



®

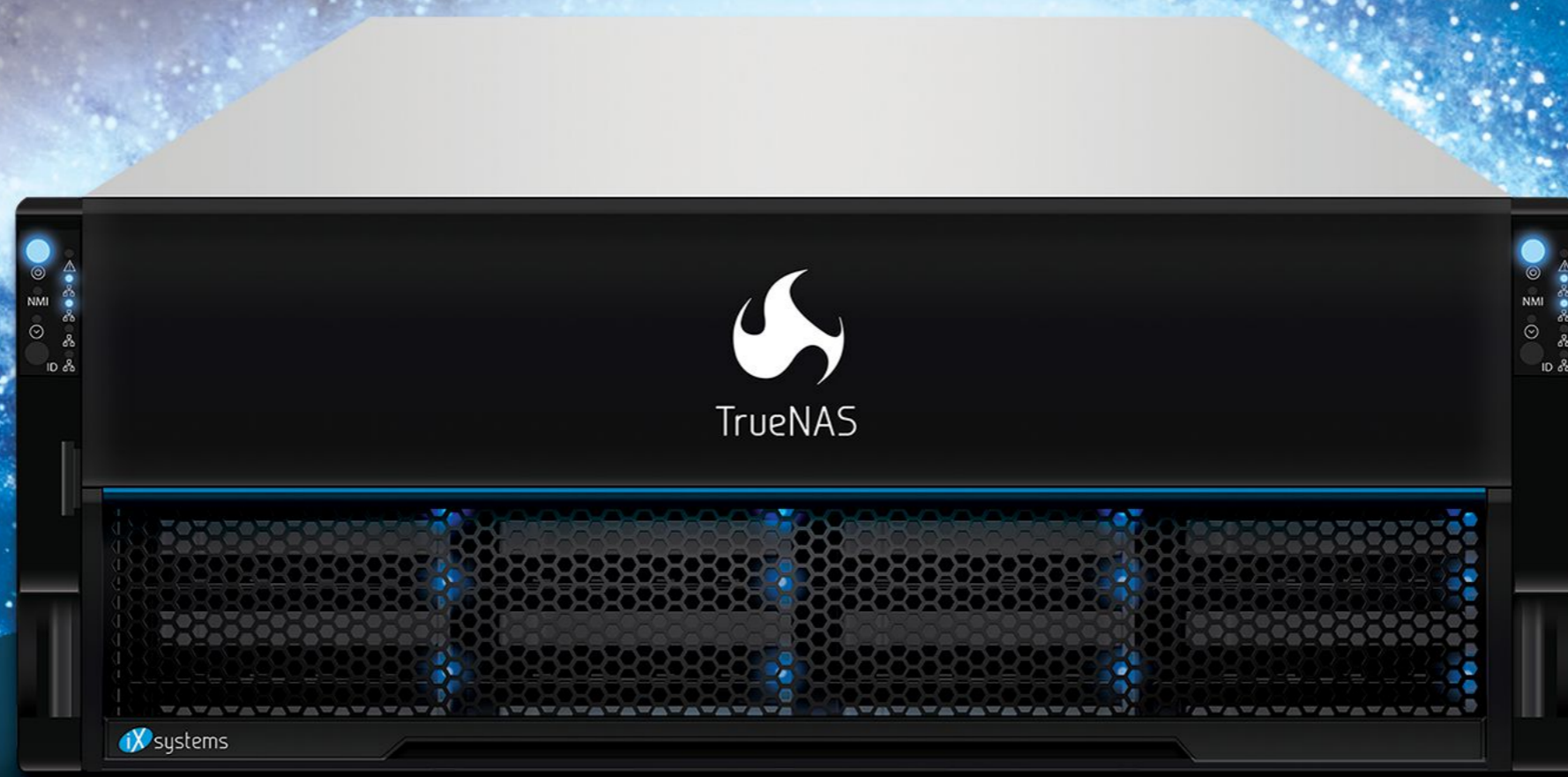
FreeBSD[®] JOURNAL

November/December 2020

- FreeBSD mini-Git Primer
- Kernel Fuzzing with syzkaller
- FreeBSD Foundation Year-end Update
- Tips for Running an Online Conference
- Network Monitoring on the Console

TrueNAS® M-SERIES
Powerfully Scalable Enterprise Storage

OPEN STORAGE FOR ENTERPRISE WORKLOADS



UTILIZES FLASH-OPTIMIZED ZFS TECHNOLOGY
IDEAL FOR LATENCY-SENSITIVE AND BUSINESS-CRITICAL
VIRTUAL MACHINES AND PHYSICAL WORKLOADS.

**PERFORMANCE AND SCALE
WITHOUT COMPROMISE**

**INTELLIGENT STORAGE
OPTIMIZATION**

**SELF-HEALING DATA
PROTECTION**

**UNLIMITED SNAPSHOTS AND
REPLICATION**

Contact iXsystems to Learn More about what TrueNAS® can do for your business!

[ixsystems.com/TrueNAS](https://www.ixsystems.com/TrueNAS) | (855) GREP-4-iX



FreeBSD[®] JOURNAL

Editorial Board

- John Baldwin • FreeBSD Developer and Chair of FreeBSD Journal Editorial Board.
- Justin Gibbs • Founder of the FreeBSD Foundation, President of the FreeBSD Foundation, and a Software Engineer at Facebook.
- Daichi Goto • Director at BSD Consulting Inc. (Tokyo).
- Tom Jones • FreeBSD Developer, Internet Engineer and Researcher at the University of Aberdeen.
- Dru Lavigne • Author of *BSD Hacks* and *The Best of FreeBSD Basics*.
- Michael W Lucas • Author of *Absolute FreeBSD*.
- Ed Maste • Director of Project Development, FreeBSD Foundation and Member of the FreeBSD Core Team.
- Kirk McKusick • Treasurer of the FreeBSD Foundation Board, and lead author of *The Design and Implementation* book series.
- George V. Neville-Neil • Director of the FreeBSD Foundation Board, Member of the FreeBSD Core Team, and co-author of *The Design and Implementation of the FreeBSD Operating System*.
- Philip Paeps • Secretary of the FreeBSD Foundation Board, FreeBSD Committer, and Independent Consultant.
- Kristof Provost • Treasurer of the EuroBSDCon Foundation, FreeBSD Committer, and Independent Consultant.
- Hiroki Sato • Director of the FreeBSD Foundation Board, Chair of Asia BSDCon, Member of the FreeBSD Core Team, and Assistant Professor at Tokyo Institute of Technology.
- Benedict Reuschling • Vice President of the FreeBSD Foundation Board and a FreeBSD Documentation Committer.
- Robert N. M. Watson • Director of the FreeBSD Foundation Board, Founder of the TrustedBSD Project, and University Senior Lecturer at the University of Cambridge.
- Mariusz Zaborski • FreeBSD Developer, Manager at Fudo Security.

S&W PUBLISHING LLC

PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Design & Production • Reuter & Associates

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).

Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsd.foundation.org

Copyright © 2020 by FreeBSD Foundation. All rights reserved. This magazine may not be reproduced in whole or in part without written permission from the publisher.

LETTER

from the Foundation

*Dear Readers,
We send you all
the very best wishes
for a happy year ahead.*



FreeBSD Foundation





8 FreeBSD mini-Git Primer

By Warner Losh

16 Kernel Fuzzing with syzkaller

By Mark Johnston

29 FreeBSD Foundation Year-end Update

By Deb Goodkin

3 Foundation Letter

Foundation Letter Happy New Year! *By Deb Goodkin*

5 We Get Letters

Dear Sick of Unreasonable Requests. *By Michael W Lucas*

32 Book Review

Mastering Vim Quickly by Jovica Ilic
reviewed by Benedict Reuschling

33 Conferences

Tips for Running an Online Conference *By Dan Langille*

36 Practical Ports

Network Monitoring on the Console *By Benedict Reuschling*

40 Events Calendar

By Anne Dickison

WeGetletters

by Michael W Lucas



Dear Last Desperate Chance,

I've been round and round with the boss, explaining over and over that systems administration is an art as much as a craft and I can't write a complete procedure for every last thing I do. He's got a copy of the Policies and Procedures manual from his previous job at the StarBux Coffee Hole and says that figuring out bad ARP caches can't be nearly as bad as being a barista, and that's documented down to exactly how to make the foam in the top look like the corporate logo and how covering up the cup's copyright notice is a termination offense. I've tried everything else, so now I'm trying you. Please, give me an **Argument From Authority** that declares documenting systems administration is doomed to fail.

—Sick of Unreasonable Requests

Dear SOUR,

"Documenting systems administration is doomed to fail." See, I can lie with the best of them.

You did quite well in waiting to contact me until the very end. Unfortunately, you desecrated that immaculate record by contacting me. It's this failure your coworkers, family, and the random strangers reading this column will remember you for. But today, it's either correct your ignorance or finish writing up the apology letter my settlement with the Avocado Liberation Front demands, so I'll give it a stab.

I'm fairly certain that you don't even know what your job is. Yes, you received a farcical document when you started that said things like "install the software" and "debug PHPython" and that oh so precious "other duties as events warrant" meaning that the boss can drop a mountain of what he's been told is web server load balancer droppings on your desk and tell you to "grope" it for salvageable SMTP headers. You'll probably react by declaring to those entities unlucky enough to live with you that your boss is an idiot who doesn't even know how to spell SMTP rather than the far more productive process of determining exactly who on the network team dared displease you and how to best demonstrate the distinctly discomforting consequences of doing so upon them and anyone within smelling distance of their cubicle.

None of this is your job, mind you. It's simply a prerequisite to doing your job. Your job? Your real job? The thing you're paid to do? It has nothing to do with system administration.

Your job is to make your boss happy.

Not your employer. Not the company.

Your boss. Your immediate supervisor.

That's it. That's the whole job. You were hired to make him happy--in a computery way.

Sure, he'll disguise it behind fancy lingo like *stockholder value* and *delight customers* and *FIPS compliance*, but it's all about making him happy. He exists to make his boss happy, and so on. A business is a tree of boot-kissing, like a TLS Chain of Trust but even more malignant.

Your boss doesn't truly want a manual on how to use `ls(1)`. If he insists he does, make him get out his crowbar and pry open his wallet to pay for a copy of *Nemeth's Unix and Linux System Administration Handbook*. What he wants is a great big teddy binder that he can cuddle and show off to his boss. He exists to make his boss happy, after all.

So, give him what he wants, not what he asked for.

Start with a wiki. You young punks like wikis. I don't know why you can't be bothered to learn Docbook and SGML and just pretend to be a competent worthwhile person, but if I concern myself with your lack of character, this column will go on far too long, and if I don't get that inane apology letter in the post, the judge will hold me in contempt again. My attorney insists that reaching an even dozen citations will not make me go up a level when that's clearly untrue.

So. A wiki. Or a Markdown. One of them.

Pick the most tedious task you perform—say, installing software on a server. The first time you make a server reach across the Internet and grab software and install it all on its own you might feel a frisson of wonder, but as a professional sysadmin, you're too aware of all the times a simple install plunged you into the infernal abyss. Today you type `pkg install fubar` and watch as the package tools update the repository and search for incompatibilities and meticulously trash your LDAP database. Cast back your mind to the days when you cared about your job—yes, I know it's difficult to dig that far back, and recalling that chipper youth who was going to change the world threatens your carefully maintained shell of indifference, but that brittle shell needs substantial reinforcement and you won't develop such without fierce practice. It won't be sufficiently robust until anyone who dares poke it by asking you an innocent question gets drenched in bitter torrents of bile.

If you cared, you'd back up the host before installing anything on it. Maybe not the whole host. User home directories can burn and die, of course, because the peasantry has been told not to trust computers with anything important, but the software configuration files and data files and all those things that you're responsible for, sure, they should get backed up. Or snapshotted, or tarsnapped, or microengraved onto mysterious three-sided steel monoliths and erected in the Utah desert as a monument to all the disaster recovery plans that never got acted on because the hurricanes and avalanches were so inconsiderate as to skip the party.

So, scribble "backup" on a piece of scrap paper.

Not legibly, mind you. Just clear enough that the sight of the scrawl makes you think backup, but not plainly enough to make the housekeeper emptying your trash bin think you're considering backups. Let's say that's all you can think of. Thinking is a skill like any other, and you can improve if you keep practicing.

Maybe all the backup you need is a boot environment. Boot environments are free so long as you don't churn your data. Everybody likes free. So, write on your wiki.

Installing Software

1. *Create a boot environment*
2. *Run `pkg install whatever`*

Now comes the selfish bit. Your job is to make your boss happy, but that's not your *goal*. The true goal of system administration is to minimize sysadmin suffering (Sysadmin Rule #5). Minimizing sysadmin suffering demands consistency. Consistency means scripting. Sysadmins like to script.

So, write a script for installing software the way you want it installed.

Add a note at the bottom of your wiki that says, *This procedure is implemented as `break-everything.sh`.*

As you slog through the muck of making your boss happy and crafting an unusual web page that will have amusing affects on the load balancer and give the network folks their due cardiac tremors, maybe you'll hit a problem that leads to a troublesome library on a host. Was that library there last week, before you ran the software install? You trudge through boot environments and find out. A list of what software was installed on a host before you installed a package would reduce your suffering, though. Add that to your procedure, and your script. Yes, this bears a suspicious resemblance to programmers having to document their code. Procedures are programs.

The next time your boss brings up the documentation thing, print out your wiki and hand it to him.

You need more procedures? Well, what other scripts have you written to make your life easier?

Bleed out documentation quickly enough to content the boss, but not so quickly as to make him jaded. He'll be happiest if he sincerely believes you work really hard on the tasks he assigns.

Keep it up long enough, and you'll be able to hand your job to some optimistic newcomer and get a new job, where you get an entire team of people who don't yet understand that their job is to make you happy.

Have a question for Michael?
Send it to letters@freebsdjournal.org



MICHAEL W LUCAS's most recent books include *SNMP Mastery*, *Cash Flow for Creators*, and *Drinking Heavy Water*, plus a bunch more at <https://mwl.io>. Under no circumstances is he allowed near users.

FreeBSD mini-Git Primer

BY WARNER LOSH

The FreeBSD project has begun its transition from Subversion to Git. This move has been over a year in the planning and represents the next step in FreeBSD's continuing efforts to improve its workflow. The project hopes that the larger ecosystem for Git will help it improve its continuous integration (CI) efforts; make it easier to submit patches; and generally increase the quality of the project.

This article is aimed at the FreeBSD user who downloads sources, has local changes and sometimes contributes them back to the project. It will provide an introduction to the FreeBSD's use of Git for an audience already generally familiar with the basics of Git. Where possible, a pointer to a more in-depth treatment of Git will be provided. There are many primers for Git on the web, but the [Git Book](#) provides one of the better treatments.

Keeping Current With FreeBSD src Tree

First step: cloning a tree. This downloads the entire tree. There are two ways to download. Most people will want to do a deep clone of the repo. However, there are times that you may wish to do a shallow clone.

Branch Names

The branch names in the new Git repo are similar to the old names. For the stable branches, they are stable/X where X is the major release (like 11 or 12). The main branch in the new repo is `main`. The main branch in the old GitHub mirror is `master`. Both reflect the defaults of Git at the time they were created. The main/master branch is the default branch if you omit the `-b branch` or `--branch branch` options below.

Repositories

At the moment, there are two repositories. The hashes are different between them. The old GitHub repo is similar to the new cgit repo. However, there are a large number of mistakes in the GitHub repo that required us to regenerate the export when we migrated to having a Git repo be the source of truth for the project.

The GitHub repo is at <https://github.com/freebsd/freebsd.git> The new production repo is at either <https://git.freebsd.org/src.git> or <ssh://anonssh@git.freebsd.org/src.git> depending on which transport you wish to use. These will be \$URL in the commands below.

Note: The project doesn't use submodules as they are a poor fit for our workflows and development model. How we track changes in third-party applications is discussed elsewhere and generally of little concern to the casual user.

Deep Clone

A deep clone pulls in the entire tree, as well as all the history and branches. It's the easiest to do. It also allows you to use Git's worktree feature to have all your active branches checked out into separate directories but with only one copy of the repository.

```
% git clone -o freebsd $URL -b branch [dir]
```

is how you make a deep clone. `branch` should be one of the branches listed in the previous section. It is optional if it is the main/master branch. `dir` is an optional directory to place it in (the default will be the name of the repo you are cloning (freebsd or src)).

You'll want a deep clone if you are interested in the history, plan on making local changes, or plan on working on more than one branch. It's the easiest to keep up to date as well. If you are interested in the history, but are working with only one branch and are short on space, you can also use `--single-branch` to only download the one branch (though some merge commits will not reference the merged-from branch which may be important for some users who are interested in detailed versions of history).

Shallow Clone

A shallow clone copies just the most current code, but none or little of the history. This can be useful when you need to build a specific revision of FreeBSD, or when you are just starting out and plan to track the tree more fully. You can also use it to limit history to only so many revisions.

```
% git clone -o freebsd -b branch --depth 1 $URL [dir]
```

This clones the repository, but only has the most recent version in the repository. The rest of the history is not downloaded. Should you change your mind later, you can do `git fetch --unshallow` to get the old history.

Building

Once you've downloaded, building is done as described in the handbook, eg:

```
% cd src
% make buildworld
% make buildkernel
% make installkernel
% make installworld
```

so that won't be covered in depth here.

Updating

To update both types of trees uses the same commands. This pulls in all the revisions since your last update.

```
% git pull --ff-only
```

will update the tree. In Git, a `fast forward` merge is one that only needs to set a new branch pointer and doesn't need to re-create the commits. By always doing a `fast forward` merge/

pull, you'll ensure that you have an identical copy of the FreeBSD tree. This will be important if you want to maintain local patches.

See below for how to manage local changes. The simplest is to use `--autostash` on the `git pull` command, but more sophisticated options are available.

Selecting a Specific Version

In Git, the `git checkout` checks out both branches and specific versions. Git's versions are the long hashes rather than a sequential number.

When you checkout a specific version, just specify the hash you want on the command line (the Git Log command can help you decide which hash you might want):

```
% git checkout 08b8197a74
```

and you have that checked out. You'll be greeted with a message similar to the following:

```
Note: checking out '08b8197a742a96964d2924391bf9dfef788865d'.
```

```
You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.
```

```
If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:
```

```
git checkout -b
```

```
HEAD is now at 08b8197a742a hook gpiokeys.4 to the build
```

where the last line is generated from the hash you are checking out and the first line of the commit message from that revision. The hash can be abbreviated to the shortest unique length. Git itself is inconsistent about how many digits it displays.

Bisecting

Sometimes, things go wrong. The last version worked, but the one you just updated to does not. A developer may ask to bisect the problem to track down which commit caused the regression.

If you've read the last section, you may be thinking to yourself "How the heck do I bisect with crazy version numbers like that?" then this section is for you. It's also for you if you didn't think that, but also want to bisect.

Fortunately, one uses the `git bisect` command. Here's a brief outline of how to use it. For more information, I'd suggest <https://www.metaltoad.com/blog/beginners-guide-git-bisect-process-elimination> or <https://git-scm.com/docs/git-bisect> for more details. The man page is good at describing what can go wrong, what to do when versions won't build, when you want to use terms other than good and bad, etc., none of which will be covered here.

`git bisect start` will start the bisection process. Next, you need to tell a range to go through. `git bisect good XXXXXX` will tell it the working version and `git bisect bad`

XXXXX will tell it the bad version. The bad version will almost always be HEAD (a special tag for what you have checked out). The good version will be the last one you checked out.

A quick aside: if you want to know the last version you checked out, you should use `git reflog`:

```
5ef0bd68b515 (HEAD -> master, freebsd/master, freebsd/HEAD) HEAD@{0}:
pull --ff-only: Fast-forward
a8163e165c5b (upstream/master) HEAD@{1}: checkout: moving from b6fb97efb682994f59b-
21fe4efb3fcfc0e5b9eeb to master
```

shows me moving the working tree to the master branch (a816...) and then updating from upstream (to 5ef0...). In this case, bad would be HEAD (or 5ef0bd68) and good would be a8163e165. As you can see from the output, HEAD@{1} also often works, but isn't foolproof if you've done other things to your git tree after updating, but before you discover the need to bisect.

Back to git bisect. Set the good version first, then set the bad (though the order doesn't matter). When you set the bad version, it will give you some statistics on the process:

```
% git bisect start
% git bisect good a8163e165c5b
% git bisect bad HEAD
Bisecting: 1722 revisions left to test after this (roughly 11 steps)
[c427b3158fd8225f6afc09e7e6f62326f9e4de7e] Fixup r361997 by balancing parens. Duh.
```

You'd then build/install that version. If it's good you'd type `git bisect good` otherwise `git bisect bad`. You'll get a similar message to the above each step. When you are done, report the bad version to the developer (or fix the bug yourself and send a patch). `git bisect reset` will end the process and return you back to where you started (usually tip of main). Again, the git-bisect manual (linked above) is a good resource for when things go wrong or for unusual cases.

Ports Considerations

The ports tree operates the same way. The branch names are different and the repos are in different locations.

The GitHub mirror is at <https://github.com/freebsd/freebsd-ports.git>. The cgit mirror is <https://cgit-beta.freebsd.org/ports.git> for now. The production Git repo will be <https://git.freebsd.org/ports.git> or <ssh://anonsshgit.freebsd.org/ports.git> when the time comes. The plan is to switch the ports repository from Subversion to Git at the end of Q1 2021.

As with ports, the current branches are `master` and `main` respectively. The quarterly branches are named the same as in FreeBSD's svn repo. Due to bugs in the converter, there will likely be a hash respin when the ports svn repo migrates to git, just like the src and doc repos.

Coping with Local Changes

This section addresses tracking local changes. If you have no local changes, you can stop reading now (it's the last section and OK to skip).

One item that's important for all of them: all changes are local until pushed. Unlike svn, Git uses a distributed model. For users, for most things, there's very little difference. However, if you have local changes, you can use the same tool to manage them as you use to pull in changes from FreeBSD. All changes that you've not pushed are local and can easily be modified (git rebase, discussed below, does this).

Keeping Local Changes

The simplest way to keep local changes (especially trivial ones) is to use `git stash`. In its simplest form, you use `git stash` to record the changes (which pushes them onto the stash stack). Most people use this to save changes before updating the tree as described above. They then use `git stash apply` to re-apply them to the tree. The stash is a stack of changes that can be examined with `git stash list`. The git-stash man page (<https://git-scm.com/docs/git-stash>) has all the details.

This method is suitable when you have tiny tweaks to the tree. When you have anything non trivial, you'll likely be better off keeping a local branch and rebasing. Stashing is also integrated with the `Git pull` command: just add `--autostash` to the command line.

Keeping a Local Branch

It's much easier to keep a local branch with Git than Subversion. In Subversion you need to merge the commit, and resolve the conflicts. This is manageable, but can lead to a convoluted history that's hard to upstream should that ever be necessary, or hard to replicate if you need to do so. Git also allows one to merge, along with the same problems. That's one way to manage the branch, but it's the least flexible.

In addition to merging, Git supports the concept of `rebasing` which avoids these issues. The `git rebase` command replays all the commits of a branch at a newer location on the parent branch. We'll cover the most common scenarios that arise using it.

Create a Branch

Let's say you want to make a hack to FreeBSD's `ls` command to never, ever do color. There are many reasons to do this, but this example will use that as a baseline. The FreeBSD `ls` command changes from time to time, and you'll need to cope with those changes. Fortunately, with `git rebase` it usually is automatic.

```
% cd src
% git checkout main
% git checkout -b no-color-ls
% cd bin/ls
% vi ls.c # hack the changes in
% git diff # check the changes
diff --git a/bin/ls/ls.c b/bin/ls/ls.c
index 7378268867ef..cfc3f4342531 100644
--- a/bin/ls/ls.c
+++ b/bin/ls/ls.c
@@ -66,6 +66,7 @@ __FBSDID("$FreeBSD$");
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```



```

+#undef COLORLS
#ifdef COLORLS
#include <termcap.h>
#include <signal.h>
% # these look good, make the commit...
% git commit ls.c

```

The commit will pop you into an editor to describe what you've done. Once you enter that, you have your own local branch in the Git repo. Build and install it like you normally would, following the directions in the handbook. Git differs from other version control systems in that you have to tell it explicitly which files to use. I've opted to do it on the commit command line, but you can also do it with `git add` which many of the more in depth tutorials cover.

Time to Update

When it's time to bring in a new version, it's almost the same as w/o the branches. You would update like you would above, but there's one extra command before you update, and one after. The following assumes you are starting with an unmodified tree. It's important to start rebasing operations with a clean tree (Git usually requires this).

```

% git checkout main
% git pull --no-ff
% git rebase -i main no-color-ls

```

This will bring up an editor that lists all the commits in it. For this example, don't change it at all. This is typically what you are doing while updating the baseline (though you also use the `git rebase` command to curate the commits you have in the branch).

Once you're done with the above, you've move the commits to `ls.c` forward from the old version of FreeBSD to the newer one.

Sometimes there's merge conflicts. That's OK. Don't panic. You'd handle them the same as you would any other merge conflicts. To keep it simple, I'll just describe a common issue you might see. A pointer to a more complete treatment can be found at the end of this section.

Let's say the merge includes changes upstream in a radical shift to `terminfo` as well as a name change for the option. When you updated, you might see something like this:

```

Auto-merging bin/ls/ls.c
CONFLICT (content): Merge conflict in bin/ls/ls.c
error: could not apply 646e0f9cda11... no color ls
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply 646e0f9cda11... no color ls

```

which looks scary. If you bring up an editor, you'll see it's a typical 3-way merge conflict resolution that you may be familiar with from other source code systems (the rest of `ls.c` has been omitted):


```

<<<<<<< HEAD
#ifdef COLORLS_NEW
#include <terminfo.h>
=====
#undef COLORLS
#ifdef COLORLS
#include <termcap.h>
>>>>>> 646e0f9cda11... no color ls

```

The new code is first, and your code is second. The right fix here is to just add a `#undef COLORLS_NEW` before `#ifdef` and then delete the old changes:

```

#undef COLORLS_NEW
#ifdef COLORLS_NEW
#include <terminfo.h>

```

save the file. The rebase was interrupted, so you have to complete it:

```

% git add ls.c
% git rebase --continue

```

which tells Git that `ls.c` has changed and to continue the rebase operation. Since there was a conflict, you'll get kicked into the editor to update the commit message if necessary. If the commit message is still accurate, just exit the editor.

If you get stuck during the rebase, don't panic. `git rebase --abort` will take you back to a clean slate. It's important, though, to start with an unmodified tree.

For more on this topic, <https://www.freecodecamp.org/news/the-ultimate-guide-to-git-merge-and-git-rebase/> provides a rather extensive treatment. It is a good resource for issues that arise occasionally but are too obscure for this guide.

Switching to a Different FreeBSD Branch

If you wish to shift from `stable/12` to the current branch and if you have a deep clone, the following will suffice:

```

% git checkout main
% # build and install here...

```

If you have a local branch, though, there are one or two caveats. First, rebase will rewrite history, so you'll likely want to do something to save it. Second, jumping branches tends to encounter more conflicts. If we pretend the example above was relative to `stable/12`, then to move to `main`, I'd suggest the following:

```

% git checkout no-color-ls
% git checkout -b no-color-ls-stable-12 # create another name for this branch
% git rebase -i stable/12 no-color-ls --onto main

```


What the above does is `checkout no-color-ls`. Then create a new name for it (`no-color-ls-stable-12`) in case you need to get back to it. Then you rebase onto the main branch. This will find all the commits to the current `no-color-ls` branch (back to where it meets up with the `stable/12` branch) and then it will replay them onto the main branch creating a new `no-color-ls` branch there (which is why I had you create a place holder name).

Migrating from an Existing Git Clone

If you have work based on a previous Git conversion or a locally running `git-svn` conversion, migrating to new repository can encounter problems because Git has no knowledge about the connection between the two.

If do not have a lot of local changes, the easiest way would be to cherry-pick your changes to the new base:

```
% git checkout main
% git cherry-pick old_branch..your_branch
```

Or alternatively, you can do the same thing with rebase:

```
% git rebase --onto main master your_branch
```

If you do have a lot of changes, you would probably want to perform a merge instead. The idea is to create a merge point that consolidates the history of the `old_branch`, and the new source of truth (`main`).

We intend to publish a set of pairs of SHA1s for this, but if you are running a local conversion, you can find out by looking up the same commit that are found on both parents:

```
% git show old_branch
```

You will see a commit message, now search for that in the new branch:

```
% git log --grep="commit message on old_branch" freebsd/main
```

You would get a SHA1 on the new main branch, create a helper branch (in the example we call it `stage`) from that SHA1:

```
% git checkout -b stage SHA1_found_from_git_log
Then perform a merge of the old branch:
% git merge -s ours -m "Mark old branch as merged" old_branch
```

With that, it's possible to merge your work branch or the main branch in any order without problem. Eventually, when you are ready to commit your work back to main, you can perform a rebase to main or do a squash commit by combining everything into one commit.

WARNER LOSH is a Senior Software Engineer at Netflix and has been a FreeBSD contributor for over 20 years. Warner has improved a number of systems—for example, the boot loader—in FreeBSD. Prior to Netflix, he produced flash drives and measured atomic clocks for accuracy. His code still measures some of the clocks that create UTC!

Kernel Fuzzing with syzkaller

BY MARK JOHNSTON

If you have ever been unlucky enough to fall victim to a FreeBSD kernel panic, you would be well-justified in asking just how those sloppy kernel programmers test their code. The kernel is the backbone of the entire system and changes to it should of course have been meticulously tested before users can boot up the latest and greatest build. In our defense, however, kernel programmers work in a harsh, inhospitable environment. The FreeBSD kernel is written in C, a programming language infamous for its subtle pitfalls and lack of amenities. The kernel also has to deal with several adversaries: first, it executes and provides services to all sorts of software, some of which may have malicious goals; second, it interacts with the computer's hardware and all of its associated warts, convoluted designs and outright bugs. Many kernel developers have spent sleepless nights debugging memory corruption that ultimately was the result of buggy device firmware that overwrites system memory when prodded a certain way. Finally, like any modern OS kernel, FreeBSD's makes use of all of the CPUs available in the computer, and kernel developers have to grapple with all of the intrinsic complexity of writing efficient, scalable and correct software for multi-core systems. In short, it's a tricky problem.

FreeBSD's developers put a great deal of effort into shipping stable, well-tested releases. It is worth thinking for a while about how one might test, say, a change to an existing system call, or a new system call. System calls are in a sense the front-end of the kernel: they provide the low-level abstractions used by all programs, and the invocation of a single system call may cause the kernel to execute thousands of lines of code on the invoker's behalf. A developer adding a new system call will certainly write some test programs to verify that it behaves according to its specification, but generally it is not possible to exhaustively test all possible inputs to a lone system call. Furthermore, test programs cannot prove the absence of a bug; even if the system call produced a correct result, a bug may have corrupted a piece of kernel memory in a way that is not detectable for a long time after the fact. System calls may also interact with each other: a multi-threaded program will often execute multiple system calls simultaneously, each updating some kernel state, so our hypothetical kernel developer must think carefully about the synchronization of these calls and how the hundreds of existing system calls might interact with the one in question.

These kinds of problems are not specific to kernel programming and we have many conceptual and technological tools that let us attack the stark complexity of writing bug-free kernel code, and deliver stable FreeBSD releases with confidence. Over the past several years a new such tool, syzkaller, has been extraordinarily successful at finding severe bugs in all major operating systems, including FreeBSD.

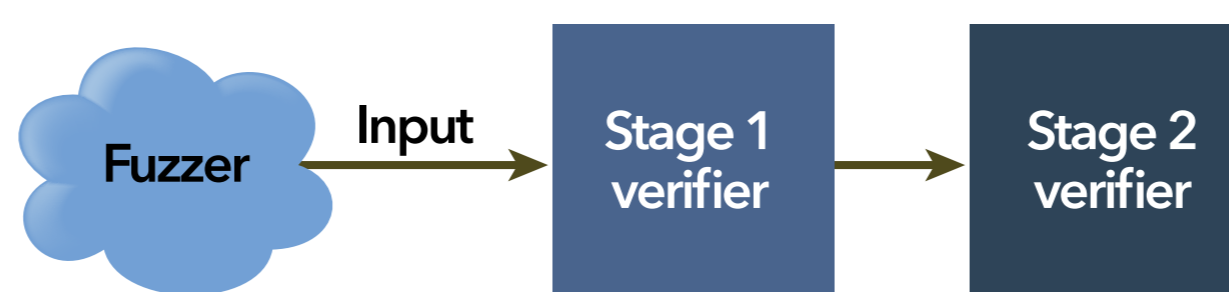
Coverage-guided Fuzzing

One important testing method for software that accepts untrusted input is fuzzing. Roughly speaking, fuzzing is the technique of programmatically generating inputs for the software under test, feeding that input to the software, and monitoring for unexpected results or side effects. This is an effective technique for finding bugs in the code that handles input validation, and has become an indispensable software testing tool. For instance your PDF reader, which you presumably use to open files found on the world wide web, will hopefully have been tested using a fuzzer among other things: the PDF specification is rather complicated and the code which parses it will be correspondingly so, making PDFs an attractive vector for malware authors. Indeed, fuzzers are often used by security researchers and malware authors to find security holes.

Fuzzing is one technique of many used to test software. One of its significant limitations is that it cannot generally verify that software is behaving correctly, only that it is not misbehaving according to some set of criteria. For instance, a fuzzer for a language parser would try to find input that causes the parser to crash, but the absence of a crash for a given input does not imply that the input was handled correctly according to the parser's specification. Fuzzers instead excel at finding corner cases and rarely executed code paths overlooked by other software testing methods and which are therefore quite likely to contain bugs. To maximize effectiveness, the software under test should use assertions and other forms of runtime checking to detect invalid states as early as possible.

Fuzzers vary in their level of sophistication. A naive fuzzer might generate purely random data and feed it directly to the software under test. While this approach may yield some fruit, it is unlikely to find anything other than very basic input validation bugs while consuming a large amount of computing resources. Consider a compiler fuzzer which simply generates random ASCII strings: most such strings are not valid programs and so will be rejected very quickly by the compiler's parser, and as a result many components of the compiler, such as optimization and code generation logic, will not be exercised. Intelligent fuzzers have some knowledge of the input format so that they can generate valid-looking inputs that pass basic verification logic. For instance, a fuzzer which aims to test an IPv6 packet processor would ensure that inputs at least start with the 4-bit version number that begins all valid IPv6 packet headers. It could achieve this by using a corpus of valid IPv6 packets as a starting point, or with some built-in knowledge of the IPv6 packet header layout, or likely some combination of the two.

A second effective optimization involves providing feedback to the fuzzer. A naive fuzzer would, in a loop, generate an input, feed it to the software under test, and wait for either a crash or graceful termination of the program. It has no general way to determine whether a given input helped improve test coverage of the software or not, and so cannot focus on "interesting" inputs. Consider a fuzzer target which performs input validation in two stages:



Stage 1 might simply verify that various components of the input have the correct length, while stage 2 verifies that the individual components contain valid values. If most input fails stage 1 validation, then stage 2 validation is left largely untested. However, if the fuzzer can

dynamically learn which inputs pass stage 1 validation, it can improve its coverage of stage 2 validation by prioritizing inputs known to pass stage 1.

There are multiple ways for a fuzzer to obtain feedback. For instance, it might measure the amount of time taken to process a given input and use a heuristic which discards inputs that are processed very quickly, under the assumption that such inputs are failing basic validity tests. Another technique, used by state-of-the-art fuzzing frameworks such as libFuzzer, AFL and syzkaller, measures code coverage. By leveraging software instrumentation facilities, a coverage-guided fuzzer can “trace” the code paths executed when processing a given input, and use that information to try and generate inputs which uncover previously unexecuted code. Fuzzers use this technique to achieve high levels of test coverage very efficiently, and indeed, the aforementioned fuzzers have been used to find thousands of severe bugs in all sorts of software projects, even those considered mature and well-tested.

syzkaller

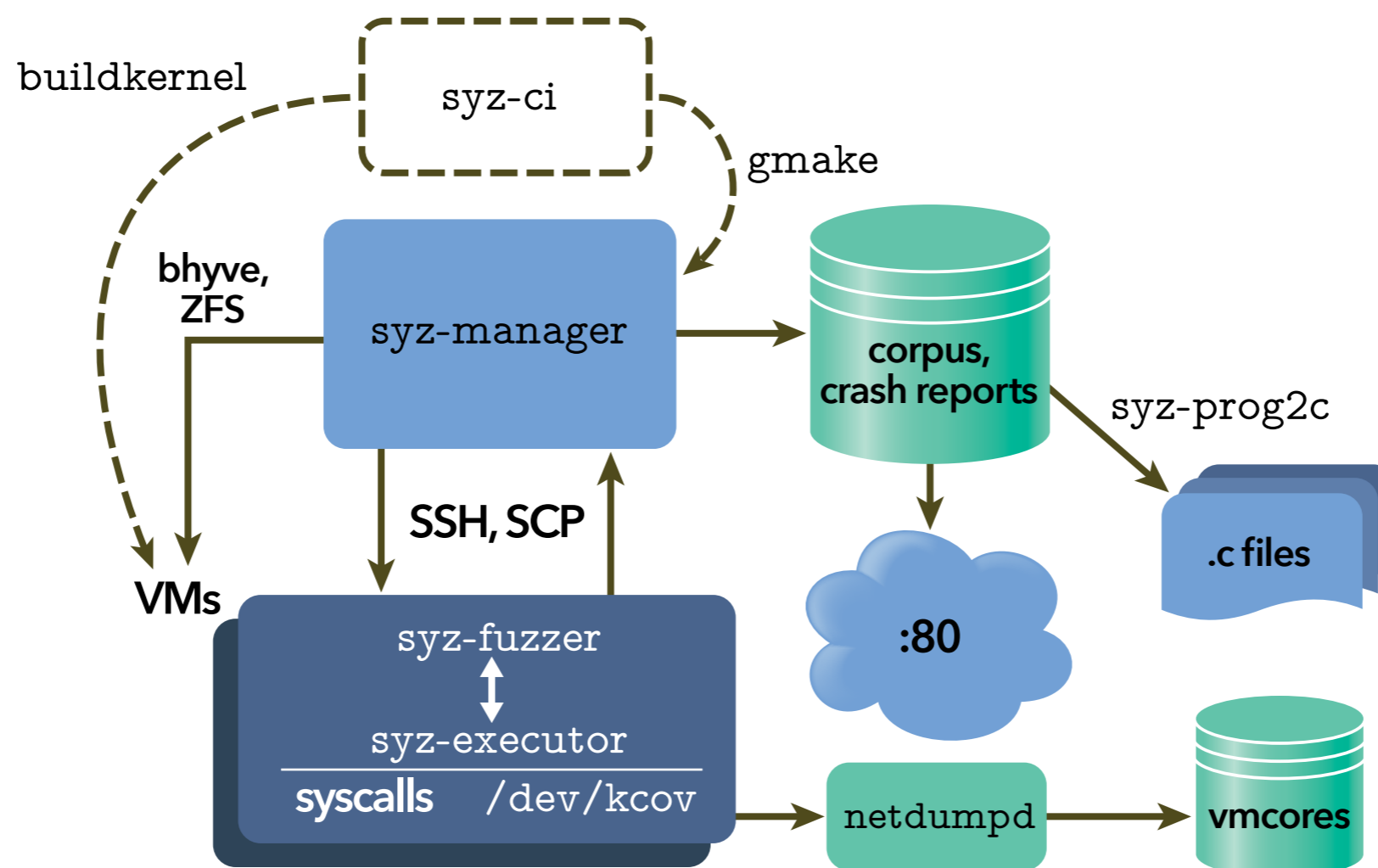
Operating system kernels handle input from a variety of untrusted sources: unprivileged processes will invoke system calls and may be trying to take control over the computer; a system connected to the internet processes network packets from untrusted sources; the kernel may be asked to mount a file system with invalid contents; a computer may support pluggable peripheral devices which can communicate directly with the kernel. In short, a useful kernel presents a massive attack surface, and years of high-profile kernel security holes show that there is much room for improvement among popular operating systems. syzkaller seeks to improve this state of affairs.

[syzkaller](#) is an open-source coverage-guided kernel fuzzer by Dmitry Vyukov. It originally targeted Linux but has since expanded to support nearly a dozen other operating systems. syzkaller is sometimes described as a system call fuzzer but is flexible enough to target other operating system interfaces; for example, it has been used to fuzz Linux’s USB stack and has found dozens of bugs in the [USB subsystem alone](#). The details are complicated but the idea is simple: generate a program which invokes one or more system calls (or injects a packet into the network, etc.), run it, and check to see if the system diagnosed an error (for example by panicking). If not, collect kernel code coverage information and decide whether to try iterating upon the previous test program, or start anew. If so, collect information about the crash and try to discover a minimal test case that triggers the crash.

syzkaller is written mostly in Go and consists of a dozen or so loosely-coupled programs, all prefixed with `syz-`, that together provide a self-contained system to do all of the following:

- Start and run a set of operating system instances, typically in virtual machines.
- Monitor those virtual machines for crashes or other diagnostic reports, typically by monitoring console output.
- Generate programs to run under the target operating system, using coverage information to drive decisions about what to try next, and run them.
- Maintain a database of observed crashes and diagnostic reports, to try and classify distinct bugs found.
- Provide a web dashboard displaying statistics, code coverage information, and observed crashes and their reproducers if any.
- Periodically update itself and the operating system under test without any manual intervention.
- Attempt to bisect new crashes down to the commit introducing the bug.

The high-level components of this system as it might run on FreeBSD are depicted here:



Thanks to Google, the syzkaller developers provide numerous public syzkaller instances running in continuous integration mode, wherein syzkaller updates itself and the target operating system regularly. These “syzbot” instances find bugs in the latest builds of their targets, so regressions are reported quickly and completely automatically. The [FreeBSD instances](#) have found numerous bugs and reproducers, enabling developers to both diagnose and fix bugs quickly and to provide higher-quality releases.

kcov(4)

syzkaller is not the first kernel fuzzer but is undoubtedly the most prominent. Newsgroup posts from the early 1990s describe programs which bombard UNIX kernels with random system calls to great effect. Peter Holm’s stress2 test suite for FreeBSD performs some targeted fuzzing of certain system calls (among many other things). However, syzkaller introduces a key innovation in its use of code coverage to drive test case generation. This makes use of the `kcov(4)` kernel subsystem, written also by Dmitry Vyukov for Linux but later ported to other operating systems by their respective developers. While syzkaller does not strictly require code coverage information, it is much more effective with this extra feedback from the kernel.

In FreeBSD, `kcov(4)` is a wrapper for [LLVM’s SanitizerCoverage](#). Sanitizers are compiler features which inject bits of code enabling certain types of introspection into the compiled result. For example, LLVM’s AddressSanitizer inserts special function calls before every single memory access by the generated machine code; the calls can be used to determine whether the memory access is somehow invalid, for example because it corresponds to a use-after-free. This provides powerful bug-detection facilities similar to Valgrind but using different mechanisms: Valgrind works by running the unmodified target program in a software virtual machine which can intercept memory accesses and perform validation, whereas sanitizers are implemented by the compiler itself and require special compilation flags. Sanitizers and Valgrind both introduce significant performance overhead and are generally used only in testing scenarios. Sanitizers have the added benefit that they can sometimes be used to validate a kernel, while Valgrind cannot.

SanitizerCoverage inserts function calls according to the control flow of the generated code. Most CPU instructions do not modify control flow: once the instruction is completed, the CPU fetches and executes the subsequent instruction from RAM. Control flow instructions cause

the CPU to jump to a different address and begin execution there instead. This is how basic programming language constructs like if-statements, loops and goto work under the hood. A compiled program can thus be broken down into a set of “basic blocks,” where a basic block is a sequence of non-control flow instructions. Following the end of each basic block is a control flow instruction. Then, if the goal is to figure out which pieces of code get executed in response to a given input, it suffices to trace out which basic blocks get executed.

SanitizerCoverage, roughly speaking, inserts function calls in between each basic block, as in this machine code for the FreeBSD kernel function `vm_page_remove()`:

```

////////////////////////////////////
<+0>:    push   %rbp
<+1>:    mov    %rsp,%rbp
<+4>:    push   %r15
<+6>:    push   %r14
<+8>:    push   %rbx
<+9>:    push   %rax
<+10>:   mov    %rdi,%rbx
<+13>:   callq  0xffffffff81167dc0 <__sanitizer_cov_trace_pc>
<+18>:   mov    %rbx,%rdi
<+21>:   callq  0xffffffff815b7c50 <vm_page_remove_xbusy>
<+26>:   mov    %eax,%r14d
<+29>:   mov    $0x1,%ecx
<+34>:   xor    %esi,%esi
<+36>:   mov    $0x2,%eax
<+41>:   lock  cmpxchg %ecx,0x54(%rbx)
<+46>:   setne %r15b
<+50>:   sete  %sil
<+54>:   xor    %edi,%edi
<+56>:   callq  0xffffffff81167ea0 <__sanitizer_cov_trace_const_cmp1>
<+61>:   test   %r15b,%r15b
<+64>:   jne   0xffffffff815b78f9 <vm_page_remove+73>
<+66>:   callq  0xffffffff81167dc0 <__sanitizer_cov_trace_pc>
<+71>:   jmp   0xffffffff815b790d <vm_page_remove+93>
<+73>:   callq  0xffffffff81167dc0 <__sanitizer_cov_trace_pc>
<+78>:   movl  $0x1,0x54(%rbx)
<+85>:   mov   %rbx,%rdi
<+88>:   callq  0xffffffff81107950 <wakeup>
<+93>:   mov   %r14d,%eax
<+96>:   add   $0x8,%rsp
<+100>:  pop   %rbx
<+101>:  pop   %r14
<+103>:  pop   %r15
<+105>:  pop   %rbp
<+106>:  retq
////////////////////////////////////

```

Here the `__sanitizer_cov_trace_*` function calls are inserted by SanitizerCoverage and can be implemented by the kernel. `kcov(4)` works by implementing these functions.

In typical usage, a user program allocates a buffer to store coverage information, opens `/dev/kcov` and uses `ioctl(2)` to map the buffer into the kernel and to enable tracing of the current thread. When the thread subsequently enters the kernel, perhaps to execute a system call, the coverage tracing hooks log the address of each basic block into the buffer. When the thread disables tracing, again using an `ioctl(2)` call, it can make use of the information provided in the buffer. For instance, the recorded addresses could be piped into the `addr2line(1)` program to find the file and line number of the traced C code. The `kcov(4)` manual page contains the details of this `ioctl(2)` interface as well as some example code.

syzlang

Earlier we pointed out that fuzzers work better when they have some knowledge of the software’s input format, rather than treating it as a black box. While `syzkaller` could theoretically invoke system calls without any knowledge of what they do or what parameters they take — using only coverage information to try and “learn” which parameter values result in more code execution — this is both inefficient and potentially counter-productive. Consider what happens if a fuzzer invokes `kill(-1, SIGKILL)`: the kernel will do what it was asked to do and immediately kill the fuzzer process.

Unfortunately, system call interfaces cannot be discovered programmatically. In other words, there is generally no way to ask the kernel to describe the set of system calls that it implements. Even a set of C function prototypes omits many important details. Consider `read(2)`:

```
read(int fd, void *buf, size_t nbytes);
```

First, `fd` is here represented by an integer, but really must be a valid file descriptor as well. There are 4,294,967,295 possible values for `fd` and all but a tiny fraction of them are invalid. Second, it is not clear what the kernel is expected to do with `buf`: is the kernel supposed to read data from that address, or write to it, or both, or neither? Third, `nbytes` is supposed to represent the size of the buffer `buf` but the prototype gives no indication that these two parameters are related; the C language is simply not expressive enough to do so. If you are familiar with `ioctl(2)`, think for a bit about how it makes a bad situation even worse.

To solve these problems, `syzkaller` introduces [syzlang](#): a language for modeling the kernel’s programming interfaces. It is flexible enough to define data layouts that are binary-compatible with C types, and expressive enough to describe inter-related C parameters, among other things. In `syzlang`, the `read(2)` prototype above becomes:

```
read(fd fd, buf buffer[out], count len[buf])
```

Unlike C (but like Go), the parameter name comes first, followed by the type. Right away we can see that this definition provides more information than the C prototype: there is an `fd` type, to distinguish file descriptors from plain integers; `buf` is a pointer to a buffer mapped in the caller’s address space, and the `[out]` annotation signifies that it is an “out-parameter,” i.e., the kernel is supposed to write data to the buffer; `count` is the length, in bytes, of the buffer `buf`.

The use of specialized types to represent file descriptors and other kernel resources is important for generating programs that do “interesting” things since many system calls take as input the results of previous system calls. For example, to read data from a file a program might execute the following sequence of system calls:


```

const size_t len = 4096;
void *buf = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_ANON, -1, 0);
int fd = open("/tmp/foo", O_RDWR);
read(fd, buf, len);
close(fd);
munmap(buf, len);

```

Note that the results of the `mmap(2)` and `open(2)` calls are used as input to subsequent calls. Fuzzing `munmap(2)` and `close(2)` does not make much sense without earlier calls to `mmap(2)` and `open(2)`, so `syzlang` models the corresponding resources and `syzkaller` creates chains of system calls using these relationships to guide its choices.

The use of `mmap(2)` also illustrates a need to define the set of valid values for “flag” parameters such as the third and fourth arguments. `syzlang` has a built-in `flags` type for this:

```

mmap(addr vma, len len[addr], prot flags[mma_prot],
     flags flags[mmap_flags], fd fd, offset fileoff)

mmap_prot = PROT_EXEC, PROT_READ, PROT_WRITE
mmap_flags = MAP_SHARED, MAP_PRIVATE, MAP_ANON, ...

```

The fuzzer will chose zero or more flag values when creating an argument for a flag parameter.

`syzlang` can also use subtyping to more accurately model system call interfaces. Network connections and open files are both represented by file descriptors in the system call interface, but many system calls accept only certain types of file descriptors. For instance, one of `sendfile(2)`’s parameters is a file descriptor corresponding to a network connection on which to send a file’s data. Such descriptors are created using `socket(2)` or `socketpair(2)`. Passing a descriptor for a regular file here will fail, so to save the fuzzer time we can define a new `socket` type. First we have the definition for `fd`:

```

resource fd[int32]: 0xffffffffffffffff, AT_FDCWD

```

This derives `fd` from the built-in integer type `int32` and defines a couple of special values: `-1`, for system calls where a parameter of type `fd` is optional (such as `mmap(2)`), and `AT_FDCWD`, used by `openat(2)` and similar system calls. Then we can define a derived resource type for sockets:

```

resource sock[fd]

socket(...) sock
sendfile(fd fd, s sock, ...)

```

A similar trick is used for system calls where layout of a parameter depends on the value of another parameter. `ioctl(2)` is the prime example of this, but `fcntl(2)`, `setsockopt(2)` and `bind(2)` behave similarly. For example, `pf(4)` defines a large set of `ioctl` commands, but each has its own argument type. We can describe them precisely in `syzlang`:


```
resource fd_pf[fd]
openat$pf(fd const[AT_FDCWD], file ptr[in, string["/dev/pf"]], ...) fd_pf
ioctl$DIOCRADDTABLES(fd fd_pf, cmd const[DIOCRADDTABLES], arg ptr[in, pfioc_table])
```

Here we define a new `fd` type which corresponds to a file descriptor for `/dev/pf`. Then, special “flavours” of `openat(2)` and `ioctl(2)` describe how the fuzzer can open a `pf(4)` device and issue the `DIOCRADDTABLES` `ioctl`, used by `pfctl(8)` to define an address table.

`syzlang` definitions are compiled at build-time into tables used by `syzkaller`’s fuzzer. `syzkaller` maintains its own internal representation of system calls and their parameters, and its representation of a program is simply a list of calls and parameters. All fuzzing is performed using these representations; when a reproducer for a kernel bug is found, it is finally translated into a standalone C program. This can be done manually using `syz-prog2c` but this is typically handled automatically.

New system call definitions are added frequently since in most cases `syzkaller` is still playing catch-up: the existing kernel interfaces are massive and defining them in `syzlang` requires time and effort. In particular, many components of FreeBSD are not yet described by `syzlang` and therefore do not get tested by `syzbot`. Adding to the FreeBSD `syzlang` definitions is a great way to start contributing to the `syzkaller` project and to help ensure that FreeBSD gets as much test coverage as possible.

syz-manager

So far we have looked at the mechanisms by which `syzkaller` addresses the generic technical problems faced by all fuzzers: obtaining feedback from the target software (via `kcov(4)`) and describing kernel interfaces (with `syzlang`). Now we can look more at some of the machinery required to fuzz an operating system kernel.

A fuzzer’s goal is to “exercise” the code being tested, so it needs an environment in which to execute the code and provide input. Fuzzing a kernel poses some extra challenges: the fuzzer needs to run in the same system as the kernel being tested, so if it achieves its goal and triggers a kernel panic, all of the fuzzer’s state will be lost. `syzkaller`’s solution is to run the target kernel in a set of virtual machines which can be safely wiped without losing anything important. These VMs can run on the same host as `syzkaller` or in cloud environments such as Google Compute Engine.

`syz-manager` is the main front-end program of `syzkaller`. It takes a configuration file as input and starts a number of VMs according to the configuration. It automatically installs and starts the fuzzer programs in each VM instance and communicates with them using an RPC interface over SSH. `syz-manager` also monitors the VM consoles to detect crashes. When a crash occurs, the VM is automatically re-created. VMs are restarted periodically even in the absence of a crash; one reason for this is to enable “corpus rotation.”

`syz-manager` also maintains the instance’s crash database. When a crash is discovered, `syz-manager` adds an entry to the crash database. Crashes are identified by the panic message printed by the kernel, and when a new crash is found, `syz-manager` dedicates a subset of the VMs to spend time attempting to reproduce the crash. Programs that were executed leading up to the crash are replayed, and if a crash can be reproduced, `syzkaller` also attempts to find a minimal reproducible for the crash to add to the crash database. Finally, `syzkaller` attempts to translate the crashing program into a standalone C program, making it easy for developers to debug crashes without needing a `syzkaller` installation on hand.

Most of the work to set up syzkaller involves creating a VM image containing the target operating system. The VM's kernel should have `kcov(4)` enabled, and an SSH key for the root user must be installed. The VM image is used as template; when `syz-manager` starts, it creates a snapshot of the image before starting VMs, and each VM uses a local copy of the template. When a VM is restarted, it gets a fresh copy of the template image, so any damage done by the fuzzer is discarded.

`syz-manager` supports a number of different hypervisors and cloud APIs. On FreeBSD one can use `bhyve` as the back-end hypervisor. To create a VM image template `syz-manager` uses ZFS clones, since `bhyve` lacks support for creating disk image snapshots. Upon starting up `syz-manager` also starts a web server, providing a dashboard containing statistics and code coverage information, deduplicated crash reports, and crash reproducers. A sample `syz-manager` configuration looks like this:

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
{
  "target": "freebsd/amd64",
  "http": "0.0.0.0:8080",
  "workdir": "/data/syzkaller",
  "image": "/data/syzkaller/vm.raw",
  "syzkaller": "/home/markj/go/src/github.com/google/syzkaller",
  "procs": 4,
  "type": "bhyve",
  "ssh_user": "root",
  "sshkey": "/data/syzkaller/id_rsa",
  "kernel_obj": "/usr/obj/usr/home/markj/src/freebsd/amd64.amd64/sys/SYZKALLER",
  "kernel_src": "/",
  "vm": {
    "bridge": "bridge0",
    "count": 32,
    "cpu": 2,
    "hostip": "169.254.0.1",
    "dataset": "data/syzkaller"
  }
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

This configuration specifies 32 VM instances; in general, more VMs is better since each VM runs test programs in parallel with the others. The `cpu` parameter defines the number of virtual CPUs given to each VM, and the `procs` parameter defines the number of fuzzer processes that will run in each VM. Having multiple virtual CPUs and fuzzer processes improves the odds of finding certain types of bugs, but over-subscribing the host may decrease the effectiveness of fuzzing by making it hard to reproduce bugs. It is reasonable to configure one or two virtual CPUs per host CPU, but more than that is probably too many.

See the FreeBSD/syzkaller [documentation](#) for details on how to build and configure your own syzkaller setup. With a configuration file written, syzkaller can be started with:

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
# syz-manager -config /path/to/config
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

When using `bhyve`, syzkaller needs to run as root in order to create virtual machines. It is possible to run syzkaller in a jail with some effort; some ongoing work aims to make this simpler.

If you wish to run your own private syzkaller instance, do be prepared to be patient — now that much of the low-hanging fruit has been fixed, it can take days for syzkaller to find a new kernel bug.

Fuzzing the Kernel

Now that we have encountered most of the machinery that syzkaller uses to fuzz operating system kernels, we are equipped to start looking at the brains of syzkaller.

Aside from the crash database, syzkaller's main piece of persistent state consists of the corpus: a representative set of programs whose execution generates coverage of the kernel. The corpus — initially empty — is effectively a seed for the fuzzer: a new test program is generated by taking a program from the corpus, mutating it in some small way, and checking to see if previously uncovered kernel code was covered by the new program. If so, the program may be added to the corpus and subsequently used as the starting point for other programs. Algorithmically, the fuzzer does nothing except try to increase the size of the corpus. Many heuristics are applied to try and make this more effective for syzkaller's real purpose — finding bugs — but the core idea is very simple.

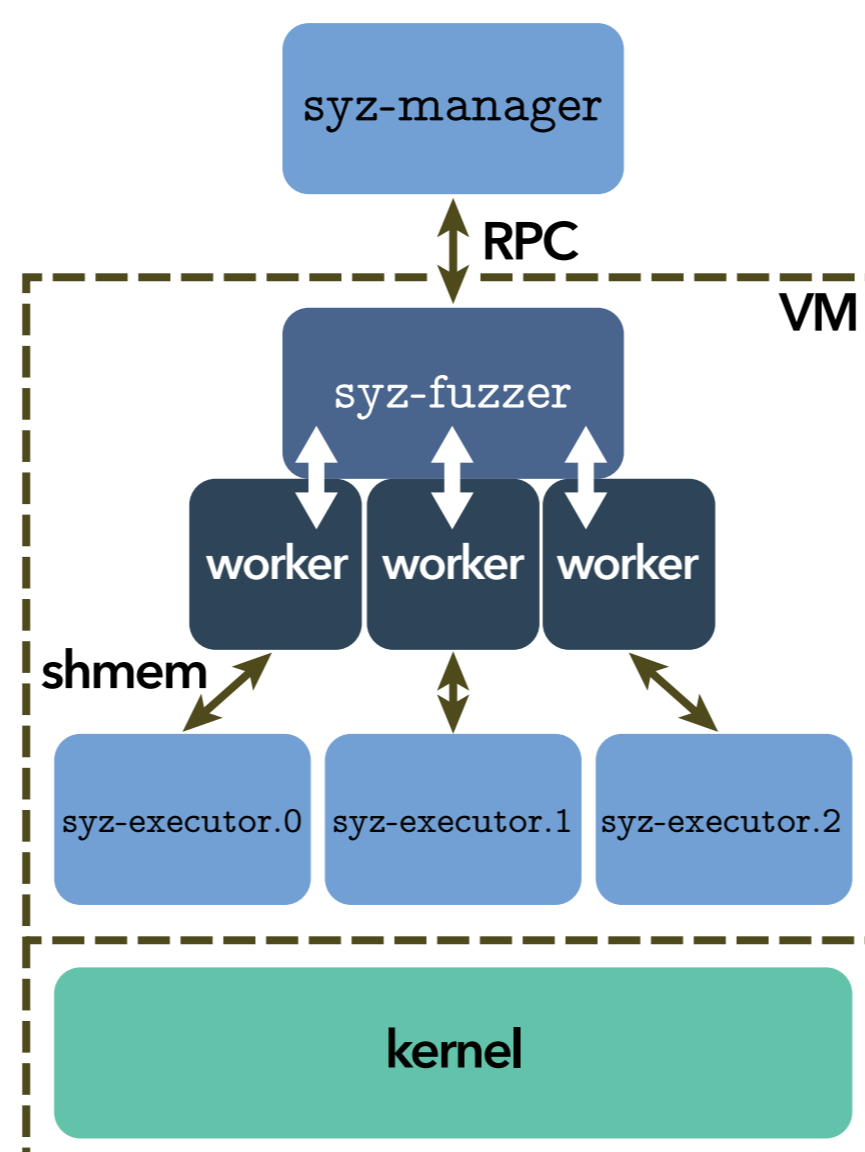
Several types of program mutations are possible. The fuzzer might:

- splice several programs together
- insert a new system call
- remove an existing system call
- modify one of the parameters to a call in the program

If the corpus is empty, the fuzzer will create a new program by generating a random list of system calls with randomly selected arguments. This is also done periodically even when the corpus is non-empty.

Inside each VM managed by syzkaller runs a pair of programs, `syz-fuzzer` and `syz-executor`, which communicate using a shared memory interface. As their names suggest, `syz-fuzzer` generates test programs and `syz-executor` actually executes them. `syz-fuzzer` and `syz-manager` coordinate using a simple RPC protocol; since `syz-fuzzer` generates programs which may crash the VM, it relies on `syz-manager` to store the corpus.

`syz-fuzzer` is started by `syz-manager` over SSH. When it begins, it establishes an RPC connection with `syz-manager` and creates a number of work queues, each of which is managed by a thread (really, a goroutine). The worker threads each spawn a `syz-executor` instance and immediately begin fuzzing, yielding the following picture:



Worker threads perform most of the work of adding to the corpus: they generate new programs and mutate existing ones. They also handle special types of work:

- Triage: when a program appears to generate new coverage it is placed in the triage queue for further refinement. The triage step tries to determine whether the program behaves consistently (i.e., re-runs generate the same coverage info), and if so, tries to minimize the size of the program while maintaining its coverage.
- Smashing: when a program has been triaged and appears worthy of being added to the corpus, the worker spends extra time mutating it to look for new coverage.
- Candidate processing: `syz-manager` may send candidate programs to the fuzzers in some cases. The worker executes them, potentially creating triage or smash work.

In steady-state operation, `syz-fuzzer` uses two RPCs to communicate with the manager: `Poll` and `NewInput`.

`Poll` is invoked periodically to update the fuzzer's snapshot of the corpus and global coverage information, and to collect candidate programs for fuzzing. It also serves to re-process the existing corpus when `syzkaller` starts up; a typical `syzkaller` installation will periodically update itself and the target kernel, and must subsequently restart. The saved corpus is immediately distributed among the fuzzers for execution and triage since the updated kernel may handle existing corpus items differently from when they were last evaluated.

`NewInput` is used to send triaged programs back to `syz-manager` as possible candidates for the global corpus. `syz-manager` will reject new inputs in some cases, for instance to avoid blowing up the size of the corpus, or if another fuzzer had already discovered a similar program. If accepted, new corpus programs eventually become visible to other fuzzer instances via `Poll`.

Unfortunately, code coverage is not an ideal metric: 100% code coverage of a program does not preclude the existence of detectable bugs, especially in multi-threaded code such as a modern operating system kernel. Optimizing for an imperfect metric tends to yield suboptimal results — we (hopefully!) do not evaluate programmers based on the number of lines of code they have written. In `syzkaller`'s case, valuable test programs may be discarded if they do not add to the corpus' code coverage. To try and alleviate this problem, `syzkaller` performs corpus "rotation": some system calls and corpus programs are hidden from individual fuzzers to force them to find programs with equivalent coverage but hopefully new characteristics. This can result in duplicated effort but helps to ensure that the system does not become "stuck" by finding local maxima.

Program Execution

To round off our examination of `syzkaller` a look at `syz-executor` is in order. `syzkaller` uses an internal representation of system call programs for the purpose of fuzzing, but of course has to actually run them somehow. `syz-executor` is the component of `syzkaller` that performs this task; unlike the rest of `syzkaller`, it is written in C++.

The executor is spawned by `syz-fuzzer` worker threads and uses a simple shared memory interface to communicate with the worker. It first creates a pool of threads to actually execute system calls, and then opens `/dev/kcov` and uses `ioctl(2)` to enable collection of code coverage information that is returned to the worker. Quite a lot of additional initialization may happen at this point, depending on how `syzkaller` is configured. For instance, the executor may enter a software sandbox in an attempt to limit the effects of the test program: a program which sends signals to unsuspecting processes is likely to wedge the VM and trigger a

costly timeout and restart. It may also initialize devices or network facilities as part of a targeted fuzzing regime.

When it comes time to execute system calls, `syz-executor` iterates over the call list and assigns an idle thread to each one, waiting for threads to become free if necessary. Initially, the main thread waits for a short period after each call is dispatched. Once the input program has finished, it is executed a second time in “collision mode”: rather than waiting for a short period after each call is dispatched, pairs of system calls are allowed to execute concurrently, helping to trigger race conditions in the kernel that would otherwise be left unexercised.

Actual system call execution is achieved using the handy `syscall(2)` system call, a generic system call which takes a system call number and variable list of parameters as arguments. Internally the kernel uses the system call number to route the call to the requested handler. The system call’s result is also recorded for use in prioritizing and triaging programs: a successful system call is weighted more favorably than a failed system call.

Conclusion

If you managed to get this far, please don’t stop here! `syzkaller` is the subject of quite few talks, articles and even research papers — check out `syzkaller`’s [documentation](#) for some curated links. This article only scratches the surface of `syzkaller`’s internals, and the sources are as usual the authoritative reference on how `syzkaller` actually works.

Fuzzing is a fascinating subject and there is a certain thrill to watching a fuzzer in action — particularly when it finds bugs in your favorite operating system. We encourage you to give it a try.

MARK JOHNSTON is a contractor and FreeBSD src committer based in Toronto, Canada. He is particularly interested in kernel debugging and in finding new ways to help improve the stability of FreeBSD. In his spare time he enjoys cooking, playing Bach’s cello suites, and impeding his productivity by experimenting with custom keyboard layouts.

Thank you!

The FreeBSD Foundation would like to acknowledge the following companies for their continued support of the Project. Because of generous donations such as these we are able to continue moving the Project forward.



FreeBSD
FOUNDATION

Are you a fan of FreeBSD? Help us give back to the Project and donate today! freebsd.foundation.org/donate/

Please check out the full list of generous community investors at freebsd.foundation.org/donors/

Uranium

Koum Family Foundation

Iridium

arm

NGINX

NetApp

Platinum

NETFLIX

Gold

JUNIPER
NETWORKS

Silver

BECKHOFF

Microsoft

moz://a

vmware



STORMSHIELD

Tarsnap



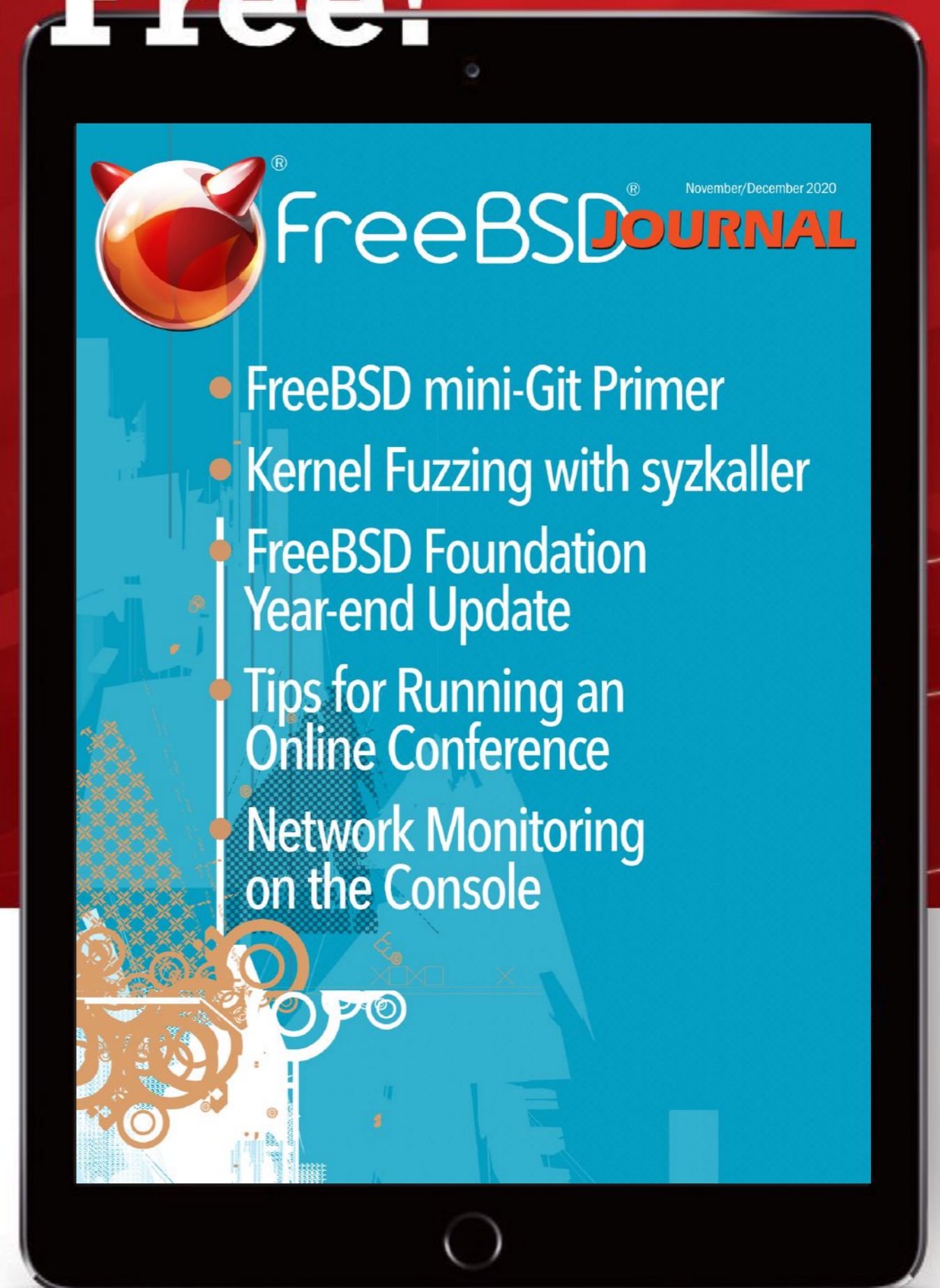
FreeBSD[®] JOURNAL

The FreeBSD Journal is Now Free!

Yep, that's right Free.

The voice of the FreeBSD Community and the BEST way to keep up with the latest releases and new developments in FreeBSD is now openly available to everyone.

DON'T MISS A SINGLE ISSUE!



2021 Editorial Calendar

- Case Studies (January-February)
- FreeBSD 13 (March-April)
- Security (May-June)
- Desktop/Wireless/Graphics (July-August)
- Cloud (September-October)
- Embedded (November December)

Find out more at: freebsd.foundation/journal

FreeBSD Foundation Year-end Update

BY DEB GOODKIN

The Foundation has faced some challenging years during my tenure, but I'd say 2020 is hands down the winner. Running a non-profit is challenging in itself. However, I can easily say, running a donation-driven, non-profit whose sole purpose is to support a free and open-source computer operating system, during a pandemic, is extremely difficult! That being said, I could not be prouder of what we've accomplished this past year to help make FreeBSD the secure, reliable, and high-performance operating system you rely on.

Although it's been a rough year setting everyone up to work from home (while remaining productive), dealing with childcare issues, and worrying about how Covid-19 might affect

us, our family, and friends, we were still able to accomplish more than we ever have in the past!

For over 20 years, we've been here to support the FreeBSD community and Project around the globe. With the pandemic preventing in-person events this year, we recognized the growing need for us to step in and help connect the community. We know that FreeBSD contributors thrive on meeting face-to-face and that these opportunities are what drive so much work on the Project. With that in mind, we set out to determine what we could do to better help connect the community.

We pulled together as a team to provide more online/virtual opportunities for the community and beyond and to encourage more people to engage with and join our community. This included:

- Producing the highly successful FreeBSD Fridays series, which provides introductory FreeBSD talks and workshops on various areas of the operating system.
- Continuing to bring FreeBSD to new conferences and other venues such as podcasts and webinars.
- Creating more educational content, including new and updated how-to guides, case studies to show how companies are using FreeBSD, and blog posts highlighting work going on in the Project.
- Organizing and running the first ever virtual Vendor Summit that allowed more developers and commercial users from around the world to engage with each other.

Meanwhile, we funded a record-breaking number of software projects. These included LLDB backend improvement and target server, vulnerability mitigations and proactive security, Linuxu-

For over 20 years,
we've been here
to support
the FreeBSD
community and
Project around
the globe.

2020 FreeBSD Foundation Year-end Update

lator application compatibility improvements, WiFi and graphics infrastructure improvements, 5x if_bridge performance increase, Git migration support, and pkgbase development.

But that's not all! Our internal staff worked on code reviews, bug triage, 3rd-party CI integration, arm64 support, improvements and enhancements to many subsystems including the x86 pmap layer, rtd and kernel ELF loader, threading library, RISC-V, Capsicum, build system, tool chain and FreeBSD-update among others, security issues, vulnerability mitigations, Syzkaller kernel code coverage, network stack stability, new kernel interfaces, machine-dependent optimizations, docs and manpages, DTrace bug fixes, Valgrind port, and much more!

In 2021, we will continue to support software development work and online/virtual opportunities. In fact, if we meet our fundraising goal, we will be increasing our FreeBSD support by growing our software developer team to make sure FreeBSD stays not only relevant over the next few decades, but also serves as a leader in secure and trusted operating systems.

We know this is a difficult time for everyone. If you are able, please consider [supporting our efforts by giving a financial contribution](#). Your donation makes a direct impact on the FreeBSD Project. Thank you to all of you who have already made a financial contribution this year. We are grateful for your support.

DEB GOODKIN is the Executive Director of the FreeBSD Foundation. She's thrilled to be in her 16th year at the Foundation and is proud of her hardworking and dedicated team. She spent over 20 years in the data storage industry in engineering development, technical sales, and technical marketing. When not working, you'll find her on her road or mountain bike, running, hiking with her dogs, skiing the slopes of Colorado, or reading a good book.

Write
For Us!

Contact Jim Maurer
with your article ideas.
(jmaurer@freebsdjournal.com)





The FreeBSD Project is looking for

- Programmers • Testers
- Researchers • Tech writers
- Anyone who wants to get involved

Find out more by

Checking out our website

freebsd.org/projects/newbies.html

Downloading the Software

freebsd.org/where.html

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!

Already involved?

Don't forget to check out the latest grant opportunities at freebsd.foundation.org

Help Create the Future. Join the FreeBSD Project!

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

The FreeBSD Community is proudly supported by



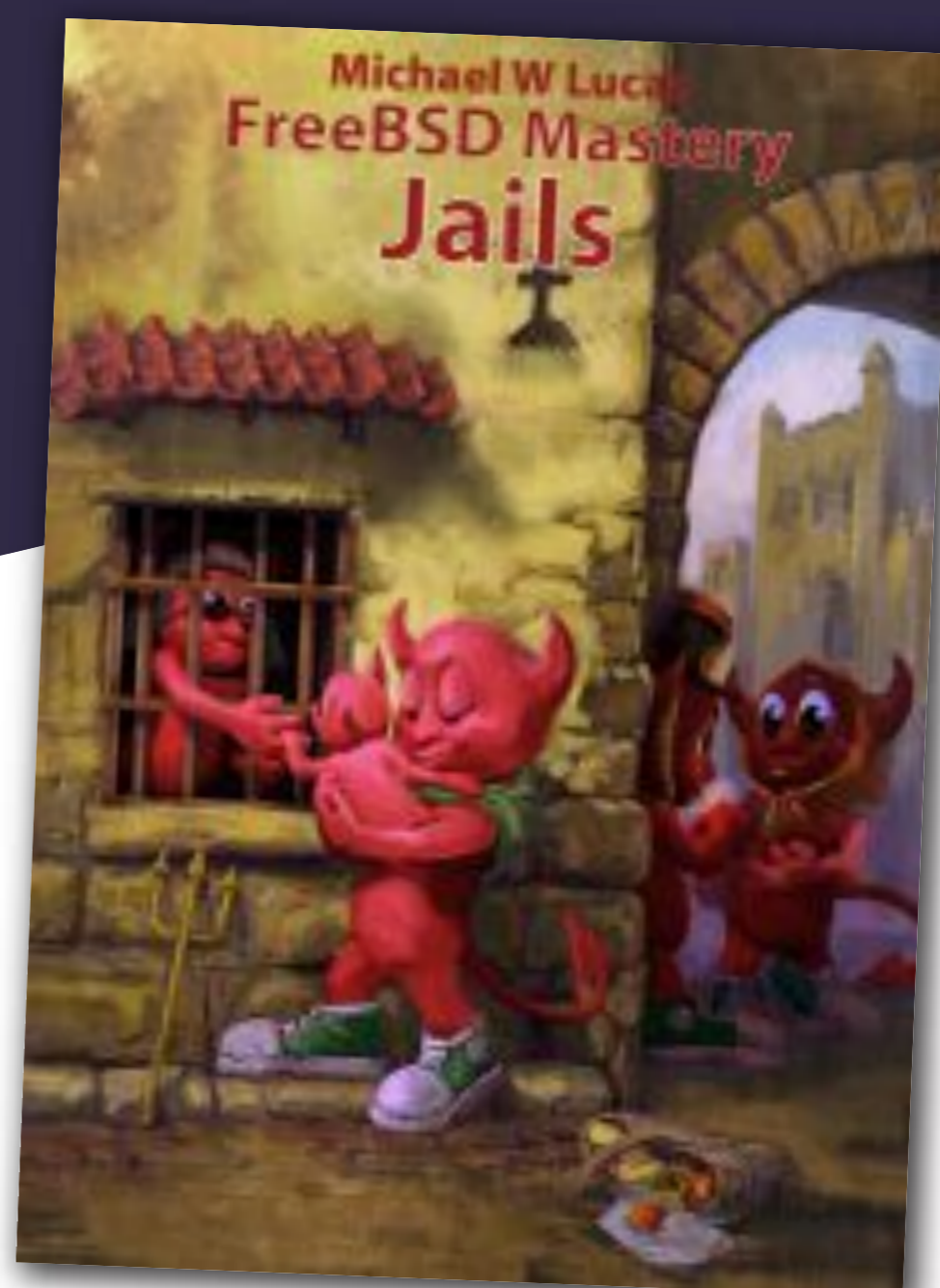
Jails ARE FreeBSD'S MOST LEGENDARY FEATURE:

KNOWN TO BE POWERFUL, TRICKY TO MASTER,
AND CLOAKED IN DECADES OF DUBIOUS LORE.

FreeBSD Mastery: Jails cuts through the clutter to expose the inner mechanisms of jails and unleash their power in your service.

Confine Your Software!

- * Understand how jails achieve lightweight virtualization
 - * Understand the base system's jail tools and the iocage toolkit
 - * Optimally configure hardware
 - * Manage jails from the host and from within the jail
 - * Optimize disk space usage to support thousands of jails
 - * Comfortably work within the limits of jails
 - * Implement fine-grained control of jail features
 - * Build virtual networks
 - * Deploy hierarchical jails
 - * Constrain jail resource usage.
- ...**And much, much more!**



FreeBSD Mastery: Jails BY MICHAEL W LUCAS **Available at All Bookstores**



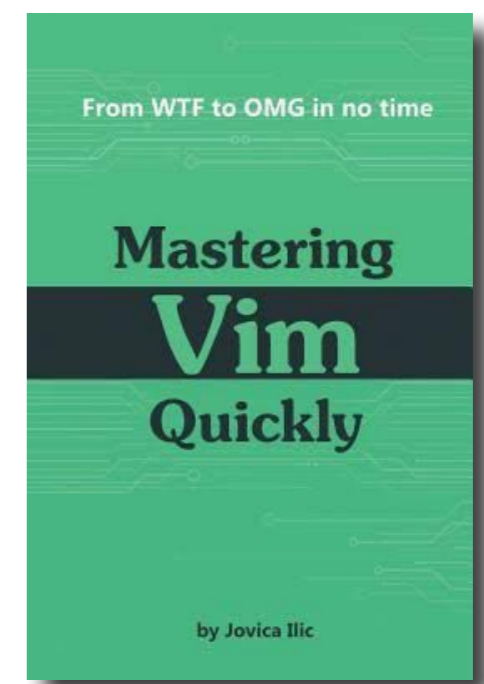
BOOK REVIEW

Mastering Vim Quickly

BY BENEDICT REUSCHLING

It's no secret that vi is the most common text editor on Unix. Omnipresent, yet difficult for beginners who don't understand the underlying philosophy. Vim (vi improved) is an enhancement of vi with the basics still present in the editor but easier to approach. You've probably seen someone doing amazing things in vim with just a few keypresses—and you may wonder how that was done.

Jovica Ilic's book was started because of that WTF feeling and it intends to teach people do these OMG editor productivity skills. There are many books on vim, but with roughly 140 pages, this one is the thinnest I've seen. But don't be fooled by its brevity, it is packed with tons of advice and examples on every page! It is written precisely for those who have shied away from vim until now or who only know some rudimentary tasks like opening, inserting, saving, and (most important of all) exiting the editor. Within a few pages, you learn how vim works and the concepts behind why it does things the way it does.



The author teaches you the “language” of vim to search within text, delete or add words by just knowing a few reusable keywords. That already gives you an edge, but the good stuff does not stop there. Common options for your .vimrc (the editors configuration file) are discussed and demonstrated. Much like a Michael W. Lucas book, it does not contain a single screenshot, yet succeeds in explaining everything using text and examples. Self-published, the book may contain the occasional typo, but that did not sour my reading experience at all. You can put the contents into action right away with short examples given by the author. The quick feedback loop paired with the occasional “I never knew vim could do that” moment make you want to learn more with each new page. From Navigation, netrw (vim's file browser), undo/redo branches (why doesn't every editor have that?), remote editing of files on other systems using SSH, to buffers, mappings, folds, windows, autocompletion (so useful and fast), and macros (you've been hiding in there from me all my life?), pretty much everything is covered. A separate chapter gives you extra productivity tips and plenty of ideas on how to apply them.

Since vim is a part of my University course, “Unix for Developers,” I've added extra vim content to my course after reading the book. Explaining these concepts to students who would typically not touch vim, I think I can convince them that they'll be much more productive for learning it. Although mostly written for beginners, I'm pretty sure that even seasoned vim users will find something useful. The book is also useful for people using many of the vim-clones (neovim comes to mind). You may need some time to make these concepts stick. However, now that I know what vim can do, I definitely use it more often. Editing becomes much easier and my productivity has certainly increased. And if you still want more after reading this book, the author maintains a weekly newsletter with additional tips.

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. He serves on the board of directors of the FreeBSD Foundation as vice president. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course “Unix for Developers” for undergraduates. Together with Allan Jude, he is host of the weekly bsdnow.tv podcast.



Tips for Running an Online Conference

BY DAN LANGILLE

I began running conferences in 2002 when I helped with [Open Source Weekend 2003](#). The next year, I started [BSDCan](#). Three years later, [PGCon](#) started. I think I've run at least 32 conferences, two of which have been online: [BSDCan 2020](#) and [PGCon 2020](#).

There have been requests to share what we did, why, and how it went. This is, for the most part, a brain dump of everything I can remember. In general, I will refer to BSDCan, but unless otherwise noted, such references can also be to PGCon.

About the Conferences

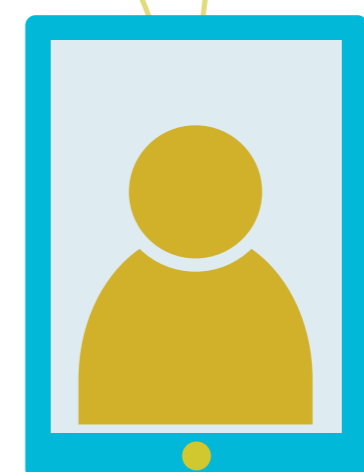
Our choices were appropriate for our conferences but bear in mind that they may not work for yours. BSDCan and PGCon are Open Source conferences and they can be vastly different from other types of conferences. Here's how:

- some people are sent by their employers and others come just because they are interested,
- we are much less expensive than most conferences — \$195 is the usual entry fee and half-day tutorials are \$75 each,
- we have funding to reimburse speakers for travel and accommodation if they need it (this is a large budget item and we do understand it is not the norm),
- we provide catering (breakfast, morning and afternoon breaks, and lunch),
- we have fantastic sponsors.

Zoom

We used Zoom, but nobody needed to have a Zoom client to attend the conference. We wanted to be sure of that. Not everyone wants to use Zoom and being able to view the talks via a standard web browser was important to us and to the attendees. For the Q&A sessions, some speakers dialed into the session using their phones, so even that path is an option.

A Zoom client was required to interact with some sessions, but the Q&A sessions via IRC (Internet Relay Chat) were available. Most open source users/developers are familiar with IRC and use it.



There have been requests to share what we did, why, and how it went.

We used Zoom for:

- live sessions with Q&A for the speaker to respond — this was live streamed to ScaleEngine who broadcast it for us,
- tutorials run by the speakers — they wanted to do this live — 3 live tutorials are easier than 35 live talks,
- closing and opening sessions.

What We Did

We asked speakers to pre-record their talks to reduce the technical issues you need to solve on the day. We recommended speakers record using OBS Project. Our [Speakers: recording your talk](#) might be useful for your conference speakers.

Attendees watched free of charge without registration.

Nearly all of the previously offered sponsorship benefits disappear online. The “sponsored-by” links remained but everything else vanished — tee-shirts were not printed and tote bags were not distributed.

We tried to retain the same schedule as previous years:

- opening session,
- start at the usual time locally,
- three concurrent sessions,
- closing session.

We used local time. Someone was going to be inconvenienced no matter what we did, so we decided to make it easier for us by having the conference during our working hours.

All recordings were uploaded and made freely available later. The only thing non-attendees missed out on is submitting questions. We used [ScaleEngine](#) to broadcast the recordings. They were able to encode all the speaker recordings into various formats and resolutions. They created a schedule and broadcasts started on time.

The Q&A sessions were optional for the speakers in case theirs might not be at a good time for them locally. We had volunteers collect the questions and put them into an [online repository](#) for the speakers to read. Questions were submitted over an IRC channel, one for each of the three concurrent sessions. Sometimes the speakers were online and monitoring the IRC channel which made the question collection repo superfluous, but we did it anyway.

Speakers joined a dedicated Q&A Zoom meeting and we had one for each concurrent session. They knew how this would proceed and what to expect as we emailed them ahead of time and provided practice sessions so they could test their Zoom client. Speakers did not need a Zoom client and they could dial in via phone if they preferred.

The Q&A session was recorded and appended to the session broadcast recording. The ScaleEngine solution allows for layers or priority of channels and we chose, in order of importance:

- broadcast recording,
- Q&A session,
- sponsor video.

If a broadcast started, it would play over top of any of the other two. When the broadcast ends, there is a short video of the sponsor logos. If the Q&A session runs over its time allotment, the next talk starts. However, the Q&A session continues to be recorded by ScaleEngine for later use.

Free

We decided not to charge because of the complexity involved with ticketing and then providing access. We might charge in 2021, but that is open to change.

We could afford not to charge because:

- we had sponsors,
- we did not have to live off the conference proceeds (which might differ for you, if conferences are your livelihood),
- our major expenses (catering, travel, accommodation, venue, tee shirts, tote-bags) were gone — vanished completely.

We had plans for how we could charge, and people would have paid. The charge would have been in the \$5.00-\$25.00 range. We were not worried about people sharing tickets. It could happen. No big deal. We trust people not to abuse the service.

Record or Live?

We chose recording to reduce problem solving. Our speakers live in varied locations and not all have great internet service. There was also the time-zone issue. Speakers recorded at their leisure and we broadcast at our convenience.

What Can Sponsors Get?

Our privacy policy does not permit us to provide sponsors with attendee details. Even if it did, we did not require registration, so there was no way to know who they were anyway. In our case, we created a few IRC channels specific to the sponsors and drove users there.

Things to Be Aware Of

Don't post meetings to Twitter. Bots will attend. Things will go very badly.

Let speakers cancel without penalty. Recording is different. Things go wrong. There's a pandemic. The speakers and their families always take priority over the conference. It's just a conference. If they cancel, thank them, and wish them well.

We posted URLs for the web-based sessions well in advance and only on our website. We had one page for each room with links to the two other rooms.

We posted URLs for Zoom sessions only just before the Zoom session. Communication with attendees was mostly over IRC. You can post updates to Twitter, but don't post Zoom meeting URLs there. As I mentioned, you'll get bots.

On IRC we had channels:

- one admin channel mainly for the volunteers running the conference,
- a main channel named after the conference) (e.g. #bsdcan),
- one IRC channel for each concurrent session — think one channel for each room at the conference venue.

DAN LANGILLE has been using open source since 1998. With a background as a software developer, Dan now works full time as a sysadmin. With his background in writing how-to guides, you are sure to find something useful, if not at least amusing. When not documenting his computer adventures, he occasionally attends a conference as opposed to running it (sometimes the two overlap). Having started his computer antics in Ottawa, he earned his first open source badges in New Zealand, and now resides near Philadelphia, where he works from home.

Network Monitoring on the Console

BY BENEDICT REUSCHLING

This column covers ports and packages for FreeBSD that are useful in some way, peculiar, or otherwise good to know about. Ports extend the base OS functionality and make sure you get something done or, simply, put a smile on your face. Come along for the ride, maybe you'll find something new.

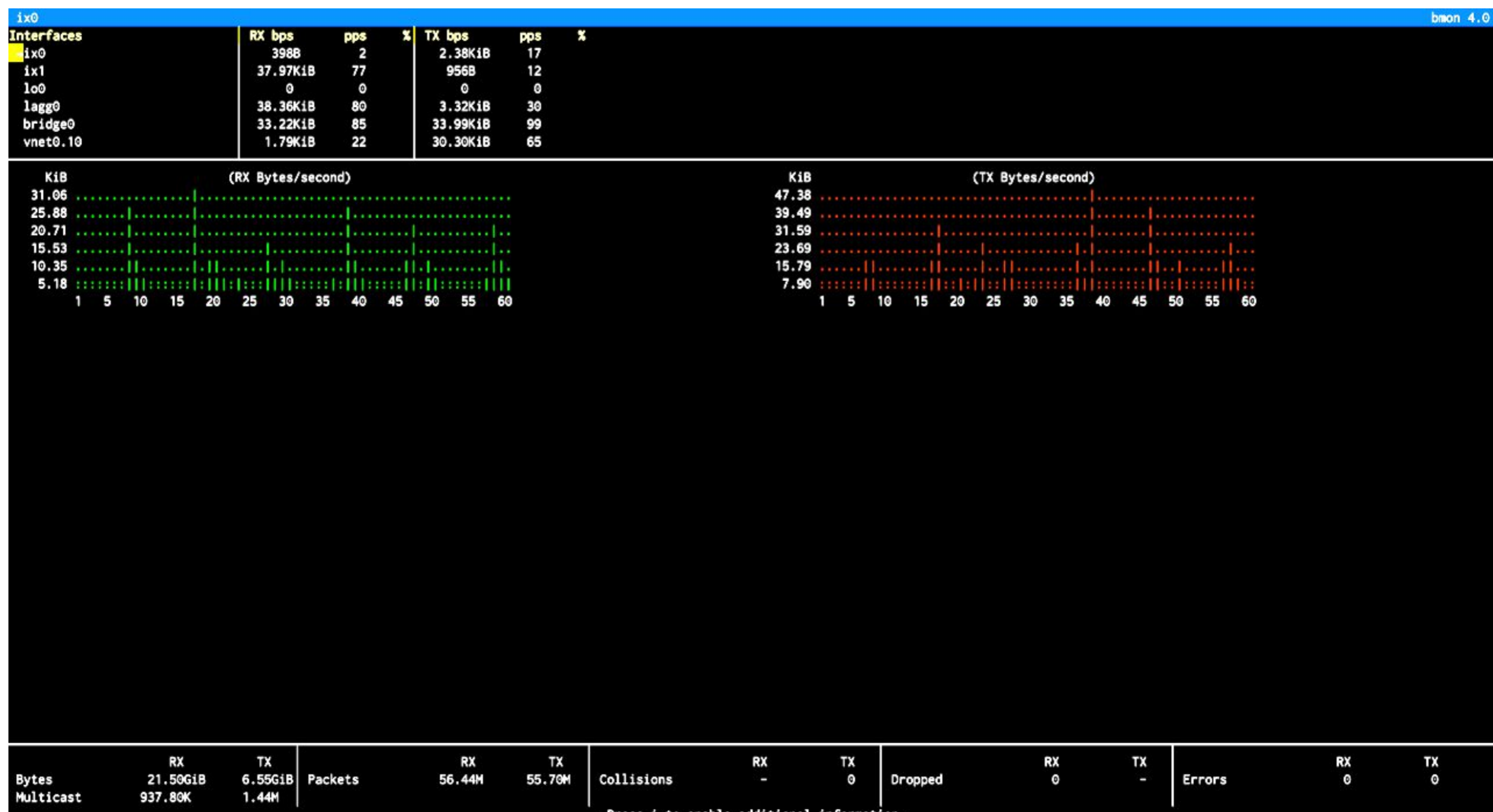
I've been doing a lot of network cabling on our big data cluster. Checking links and making sure that RX and TX lights are blinking are just a couple aspects of it. Once a link has been set up and documented, cables neatly tucked in the rack, IP addresses assigned, etc., it is time to see if the box allows the packets to flow in the right direction. There are a surprising number of tools available to help you with this. I usually breathe a sigh of relief that basic network diagnostic tools are still part of FreeBSD's base system. After all, how are people supposed to diagnose network problems if they have to install a package for that first—and from the network that is not working? Hey, who let the chickens loose in the server room? Look at all the eggs!

Be that as it may, FreeBSD contains `ping(8)` (which does both IPv4 and v6 in one as of late [<https://reviews.freebsd.org/rS368045>]) for basic ICMP checking. If you do get a link and want to see some network activity, `systat(1)` has a handful of utilities available for TCP, UDP and interface statistics monitoring. Of course, you could get really fancy and let `www/grafana` paint the most beautiful graphs on your 4k display with a bit of effort. Nothing against it, just that a lot of us prefer something in between ping and all the bells and whistles of a fully-fledged browser application. Thanks to ncurses and friends, we don't have to give up "graphics" because we chose to stay in the terminal (a.k.a. that black and white text UI).

There are a surprising number of tools available to help you

Let's beef up our ping output with some bar graphs to check long term trends. From a C library called liboping (octo's ping) stems the noping utility (`net/liboping`). Once installed, you ping a target IP and you'll see your familiar packets and sequence numbers rolling up the screen. At the bottom though, there is a trend over time showing any loss in packets. Of course, when I tried to generate a screenshot for you, not a single packet would get lost. So, I refer you to [<https://noping.cc/>] for an example.

How about something for TCP tailored towards the humans among us? You'll certainly find `net/bmon` appealing. I couldn't help myself and included an image of `bmon`'s output from one of my (not so busy) boxes. As a bandwidth monitor and rate estimator—as the description tells us—it definitely produces a nice output for that one computer screen in your office that makes you look busy. I expect to see the ASCII art for the bar graphs in a future computer-focused blockbuster movie.



As an aside, piping your “`zfs list -o space`” output to `misc/nms` gives you a 1992 Sneakers movie feeling. It's exactly the right thing to do when see you a snooping colleague approaching to glance at your screen. Impress that person by unscrambling your screen output with a single press of a button. Repeat it for other commands with a lot of text output.

If you prefer a display like `top(1)`, take a look at `net-mgmt/tcptrack`. It captures packets via `/dev/bpf` (yes, root-permissions only) on any kind of device posing as a NIC connected to your system. Once a connection has been established, source, destination, port, state information, bandwidth usage, and zodiac sign (OK, that last one was slightly exaggerated) of the packets are shown. If you like that, then you should allow `net-mgmt/iftop` to occupy some storage space in your system. The three-column output tells you exactly who your computer talks to all day long.

But who in the name of all the networking gods (the old ones and the new) is using up all that bandwidth? I have several suspects in the form of processes, so I let `net/nethogs` analyze

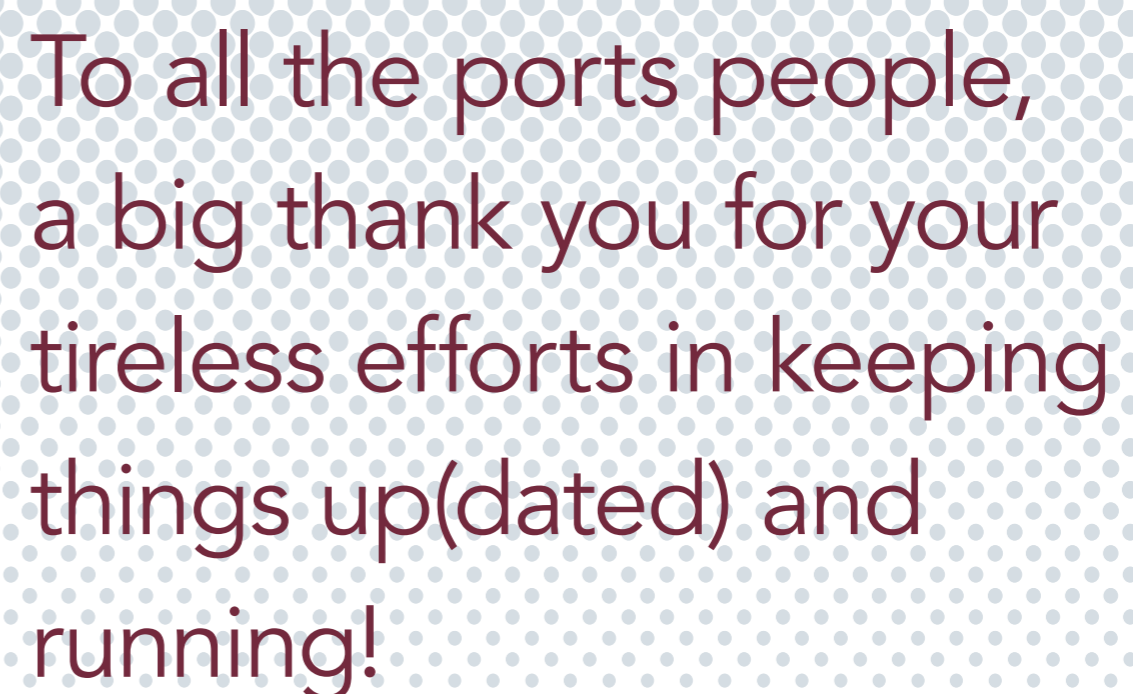
that for me. Instead of many nettop-like tools, it gives me bandwidth by process ID. I had always suspected that running services as root was a bad idea. To a jail(8) with you, bad process!

I sometimes look with envy at software that is not yet ported to FreeBSD. For example, speedometer [<https://excess.org/speedometer/>] would be nice to have. Maybe by the time this column appears, some busy ports committer or contributor will have already ported it. That would really make my day—do I hear a “challenge accepted” somewhere? After all, this column would probably not exist if there weren’t people out there working hard to make sure FreeBSD has a good, third-party software ecosystem. And I believe we take that for granted sometimes. So, to all the ports people, a big thank you for your tireless efforts in keeping things up(dated) and running!

The next time you install a tool you like, why not drop the maintainer a small thank you to brighten their day? You can find them on freshports.org (huge thanks to Dan Langille for that site!) or in the Makefile for the port itself. Unless it is ports@freebsd.org, then the port is up for you to give it some love. And when you do, you can teach others (including me) how to do that, because there can never be enough port maintainers. Check out the WantedPorts page on FreeBSD’s wiki [<https://wiki.freebsd.org/WantedPorts>] for more ports that could be included in the Ports Collection.

If you don’t mind more colors in your network output, then check out `sysutils/glances`. It’s as if `sysutils/htop` and `vmstat` had a love affair and `glances` is the result. There is even disk activity in there, in addition to the `top(1)`-like information in almost every corner of the screen. But of the many `top(1)` clones out there, only FreeBSD’s lists my ZFS ARC statistics at a glance, thanks to Allan Jude’s addition to it. I come back to it often, even with all the other applications the ports collection provides.

As a topic, `top(1)`-like ports could fill a column of its own. And it will, so stay tuned for more in a future column. If you know a great tool that should be included here, send me an email at bcr@freebsd.org.



To all the ports people,
a big thank you for your
tireless efforts in keeping
things up(dated) and
running!

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. He serves on the board of directors of the FreeBSD Foundation as vice president. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He’s also teaching a course “Unix for Developers” for undergraduates. Together with Allan Jude, he is host of the weekly `bsdnow.tv` podcast.

Support FreeBSD®



Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.
freebsd.foundation.org/donate





Events Calendar

BSD Events taking place through March 2021

BY ANNE DICKISON

Please send details of any FreeBSD related events or events that are of interest for FreeBSD users which are not listed here to freebsd-doc@FreeBSD.org.

Users with organizational software that uses the iCalendar format can subscribe to the [FreeBSD events calendar](#) which contains all of the events listed here.



FOSDEM 2021

February 6-7, 2021

[VIRTUAL](#)

Taking place, February 6-7, 2021, [FOSDEM](#) offers open source and free software developers a place to meet, share ideas and collaborate. Renowned for being highly developer-oriented, the event brings together some 8000+ geeks from all over the world. This year they will meet online. Find out more [here](#).



APRICOT

APRICOT 2021

March 1-4, 2021



Asia Pacific Regional
Internet Conference on
Operational Technologies

[VIRTUAL](#)

Representing Asia Pacific's largest international Internet conference, [Asia Pacific Regional Internet Conference on Operational Technologies \(APRICOT\)](#) draws many of the world's best Internet engineers, operators, researchers, service providers, users and policy communities from over 50 countries to teach, present, and do their own human networking. The ten-day summit consists of seminars, workshops, tutorials, conference sessions, birds-of-a-feather (BOFs), and other forums with the goal of spreading and sharing the knowledge required to operate the Internet within the Asia Pacific region. This year, the conference will happen virtually.

APRICOT is a valuable opportunity for participants and sponsors to hear and contribute to discussions concerning current and developing Internet networking technologies and trends.

Find out more [here](#).

FreeBSD Fridays

<https://freebsd.foundation.org/freebsd-fridays/>

Will resume January 29, 2021.

Past FreeBSD Fridays sessions are available at: <https://freebsd.foundation.org/freebsd-fridays/>

FreeBSD Office Hours

<https://wiki.freebsd.org/OfficeHours>

Join members of the FreeBSD community for FreeBSD Office Hours. From general Q&A to topic-based demos and tutorials, Office Hours is a great way to get answers to your FreeBSD-related questions.

Past episodes can be found at the FreeBSD YouTube Channel

<https://www.youtube.com/c/FreeBSDProject>.