests also exercise input timing and read timeouts.

Test Example 4

The following test program uses RUMP magic (see kernel/tty/t_pr.c):

```
160 ATF_TC_BODY(ptyioctl, tc)
161 {
162     struct termios tio;
163     int fd;
164
165     rump_init();
166     fd = rump_sys_open("/dev/ptyp1", O_RDWR);
167     ATF_CHECK(fd != -1);
168
169     /*
170      * This used to die with null deref under ptcwakeup()
171      * atf_tc_expect_signal(-1, "PR kern/40688");
172      */
173     rump_sys_ioctl(fd, TIOCGETA, &tio);
174
175     rump_sys_close(fd);
176 }
```

This is C code linked against rump libraries, doing (explicit) system calls to the rump kernel. As mentioned in a comment, this test case used to crash the kernel – in this context only the rump kernel (i.e. the userland test case), not the host kernel (which would have aborted the whole test run prematurely).
Besides the obvious (does not crash the test machine) advantages, this also offers great debugging options. The rump (shared) libraries are built from the exact (unmodified) kernel sources as a new kernel would be. Just like using loadable kernel modules to debug features

you can easily load and unload, without rebooting the debug/development machine, this allows trying kernel changes quickly without rebooting.
To give a quick start for debugging, ATF invokes gdb on crashed test programs and adds a backtrace to the report:

```
Test case: kernel/tty/t_pr/ptyioctl
Termination reason

XFAIL: PR kern/40688
Standard error stream

Test program crashed; attempting to get stack trace
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
Core was generated by `t_pr'.
Program terminated with signal 11, Segmentation fault.
#0  0x00007f7ff704f657 in rumpns_selnotify () from
       /usr/lib/librump.so.0
#0  0x00007f7ff704f657 in rumpns_selnotify () from
       /usr/lib/librump.so.0
#1  0x00007f7ff7805986 in rumpns_ptcwakeup ()
   from /usr/lib/librumpkern_tty.so.0
#2  0x00007f7ff7805ade in rumpns_ptyioctl () from
       /usr/lib/librumpkern_tty.so.0
#3  0x00007f7ff7059c1a in rumpns_cdev_ioctl () from
       /usr/lib/librump.so.0
#4  0x00007f7ff704df53 in rumpns_VOP_IOCTL () from
       /usr/lib/librump.so.0
#5  0x00007f7ff7426636 in rumpns_vn_fifo_bypass ()
   from /usr/lib/librumpvfs.so.0
#6  0x00007f7ff703a2ca in rumpns_sys_ioctl () from
       /usr/lib/librump.so.0
#7  0x00007f7ff70729f3 in rumpns_sys_unmount () from
       /usr/lib/librump.so.0
#8  0x00007f7ff707635c in rump___sysimpl_ioctl () from
       /usr/lib/librump.so.0
#9  0x0000000000401976 in atfu_ptyioctl_body ()
#10 0x000000000040391f in atf_tc_run ()
#11 0x00000000004025e3 in atf_tp_main ()
#12 0x0000000000401771 in ___start ()
#13 0x0000000000000005 in ?? ()
#14 0x00007f7ffffffe88 in ?? ()
#15 0x00007f7ffffffe8d in ?? ()
#16 0x00007f7ffffffea7 in ?? ()
#17 0x00007f7ffffffebf in ?? ()
#18 0x00007f7ffffffed8 in ?? ()
#19 0x0000000000000000 in ?? ()
Stack trace complete
```

Note this trace is without symbols, but it is easy to compile the test cases (or more interestingly: the rump libraries) with full symbols. It is also easy to add full grown LOCKDEBUG support to the rump libraries, while running a LOCKDEBUG kernel on the test machine slows the whole machine down quite a lot (and is impossible if testing official release builds).

Test Example 5

This is an excerpt from a quite complex test scenario (full test case at usr.bin/shmif_dumpbus/t_basic.sh):

```
28 unpack_file()
29 {
30
31     atf_check -s exit:0 uudecode
  $(atf_get_srcdir)/${1}.bz2.uue
32     atf_check -s exit:0 bunzip2 -f ${1}.bz2
33 }
67 pcap()
68 {
69
70     unpack_file d_pcap.out
71     atf_check -s exit:0 -o ignore shmif_dumpbus -p pcap
  shmbus
72 #
73 #   should not fail anymore...
74 #
75 #   atf_expect_fail "PR bin/44721"
76     atf_check -s exit:0 -o file:d_pcap.out -e ignore \
77         tcpdump -tt -n -r pcap
78 }
```

The original intention of the test program was to verify a rump component, shmif_dumpbus, which emulates a network interface and is able to store traffic logs in pcap format. However, the test case triggered unexpected failures, described in PR 44721. The tcpdump output did not use the human friendly protocol names for some ICMP packets in automatic tests, but manully running tcpdump created the expected output. To tell a long story short: tcpdump only worked correctly when not invoked as root. If run as root (which is not required in the pcap reading invocation) it did a chroot and after that could not open /etc/protocols any more. After full analysis, the fix was simply to call setprotoent(1) before the chroot.

Test Example 6

This is an example of a rump based test that exercises corner cases/atypical setups (full test case at net/if_loop/t_pr.c):

```
55 /*
56  * Prepare rump, configure interface and route to cause
  fragmentation
57  */
58 static void
59 setup(void)
60 {
61     char ifname[IFNAMSIZ];
62     struct {
63           struct rt_msghdr m_rtm;
64           struct sockaddr_in m_sin;
65     } m_rtmsg;
66 #define rtm m_rtmsg.m_rtm
67 #define rsin m_rtmsg.m_sin
68     struct ifreq ifr;
69     int s;
70
71     rump_init();
72
73     /* first, config lo0 & route */
```

```
 74     strcpy(ifname, "lo0");
 75     netcfg_rump_if(ifname, "127.0.0.1", "255.0.0.0");
 76     netcfg_rump_route("127.0.0.1", "255.0.0.0", "127.0.0.1");
 77
 78     if ((s = rump_sys_socket(PF_ROUTE, SOCK_RAW, 0)) == -1)
 79           atf_tc_fail_errno("routing socket");
 80
 81     /*
 82      * set MTU for interface so that route MTU doesn't
 83      * get overridden by it.
 84      */
 85     memset(&ifr, 0, sizeof(ifr));
 86     strcpy(ifr.ifr_name, "lo0");
 87     ifr.ifr_mtu = 1300;
 88     if (rump_sys_ioctl(s, SIOCSIFMTU, &ifr) == -1)
 89           atf_tc_fail_errno("set mtu");
 90
 91     /* change route MTU to 100 */
 92     memset(&m_rtmsg, 0, sizeof(m_rtmsg));
 93     rtm.rtm_type = RTM_CHANGE;
 94     rtm.rtm_flags = RTF_STATIC;
 95     rtm.rtm_version = RTM_VERSION;
 96     rtm.rtm_seq = 3;
 97     rtm.rtm_inits = RTV_MTU;
 98     rtm.rtm_addrs = RTA_DST;
 99     rtm.rtm_rmx.rmx_mtu = 100;
100     rtm.rtm_msglen = sizeof(m_rtmsg);
101
102     memset(&rsin, 0, sizeof(rsin));
103     rsin.sin_family = AF_INET;
104     rsin.sin_len = sizeof(rsin);
105     rsin.sin_addr.s_addr = inet_addr("127.0.0.1");
106
107     if (rump_sys_write(s, &m_rtmsg, sizeof(m_rtmsg)) !=
   sizeof(m_rtmsg))
108           atf_tc_fail_errno("set route mtu");
109     rump_sys_close(s);
110 }
       161 ATF_TC_BODY(loopmtu, tc)
162 {
163     struct sockaddr_in sin;
164     char data[2000];
165     int s;
166
167     setup();
168
169     /* open raw socket */
170     s = rump_sys_socket(PF_INET, SOCK_RAW, 0);
171     if (s == -1)
172           atf_tc_fail_errno("raw socket");
173
174     /* then, send data */
175     memset(&sin, 0, sizeof(sin));
176     sin.sin_family = AF_INET;
177     sin.sin_len = sizeof(sin);
178     sin.sin_port = htons(12345);
179     sin.sin_addr.s_addr = inet_addr("127.0.0.1");
180
181     /*
182      * Should not fail anymore, PR has been fixed...
183      *
184      * atf_tc_expect_signal(SIGABRT, "PR kern/43664");
185      */
```

```
186     if (rump_sys_sendto(s, data, sizeof(data), 0,
187         (struct sockaddr *)&sin, sizeof(sin)) == -1)
188           atf_tc_fail_errno("sendto failed");
189 }
```

This code configures the loopback interface (on the rump kernel) with a small MTU and then sends a big "ping" – which requires fragmentation due to the MTU. This test case triggered a bug in NetBSD where the shortcut optimization to avoid IP checksums on loopback interfaces was handled inconsistently, triggering a (rump) kernel assertion. The test case was crafted after somebody accidently ran into the problem on a real setup, but using rump so it could be reproduced reliably independent of local setup.

Test Example 7

This test program is similar to the above example, but a lot simpler, since it only uses already available rump binaries (full test case at usr.sbin/traceroute/t_traceroute.sh):

```
28 netserver=\
29 "rump_server -lrumpnet -lrumpnet_net -lrumpnet_netinet -
  lrumpnet_shmif"
30
31 atf_test_case basic cleanup
32 basic_head()
33 {
34
35     atf_set "descr" "Does a simple three-hop traceroute"
36 }
37
38 cfgendpt ()
39 {
40
41     sock=${1}
42     addr=${2}
43     route=${3}
44     bus=${4}
45
46     export RUMP_SERVER=${sock}
47     atf_check -s exit:0 rump.ifconfig shmif0 create
48     atf_check -s exit:0 rump.ifconfig shmif0 linkstr ${bus}
49     atf_check -s exit:0 rump.ifconfig shmif0 inet ${addr}
  netmask 0xffffff00
50     atf_check -s exit:0 -o ignore rump.route add default
  ${route}
51 }
52
53 threeservers()
54 {
55
56     atf_check -s exit:0 ${netserver} unix://commsock1
57     atf_check -s exit:0 ${netserver} unix://commsock2
58     atf_check -s exit:0 ${netserver} unix://commsock3
59
60     # configure endpoints
61     cfgendpt unix://commsock1 1.2.3.4 1.2.3.1 bus1
62     cfgendpt unix://commsock3 2.3.4.5 2.3.4.1 bus2
63
64     # configure the router
65     export RUMP_SERVER=unix://commsock2
66     atf_check -s exit:0 rump.ifconfig shmif0 create
67     atf_check -s exit:0 rump.ifconfig shmif0 linkstr bus1
```

```
 68     atf_check -s exit:0 rump.ifconfig shmif0 inet 1.2.3.1
  netmask 0xffffff00
 69
 70     atf_check -s exit:0 rump.ifconfig shmif1 create
 71     atf_check -s exit:0 rump.ifconfig shmif1 linkstr bus2
 72     atf_check -s exit:0 rump.ifconfig shmif1 inet 2.3.4.1
  netmask 0xffffff00
 73 }
 74
 82 threetests()
 83 {
 84
 85     threeservers
 86     export RUMP_SERVER=unix://commsock1
 87     atf_check -s exit:0 -o inline:'1.2.3.1\n2.3.4.5\n' -e
  ignore -x  \
 88         "rump.traceroute ${1} -n 2.3.4.5 | awk '{print \$2}'"
 89     export RUMP_SERVER=unix://commsock3
 90     atf_check -s exit:0 -o inline:'2.3.4.1\n1.2.3.4\n' -e
  ignore -x \
 91         "rump.traceroute ${1} -n 1.2.3.4 | awk '{print \$2}'"
 92 }
 93
 94 basic_body()
 95 {
 96     threetests
 97 }
 98
121 atf_init_test_cases()
122 {
123
124     atf_add_test_case basic
126 }
```

The “rump.*” binaries used here are identical to their non-rump variants, but doing all syscalls to the rump server associated with them (instead of the host kernel). There are only a bit more than a dozen of such binaries around, and their introduction has been discussed very controversial.

Test Example 8

This example uses rump to get a well defined machine state, in this case an empty CD drive (full test case at dev/scsipi/t_cd.c):

```
50 ATF_TC_BODY(noisyeject, tc)
51 {
52     static char fname[] = "/dev/rcd0_";
53     int part, fd, arg = 0;
54
55     RL(part = getrawpartition());
56     fname[strlen(fname)-1] = 'a' + part;
57     rump_init();
58     RL(fd = rump_sys_open(fname, O_RDWR));
59     RL(rump_sys_ioctl(fd, DIOCEJECT, &arg));
60
61
       ATF_REQUIRE_EQ(rump_scsitest_err[RUMP_SCSITEST_NOISYSYNC]
  , 0);
62     RL(rump_sys_close(fd));
63     atf_tc_expect_fail("PR kern/43785");
```

```
64
       ATF_REQUIRE_EQ(rump_scsitest_err[RUMP_SCSITEST_NOISYSYNC]
  , 0);
65 }
```

This code asks the first CD drive on the rump kernel to eject – knowing the drive is empty (as the rump instance has just been exclusively created for the test program). This triggers a bogus error message from the SCSI layer. The problem is still not solved.
The macro RL() is a shortcut for “require libc”, testing typical libc return values and reporting a proper message created from errno if the return value tested is -1.
The array `rump_scsitest_err` is part of the rump scsitest library that implements a CD target for this test.

A typical test result summary

At the end of an ATF run, we get a list of failed test cases, and especially a list of the not expected (already documented) failures. This is from an Anita run for amd64 (on QEMU):

```
Failed test cases:
    fs/zfs/t_zpool:create, lib/libc/stdlib/t_strtod:strtod_round,
    lib/libc/stdlib/t_strtod:strtold_inf,
    lib/libc/stdlib/t_strtod:strtold_nan,
    lib/libm/t_infinity:infinity_long_double

Summary for 455 test programs:
    2602 passed test cases.
    5 failed test cases.
    39 expected failed test cases.
    70 skipped test cases.
```

Overall, this is a bad run: only one failure (the first one, t_zpool:create) was expected. But as floating point failures on emulated hardware are immediately suspicious, these results had to be verified on real hardware and were found to be erroneous – i.e. they do not happen on real hardware. Unfortunately the exact failures seem to be highly dependent on qemu version, host hardware and the phase of the moon. The issue is still under investigation. Similar problems were found elsewhere, and there is a bug tracker report collecting everything qemu related:

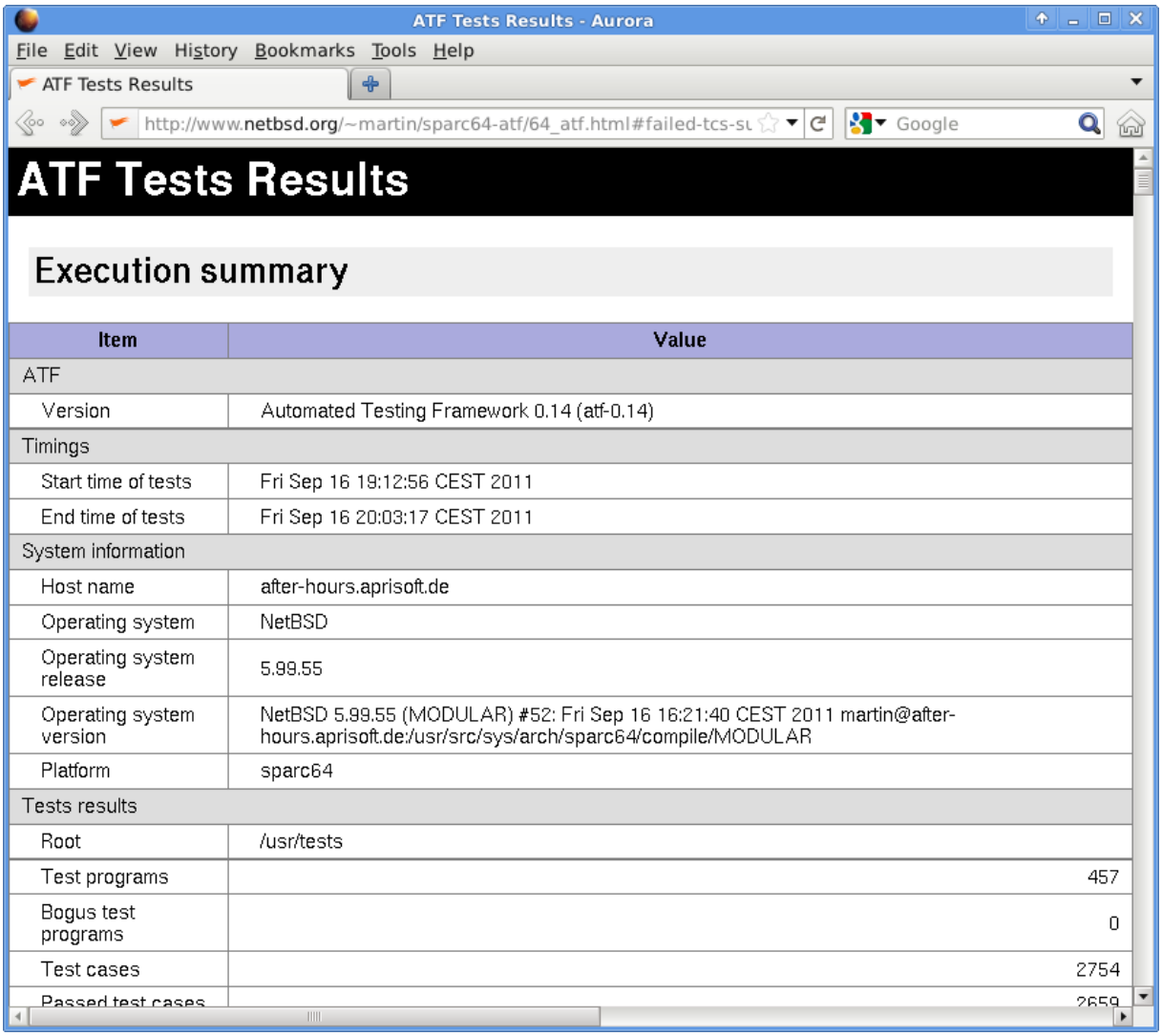
```
Module Name:   src
 Committed By:  jruoho
 Date:          Wed Sep 14 13:29:58 UTC 2011

 Modified Files:
        src/tests/lib/libm: t_cos.c t_sin.c t_tan.c

 Log Message:
 Additions to PR lib/45362: the float variants cosf(3), sinf(3), and
       tanf(3)
 do not detect NaN for positive and negative infinity on i386 (qemu).
```

See PR 45362 for details.

ATF also produces nice html formatted result with various details. A run on sparc64 (on real hardware) of a similar vintage of current produced this results:

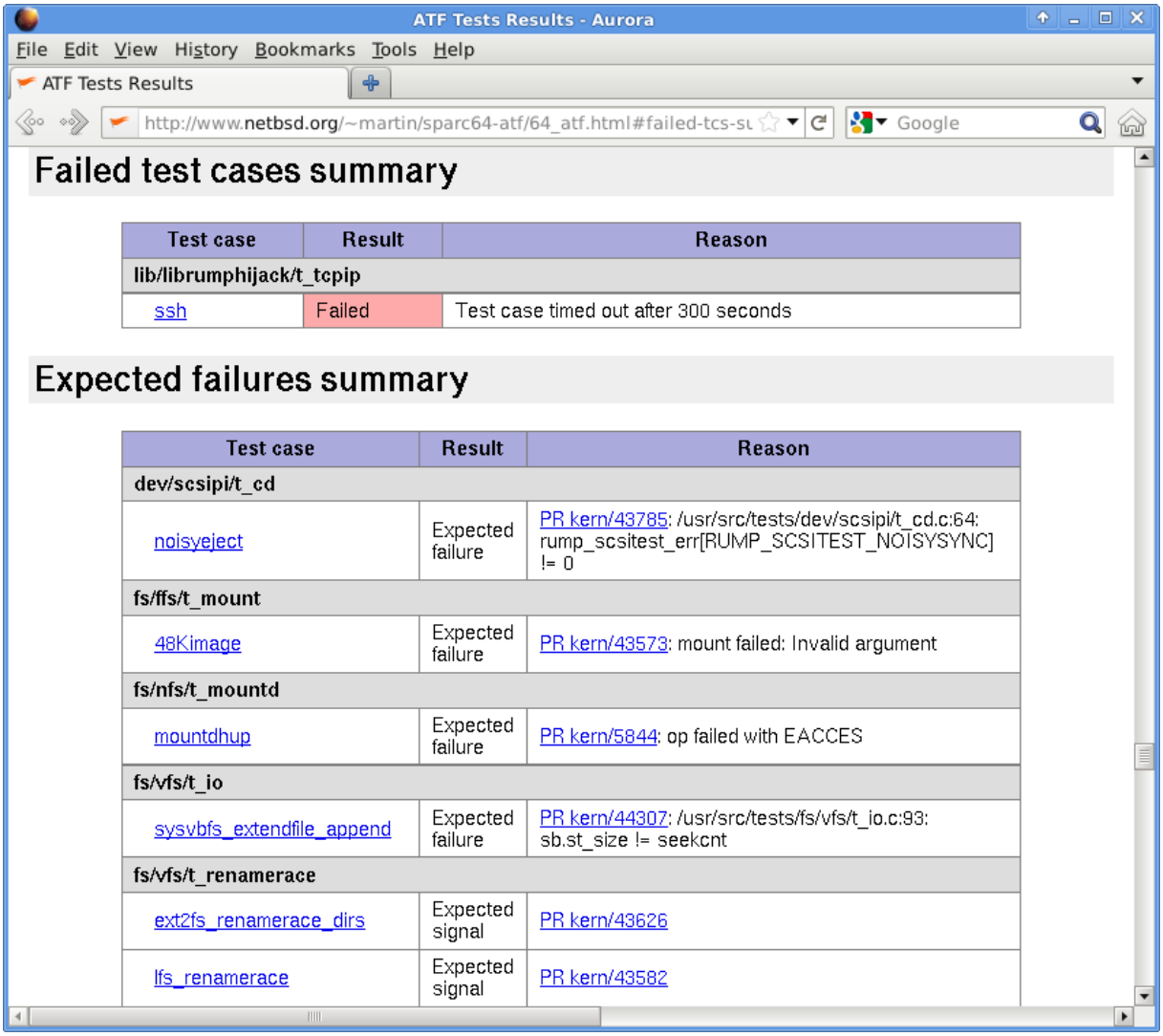


ATF Tests Results

Execution summary

Item	Value
ATF	
Version	Automated Testing Framework 0.14 (atf-0.14)
Timings	
Start time of tests	Fri Sep 16 19:12:56 CEST 2011
End time of tests	Fri Sep 16 20:03:17 CEST 2011
System information	
Host name	after-hours.aprisoft.de
Operating system	NetBSD
Operating system release	5.99.55
Operating system version	NetBSD 5.99.55 (MODULAR) #52: Fri Sep 16 16:21:40 CEST 2011 martin@after-hours.aprisoft.de:/usr/src/sys/arch/sparc64/compile/MODULAR
Platform	sparc64
Tests results	
Root	/usr/tests
Test programs	457
Bogus test programs	0
Test cases	2754
Passed test cases	2659

In this run, only a single test case (unexpectedly) failed, and the report summarizes this:



Failed test cases summary

Test case	Result	Reason
lib/librumphijack/t_tcpip		
ssh	Failed	Test case timed out after 300 seconds

Expected failures summary

Test case	Result	Reason
dev/scsipi/t_cd		
noisyeject	Expected failure	PR kern/43785: /usr/src/tests/dev/scsipi/t_cd.c:64: rump_scsitest_err[RUMP_SCSITEST_NOISYSYNC] != 0
fs/ffs/t_mount		
48Kimage	Expected failure	PR kern/43573: mount failed: Invalid argument
fs/nfs/t_mountd		
mountdhup	Expected failure	PR kern/5844: op failed with EACCES
fs/vfs/t_io		
sysvbfs_extendfile_append	Expected failure	PR kern/44307: /usr/src/tests/fs/vfs/t_io.c:93: sb.st_size != seekcnt
fs/vfs/t_renamerace		
ext2fs_renamerace_dirs	Expected signal	PR kern/43626
lfs_renamerace	Expected signal	PR kern/43582

This specific test case fails on sparc64 for a long time, but has been resistant to debugging. A possible explanation from the confusing debugging attempts is memory overwritten somewhere (maybe due to memory alignment differences not showing up [like this] on other architectures), that causes the malloc()/free() libc heap management to end up in an infinite loop.

What kind of bugs did we hit?

When doing regression tests systematically like in the setup described here, you would expect (after an initial hunt down and tame old bugs phase) to find mostly one type of bugs: regressions caused by new commits.
However, in practice this turned out to be not the case.

- Build breaking commits
  while we had automatic builds for all architectures for a long time, breakage has been silently ignored often. Now with daily test runs a working build will be watched more closely (since it prevents test results).
  This is a two-sided sword – for example mathew green was asked to back out the switch to gcc 4.5.3 because compiling it triggered a /bin/sh bug (at least on the continuous build-and-test machine), probably related to the path length of the name of the root directory used for the build.
- Bugs in test cases
  creating tests is simple, and even useful if you do not (yet) fully understand a problem you just noticed. This results in bugs in test case. When the author started doing regular test runs on sparc64 the number of failures was an order of magnitude higher than on other architectures already doing regular runs. The majority of this difference was due to bugs due to the usual portability problems in test cases, starting with alignment and not ending with different page sizes. Another typical bug we run in every now and then is a test program assuming stdin is a terminal, while formally ATF makes no such guarantee and might switch stdin to /dev/null in the future.
- Bugs in the testing framework
  An example for a (still open) bug report is PR 44731: some ATF self tests fail on alpha
- Bugs in emulators
  as already mentioned all bugs found on emulators should be verified against real hardware, especially qemu and floating point results on i386/amd64 seem to always be suspicious.
- Bugs in the compiler
  as explained below gcc on VAX fails to create proper code to do exception handling, this was discovered by running the ATF tests that exercise ATF itself. Other toolchain related tests have been shown already, for example the first test case presented doing a compile of “hello world” and testing the binary, for shared, static, and 32 bit versions and all combinations.
- Bugs in libraries
  i.e. secondary bugs, not the main target of the test case (no prominent examples)
- Generic bugs
  i.e. things “randomly” failing due to general changes – we saw this effect during introduction of TLS support, where arbitrary (at first look) tests started failing and only deeper analysis found the correlation. For example on sparc the `%g7` register is used to hold the thread local storage pointer, but this register was not preserved properly during all system and libc assembler function calls.
- Real regressions

What did we learn?

Testing (and/or setting your mind up for testing early during development of new features) changes the ways developers work. A few examples:

- Manuel Bouyer recently reworked the file system quota framework to be mostly independent of the FFS on disk structures. He started with creating lots of test cases upfront, making sure not to break anything in the existing quota implementation and keeping user land compatibility.
- Jeff Rizzo started working on the file system resize utility resize_ffs, which had been rarely used because it was rumored to be buggy. Jeff collected all open bug reports about it, and while working himself into the source created automated test cases for all documented problems and new ones he found. Afterwards he understood the code pretty well and started fixing the problems.
- FreeBSD issued a security advisory about buffer overruns with UNIX (local) domain socket names. Initially we were not 100% sure NetBSD wasn't affected, so Christos Zoulas not only reviewed the code but also created a test case to "prove" it (see NetBSD Security Note 2011-1). Of course this is no formal prove, but on the other hand it is a good way to prevent accidental problems in this area, as the code path preventing the problem is not exactly obvious.

Simple, isolated test cases also are a huge benefit when trying to fix a bug. The test runs provide a stack backtrace from crashes (or ASSERTs firing) in rump kernels, which often directly point the developer examining the bug to the problematic area of code. Test cases can be run (with a little effort) under gdb, so even in the non crashing cases, they are a valuable debugging aid.

Overall, our lesson was:

- Creating test cases/programs is very simple
- Creating good test cases/programs is non trivial sometimes
- Creating a test-driven mindset increases overall quality often inspires quick build fixes
- Test log soften can pinpoint „innocent" looking commits causing big problems – but only if the tree is not broken for too long periods (which in former time often was silently accepted for –current)
- Adding test programs tot he NetBSD test suite is tedious (but this is an infrastructure problem outside the scope of this paper)
- Test cases should be heavily commented if they are non-trivial (see dev/raidframe/t_raid:raid1_comp0fail and PR 44251 for an example where the test code itself did not document the problem clear enough)

Why are we not testing on VAX?

We would like to test as many architectures supported by NetBSD as possible. Every new hardware platform makes for more bugs being exposed – both in that platforms machine dependend code, as well as in machine independend code that gets exercised slightly different.

In the early ATF design phase we considered it "no harm" to write ATF in C++, mostly because Julio (as well yours truly) are hard core C++ programmers in real life, and other parts of the NetBSD source tree already required a working C++ environment (nowadays this is only groff, and even that may be moved to pkgsrc sometime soon, since we can use mdcoml to format man pages).

We only noticed the downside when the deed was done: we have no “good enough” working C++ compiler for VAX. ATF relies on exception handling – no surprise here. But: gcc on VAX has severe problems, multiple layers of bugs affecting exception handling. Overall: it just does not work yet.
As soon as this will be fixed, the author will definitively start to run regular native tests on a VAX station 4000/96. The architecture is different in various ways (starting with the pre IEE754 floating point format), so tests will certainly provide interesting results.

Conclusion (or: why did nobody start testing like this years ago?)

Testing is not sexy to developers. It involves a huge amount of grunt work, and even more if you have no workable framework like we have now. And without stable technologies like RUMP, ATF and Anita some kind of tests would be impossible.
However, knowing current bugs, as many as possible of them, is always worth the effort.

NetBSD did have some regression tests of various kinds in src/regress, but the only framework there was at the “make” level: you could recursively build all tests and run them by doing “make regress” at the top level source directory. The only log/result/summary available would be the output of all tests, or make aborting the whole run if a test program returned an error code. Now some test cases reported errors to stderr and did explicitly not fail the return code, other test programs would cause different output if run as root. Overall, it was not easy to run, nor extract and interpret the results.

While there are still areas to improve, in the end the forced automatic testing of random “current” release builds and the upcoming NetBSD-6 release branch already helped to catch a lot of bad regressions very early, and surely will help to make the release cycle shorter as well as the quality of the release better.

If you have not set up mandatory, automatic testing for your project yet, please re-think and just do it. Testing will require valuable resources, but they are not wasted!