Reproducible Builds Summit III

Berlin, Germany. October 31 – November 2, 2017

Event Documentation

Aspiration, 2973 16th Street, Suite 300, San Francisco, CA 94103
Phone: (415) 839-6456 • info@aspirationtech.org • aspirationtech.org

Table of Contents

Agenda....4
Session Notes....10
  Day 1....10
    Agenda brainstorming....10
    Working sessions I....17
      Reviewing existing reproducible builds tools....17
      Discussing the current status of .buildinfo files....22
      What is the ecosystem around rpm?....23
      End user tools: What does exist, what is still needed....25
    Working sessions II....27
      How to fix the current issues with BUILD_PATH_PREFIX_MAP?....27
      How bootstrapping relates to reproducible builds and how to improve it....31
      What documentation is still to be created for developers who are new to reproducible builds?....34
  Day 2....36
    Working sessions III....36
      Improving reproducible builds in Java....36
      Bootstrapping: Mapping the problem space....38
      Best practices and open issues in regards to engaging upstreams....41
      How can policies help the end user to define what they want in terms of reproducibility?....43
    Working sessions IV....49
      Mapping out archive formats....49
      Building a system image from existing binaries....50
      How to preserve software integrity in different legal contexts?....52
      Marketing: Why is it valuable to support the reproducible builds work and who is our audience?....54
  Day 3....58
    Working sessions V....58
      Defining terminology: reproducible, bootstrappable, reliable....58
      SOURCE_DATE_EPOCH specification: Overview and improvements needed....61
      Setting up build environments for reproducibility....63
      What is needed to run rebuilders?....66
    Working sessions VI....71
      Mapping out our short- and long-term goals....71
      How to onboard new contributors....73
      Identifying next actionable steps for marketing outreach....75
      Funding reproducible builds work....81
    Working sessions VII....84
      Enabling cross-distro reproducibility....84
      Exploring reproducibility for Mac and Windows....88

What does code-signing means in terms of reproducibility?..90
Prioritizing the minimum viable set of tools needed for end users................................93
Discussing current status and potential improvements in regards to .buildinfo files for rpm and iso...96
Brainstorming the reproducible builds logo design..98

Agenda

Day 1 – Tuesday, October 31

9:45 Opening session (No notes taken in this session)

The Summit started with participant introductions, a welcome message from the event co-organizers, and an overview of the meeting agenda.

10:10 Sharing knowledge about reproducible builds (No notes taken in this session)

Participants gathered in small groups. The talking points suggested for their discussion were:

- If you attended one or both of the two previous Summits, which were your experience and takeaways?
- Ask Me Anything: Ask any question you have about repoducible builds
- How do you think reproducible builds are already working well today, and where do you see that there is still work to be done?

11:00 Break

11:15 Agenda brainstorming (Session notes start on page 10)

Participants generated a list of topics and questions that they wanted to discuss at the meeting. The result of the brainstorming was then taken into account to identify the working sessions that had emerged as the most compelling for those in attendance.

12:45 Lunch

13:45 Working sessions I (Session notes start on page 17)

- Reviewing existing reproducible builds tools – Chris
- Discussing the current status of .buildinfo files – Mattia
- What is the ecosystem around rpm? – Bernhard
- End user tools: What does exist, what is still needed – dkg

15:05 Break

15: 25 Working sessions II (Session notes start on page 27)

- How to fix the current issues with BUILD_PATH_PREFIX_MAP? – Ximin
- How bootstrapping relates to reproducible builds and how to improve it – Ricardo
- What documentation is still to be created for developers who are new to reproducible builds? – Holger

16.45 Closing plenary

The meeting day ended with an invitation to all participants to share their goals and wishes for the following days, and words of gratitude for a productive start of the Summit.

17:00 Adjourn

Day 2 – November 1, 2017

9:45 Opening session (No notes taken in this session)

The second day of the Summit opened with a summary of the work done during the previous day, and a brief overview of the plan for the day ahead.

9:55 Working Sessions III (Session notes start on page 36)

- Improving reproducible builds in Java – Gábor
- Bootstrapping: Mapping the problem space – Ricardo
- Best practices and open issues in regards to engaging upstreams – Chris
- How can policies help the end user to define what they want in terms of reproducibility? – dkg

11:30 Break

12:00 Skill share sessions (No notes taken in these sessions)

- How to rebuild a specific .deb from its .buildinfo
- How to do everything* in Emacs (*conditions apply)
- Ask Me Anything coreboot
- Binwalk arbitrary firmware images
- How/why not reproducible tar format is
- How to build NetBSD
- How to use reprotest
- How to autoclassify indeterminism
- How to snapshot Debian to build reproducible system images
- AMA F-Droid
- How to integrate with Travis CI

12:30 Lunch

13:30 Working sessions IV (Session notes start on page 49)

- Mapping out archive formats – Eric
- Building a system image from existing binaries – intrigeri
- How to preserve software integrity in different legal contexts? – dkg
- Marketing: Why is it valuable to support the reproducible builds work and who is our audience? – Beatrice

14.45 Break

15:00 Closing plenary

The day was brought to a close with the invitation to share proposals for the next day's working sessions, and the invitation to use the meeting space to hack together during the rest of the afternoon.

15:20 Adjourn

15:20 – 17:30 Hack time

Day 3 – November 2, 2017

9:45 Opening session (No notes taken in this session)

The final day of the Summit opened with the encouragement to focus the last day of the meeting on drafting action plans to be carried out during the following weeks and months.

10:05 Working sessions V (Session notes start on page 58)

- Defining terminology: reproducible, bootstrappable, reliable – Timothy
- SOURCE_DATE_EPOCH specification: Overview and improvements needed – Ximin
- Setting up build environments for reproducibility – Marcus
- What is needed to run rebuilders? – dkg

11:05 Break

11:20 Working sessions VI (Session notes start on page 71)

- Mapping out our short- and long-term goals – Holger
- How to onboard new contributors – Mattia
- Identifying next actionable steps for marketing outreach – Beatrice
- Funding reproducible builds work – Elisa, Chris

12:40 Lunch

13:40 working sessions VII (Session notes start on page 84)

- Enabling cross-distro reproducibility – Janneke
- Exploring reproducibility for Mac and Windows – Nicolas
- What does code-signing means in terms of reproducibility? – Chris, dkg
- Prioritizing the minimum viable set of tools needed for end users – Ricardo, intrigeri
- Discussing current status and potential improvements in regards to .buildinfo files for rpm and iso – Marek
- Brainstorming the reproducible builds logo design – Holger, Brennan

15:30 Closing plenary

The Summit ended with notes of appreciation for everyone who participated in the meeting and contributed to its success.

16:00 Adjourn

16:00 – 17:30 Hack time

Session Notes

Day 1

Agenda brainstorming

--- day 1 am agenda brainst. 1

# Marketing, etc. 1
- how can we better communicate our progress to the public? ** green
- are we making t-shirts ** green
- should we aim to publish a paper on RB in an academic journal to gain more support?
- write and sign RB manifesto ** green
- talks/presentations and other "marketing"/outreach
- how should we communicate the importance of reproducibility
- how can we let even more developers know & care about reproducible builds?

# Marketing, etc. 2
- what made you see the value of repr. builds?
- I do not see much commercial interest in RB, why? *** red
- what % of a computer user's problems are because of package mgmt/environment/reproduce
- (discuss how to) get buy-in from projects that don't yet take part in RB.
feedback results of tests.r-b.org to the developer

# Funding
- how can we become rich with Reproducible Builds? (or: Funding?) * green
- Funding our work in 2019 + 2020 *** green
- Funding reproducible builds
- who is interested in sustainable funding for R-B
- Create formal/legal organizational structure for R-B projects

# Code Signing & meta-data
- embedded signatures best practices **** green
- signing packages vs. signing metadata * green
- how to validate sources in 20 years? *** green
- how to verify sources ** green

--- day 1 am brainst. 2

# (tech) end users
- Develop user-interface to check for reproducibility **** green
- other usertools than "do you really want to install that unreproducible package" * green
- can users configure their system to only install verified reproducible pacakges? if not, what do we need? * green
- What INTERFACES to build reproducibility info do we give to USERS? ** green
- What tools can we provide to _users_ to help them verify their software? **** green
- End-user "prompts" re. unreproducible packages, interfaces, etc
- client tools/strategy for verifying binaries
- how to let users know if a package has been reproduced
- who should be trusted when reproducible package is not actually reproducible? where to report suspicious build
  results? * green

# End Users (not necessarily tech end users)
- Why is reprobuilds important for the end user?
- what RB means for end users
- which USERS should we expose build reproducibility info to? ** green
- what's the gain for the non-tech end user? * green

# Newbie docs
- Let's have a "hello world!" tutorial! * green

- Let's make the documentation noob-friendy! ** green

- Documentation for and questions from newcomers

- Is there a list of common mistakes that makes packages/systems not reproducible? = 70% timestamps

- What are the most common causes of non-reproducible builds ** green

- Define the ambition of Reproducible Builds

- How do we help projects define the SCOPE of their R.B. efforts? (e.g. inputs, build system variations...)

# debian

- What parts of RB are too specific to Debian? is it worth changing them? * green

- Rebuild Debian packages from the archive. * green

- Let's fork Debian?!! *** green, ** red

- Debian: What is our next 1y realistic goal? 95.768%? Reproducible Installer ISO? Anything else?

- reproducibng SOURCE debian package from git snapshot

- Compatibility with signed .debs

---- DAY 1 AM AGENDA BRAINST. 3

# what is reproducibility

- can we do something about communities that cannot build dependencies from source? (npm, java, compiler builders?) : 1 green

- if we make an image by deterministically composing snapshots of unreproducibly built data... do we call that image reproducible? : 1 green

- how can we ensure the term "reproducible builds" is used consistantly across the internet

# Portability

- make sure more computer architectures are supported

- do we want/need reproducibility cross distributions? ("the i386 binary")

- reproducible cross-compiling (i.e. do we get the same as native compiling?) : 4 green

# Tools

- what tools can we all SHARE?

- session/list/introduction about all tools/methods RB works with : 3 green

- how far to bootstrap back the tools to be reproducible? : 1 green

- fork squashfs

- automated analysis of non-reproducible software ("linting")

- how can we let developrs test their software for reproducibility without external tools? : 2 green

- overview/walkthrough of the Reproducible Builds tools landscape. (diffoscope/reprotest/disorderfs/s-nd) : 3 green

# Java

- making Java software reproducible : 5 green

- improve Maven packaging in Debian

- what's the status of reproducible builds in the Java/Android ecosystem?

- easy to use F-Droid rebuilding by user

- Tor Browser latest make it available in F-Droid

---- DAY 1 AM AGENDA BRAINST. 4

# binary artifact formats

- let's make comparison table of filesystem pack formats! (tar vs squashFS vs Ostree vs zip, cpio, etc) : 1 green

- binary artifact explainer workshop : 1 green

- sharing metadata about artifacts between different systems /apt, rpm, jar/ : 1 green

- binary transparency --- can we get it started? : 2 green

- what should be the source of reproducible build? tarball, repository, something else...? : 3 green

# Buildinfo

- create an overview of projects doing .buildinfo files : 1 green

- should all distributions agree on the same buildinfo format, so that we can provide tools to work with them? : 4 green

- require the .buildinfo file standard

- best practice on .buildinfo

- buildinfo storage for the next 6--12 months

- buildinfo format for RPM

- what's the current status? what are solved problems / hard challenges?

# System image

- status of generating reproducible system images? : 4 green

- reproducible debootstrap (basic OS image)

- reproducible containers? : 1 green

- reproducible Debian images eg. ISO images / etc. : 1 green

- how do we build reproducible system images? : 1 green

- are reproducible packages = reproducible OS? What about configuration?

# Distributions

- merging/moving away from being "Debian" specific, e.g. 2 mailing lists, 2 IRC rooms, etc. : 3 green

- How to make SLES join our party? (SLES = commercial SUSE)

- set up / encourage adversarial public re-builders : 1 green

- status of reproducing RPM (on Fedora?) : 2 green

- what is the status and future plans for RPM packages? : 1 green

---- DAY 1 AM AGENDA BRAINST. 5

# Encouraging upstreams 1

- how could we encourage developers to write self-sustained test suites for sub-projects? *** green

- how easy it is for a dev to make a project reproducible? * green

- what to do with uncooperative upstream projects? *** green

- feedback of reprod. to upstream
- only contribute to master branch ** red
- let other systems up on T.R-B.O * green

# Encouraging upstreams 2
- best practices for submitting patches upstream *** green
- explore alternate solutions for lost metrics (as s/w becomes mirrored)
- develop legal guidance for developers / distributors facing pressure to ship malware * green

# Web apps :)
- how do we verify reproducibility of web applications * green

# Distribute
- is there a list of similarities / differences between operating systems reproducibility? * green
- which could be better ways to distribute / check package signatures * green
- how to securely and automatically sign packages/package lists
- what to do with (not) reproducibly built software? ** green
- testing reproducibility, integrating with tests.r-b.org
- how can we make GitHub store immutable tarballs for tagged versions? (pristine tar?) * green
- how can we make a database of signed build inputs and outputs
- where should we collect reproducibility instructions / checksums of results?

---- DAY 1 AM AGENDA BRAINST. 6

# Bootstrap 1
- how acceptable is a not fully from source bootstrappable GCC
- what do we call black boxes in boostrap?
- how far are we from diverse dual compilation?
- find a way to bootstrap the Haskell compiler GHC without using a binary GHC. * green

- find a complete bootstrap path for GCC. *** green
- find a way to reduce the number of binaries we need to trust when building compilers. * green

# Bootstrap 2
- who wants to help: C compiler in Scheme, Scheme interpreter in simple assembly? * green
- realize diverse-double compilation in the real world ** green
- what does bootstrap "from source" mean?
- what is a good bootstrap vs a bad bootstrap?

# BUILD_PATH_PREFIX_MAP
- what can we do to get GCC to accept SOURCE_PATH_PREFIX_MAP? * green
- B_P_P_M and and WTH is wrong with GCC
- how to convince GCC to support BUILD_PATH_PREFIX_MAP? *** green
- final specification for BUILD_MAP_PREFIX * green

# build environment
- are Flatpak, Snapd etc the right way to build
- how can we convince people that application bundles (like Docker images) are inferior to building software reproducibly? *** green
- what are viable build environments for RB? (Docker, Vagrant, chroot...) * green
- How do we handle other, sloppy package managers (like npm, etc)...? Import tools, mirror scripts? ** green
- Is Bazel compatible with ReproBuild specs? (SOURCE_DATE_EPOCH, .buildinfo file...)
- what should be done to make not only builds but system state reproducible?
- why to use standardized build environments or not? * green

Working sessions I

Reviewing existing reproducible builds tools

diffoscope

- diff two artefacts and check if they are the same.
- if not, it will open artefact and will try to find out what's wrong.
- it does this recursetly
- e.g. a zip file, will be open and it will identify which file within
  the archive is different.
- e.g. a elf file will be opened and the section is being identified.

stripnondeterminism

- runs after the build
- remove known causes of non-reproducible
- e.g. jar's aren't reproducible and make it reproducible
- remove several timestamp
- should not exist by designed
- but upstream is hard or take a lot of time. it's an intermediate
  solution

disorderfs

- fuse filesystem
- has different modes
- e.g. random

  - ls -f / `can return every time a difference`
- e.g. invert
  - invert the listing
- e.g. order
  - will order mode

Good practices:

- runs once in "order" mode
- runs second in invert mode to have a deterministic non-deterministic
  way.

trydiffoscope.org

- web service
- you don't have to install diffoscope with the lots of dependencies
- written in django
- cmdline client to avoid main diffoscope install
  - upload
  - diff on the server
  - create a link to be shared into the bug report

reprotest

- tool to run the a build run twice
- can use container
- reprotest `make` will run it twice. but changing the environemnt like
  TZ

tests.reproducible-builds.org

- based jenkins
- doesn't run reprotest
- is running bunch of scripts to do the build twice
- is legacy, but moving to reprotest, isn't going to happen,

as long reprotest is still under development

squashfs-tools

- the
- upstream is not reproducible
- lots of distribution patches
- unclear if the reproducible patches gonna accepted
- lynxis will do a fork because the maintainer isn't
- not reproducible becaues of timestamps and scheduling problems

guix

- packages manager guarantees reproducibility
- guix challenge compares difference sources of a binary

reproducibleopensuse

----

can test a packages which has been uploaded to opensuse build system.

varies:

- hostname
- date

tool request

- to make javascript reproducible, there should be a `npm-ls`
  to get all the npm packages.

- someone created a cross-ecosystem scraper to scrape npm, pypi
- prevent build systems to talk to the internet
- run a tcpdump on the system. if the .pcap file is greater than 0, you
  know the build talked to the internet. It helps create a good bug
  report to know, to which server it talks and what.

Put the "is the build talking to the internet" feature into the build system.
- call `runc` without network
- create a namespace without network

Feedback results to the community

reproshilds.io??

- travis-ci integration
- create a tool to integrate it into their build process
- create a feedback api.
- get badges

The unreproducible packages
----

have a package full of non-reproducible things

next sessions

- squashfs forking session
- reprotest

Discussing the current status of .buildinfo files

build info craziness

- Limitations Debian implem of build info
  + no checksum of dependencies in build info, deb limitation: #802241 (debian bug)
  + no checksum of the source code.
  + no architecture of the installed dependencies (e.g arm installed on x86_64, cross-compilation, multi arch binaries, ...)
- no actual specification of build info, only a man page of the debian implementation
  + other implementation: archlinux, OBS (open build service -- openSUSE). Need to follow-up on openSUSE buildenv.
  + man pages missing some information
  + Lot of things called build info but not necessary build info.
- what is debian specific in build info?
  + None of the field is debian specific
  + But format might be
- do we want to have a single format for all distribution?
  + Not essential...
  + But ease the adoption of repro build: easier to compare, easier to share tooling, ...
  + Maybe they want to use something that feel native to them

- Build info specification on the debian wiki
  + Do we need to have a repro-build level specification? (in https://reproducible-build.org/specs)

We narrowly avoided an argument about csh

- Build info file for ISO distribution, what format would that be?
  + A separate topic that other people might want to discuss? Not a topic for us now.

What is the ecosystem around rpm?

two types of rpm: source and binary

src.rpm: tarball, build instruction, etc

rpm packages are signed; one can take signature from one rpm to another, as long as other part is the same

also, for comparing one can strip signature

package itself is reproducible already, patches already included upstream:

- %_hostname macro
- $SOURCE_DATE_EPOCH as build date
- use topmost changelog entry as $SOURCE_DATE_EPOCH (%source_date_epoch_from_changelog macro)
- clamp file timestamps to $SOURCE_DATE_EPOCH (%source_date_epoch_from_changelog macro)
- size of directories included in cpio inside of rpm -> zero
- ghost files (files owned by package, but not included in actual archive),
  rpm include hash of such file (from build environment) -> fixed; file size
  is still included (it is a feature)

rpm include compiled python files, which contains timestamp of source files;

clamping mtime (of source files) may in some cases break this - if mtime does

not match here, python recompile the file each run

PEP 552 - use source hash instead of mtime

OpenSUSE: open build service, produce "buildenv file", including hashes of build inputs

osc - client tool to build package locally; this tool is also packaged for Debian

build path doesn't matter, it is constant (chroot env) interesting topic: use reprotest for rpm packages

reproducibility testing in OpenSUSE: not automatically, but there are manual results

further tasks:
 - document current status, environment

End user tools: What does exist, what is still needed

which users are we talking about?

- people who are installing software
- admins who maintain systems for users

user stories are good

- scientific users: if you want to write reproducible papers, you must use reproducible software.
- doctors (and lawyers) are faced with the mandate to encrypt user data. with r-b they can have more confidence the software is untampered with.
- developers who want a trustworthy base to develop software on.
- verify software build or installed on systems one is not allowed to touch. useful for admins to support (external or internal) users.
- users who want confirmation they run free software!
- users who want confirmation they run the software they intend to run

the apt-hook (exist)

gnome software center (doesnt exist)

what might the user want to know, what problem they want solve?

is it helpful to show that a software is unreproducible to the user? is that meaningful information?

the user wants to set a policy on their device: only reproducible software. they dont care much about individiual unreproducible software…

there will be different policies: (incomplete list)

  - only install reproducible software which all rebuilders agreed on
  - fine if one rebuilder disagrees
  - fine if two rebuilder disagrees
  - rebuild locally in case of conflict

- always rebuild locally

- i only want reproducible software with the exception of my wifi driver and i need flash in the webbrowser - and i love that 3d tetris game too

it should be possible to upgrade to a stricter policy.

could be opt-in, people who are not interested in reproducible builds (who have not opted in) should never be informed that/if software is unreproducible.

because cavevat:

- warning fatigue. too many warnings can scare users away.

users dont care about reproducbility, its a technical detail. they care that they run the "right" software.

denial of service by malicious rebuilder

unrelated

----

we need rebuilder howtos

outcome

----

another session to enumerate possible policies

another session to write user stories

Working sessions II

How to fix the current issues with BUILD_PATH_PREFIX_MAP?

Problem: build paths get injected during compilation.

one commmon scenario is embedding debugging information in C.

next most common are __FILE__ macros and assert()

gcc's -fdebug-prefix-map is almost a solution, but…

command line options used were also embedded in the debugging

information, so the use of -fdebug-prefix-map with a path-specific string itself introduced a variation.

we convinced gcc to not store -fdebug-prefix-map.

But this still doesn't solve __FILE__ and it doesn't solve other

systems outside of gcc that also embed the build path.

So we have the BUILD_PATH_PREFIX_MAP proposal: an environment variable with a list of pathname prefixes that should be substituted for some other string, in all places, including debug symbols, __FILE__, and assert(). And if anything else ever intends to embed the full path the the source files, it should respect it as well.

For example:

```
    BUILD_PATH_PREFIX_MAP=$(pwd)=. make
```

Would strip the top-level directory from all stored paths, and replace it with "."

maps are proposed as colon-delimited, so you can provide more than one

of them.

We had some brief discussion around the similarity of the BUILD_PATH_PREFIX_MAP proposal with SOURCE_DATE_EPOCH:

  * they're both environment variables

  * they both have clearly established semantics of how we want build systems to transform/stabilize certain forms of ephemeral data

  * in many situations, the goal is for the build system to not need either env var, because the ephemeral data in question simply isn't a part of the build process at all.

We currently have patches oustanding for gcc and golang. We don't think that anyone has even asked clang for support yet.

R (and some other compilers?) doesn't need BUILD_PATH_PREFIX_MAP because it already has a

notion of the root of the source tree.

gcc is reluctant to accept the patches for the environment variable -- they apparently don't want magic options to change the build! so their concerns align with ours in some way, but the thinking about how to address the concerns are different.

Instead, gcc upstream wants to add a command-line flag, something like -path-prefix-map as a generalization of

-fdebug-prefix-map (i.e. it would include __FILE__ and assert())

dpkg would need to add this option (to $(CFLAGS)?) for each package build, but this actually doesn't solve our problems for packages that themselves embed the gcc build options (same problem again as -fdebug-prefix-map :(

Do we have stats for how many packages would be affected by this problem? Ximin did some analysis and suggests it's about 1800 packages whose reproducibility would be fixed.

another approach mooted was updating a "spec file" for gcc -- debian gcc
maintainer is reportedly not keen on spec files, so this wasn't discussed further.

some discussion on why we don't just fix the build path during build: we want people to be able to do rebuilds without any special privileges on their build system. this makes reproducibility testing easier and cheaper, and it also demonstrates that things we know to be unrelated to the build are not actually included in the build.

Current plan is for Ximin to submit a series of three patches to gcc:

1) patch for command line flag that has map (exactly what they asked for)

2) second command-line flag that says "read this env var" and use it for the value of the flag

3) default flag to true.

then if they accept only (1) then we can ship a wrapper around gcc
that reads the env var, and we can configure dpkg to invoke the
wrapper. this setting would need to be done specifically for the
packages that need it (<1800 packages). We'd do that by modifying
$PATH for those specific packages.

netbsd has been building their base system 100% reproducibly for two years. we
need to research what they're doing further. It's possible that their base system doesn't have any __FILE__ or assert(). Or its possible that they're fixing the build path.

Qubes currently uses a fixed build path, but would be interested in being able to avoid that.

Vagrant proposes putting literal string '$VARNAME' as argument for
-path-prefix-map, so gcc could then call getenv() internally. This would result in a static argument passed to the compiler (so it is reproducible even on systems that embed the arguments to gcc)

Followup discussions/hacking proposed:

 * investigate what netbsd is doing.

How bootstrapping relates to reproducible builds and how to improve it

bootstrappable.org

bootstrappable builds

open questions:

What is the relationship to reproducibility?

Why should we care?

We need to trust binaries used during build. If build binaries are not trustworthy, this makes build results less trustworthy.

#If we have a exe at the top and a lib it depends on, do we call the exe reproducible even if the lib is not reproducible?

is trust binary / black&white? no...

overlap:

build as much from source to need less trust in binaries, and to be able to read/review the source

increase trust in the software we build

trust gets added with every way you can build a software from source

diff: reproducible builds: done when 100% of packages are building reproducibly

bootstrappable: (C-part) done when one can take any C-compiler and compile the production C-compiler and with that get bit-identical binaries of anything

Requirement for doing bootstrappable builds:

A "seed set" of bootstrap binaries has to be declared by a project (e.g. see f-droid below or a binary with a specific checksum), must not be implicit (e.g. previous version of myself)

For build use a limited build environment containing only those binaries and nothing more.

large complex source can still be hard to read, understand, verify => reduce amount of trusted software

backdoors in built source code are out of scope of bootstrappable builds

Is it a software freedom problem? maybe, maybe not

long chains of A->B->C may bitrot and break over time, give less trust than A->C ; but we have archives, and containers

obsolete hardware needs to be emulated and the emulator becomes part of the binaries we need to trust.

Have build scripts that are fully specified about versions of compilers, etc to use in a boostrap-chain.

Note: Trust is not transitive (unlike a=b=c meaning a=c) so if the sister of a friend knows someone who verified this it is not as much trust as "I verified this". Possibly also beacause trusting someone very much translates to a factor of 0.9x thus for every level of indirection you lose some trust.

f-droid: using debian binaries as much as possible because they are built from source and thus more trustworthy.

guix: build archive with checksums of everything with 218MB bootstrap binaries

openSUSE: FIXME link to Ring-0

Goal: come up with very small set of auditable binaries+sources

https://gitlab.com/janneke/mes is close

https://savannah.nongnu.org/projects/stage0

Goal: need zero trust in the seed set of binaries - cannot be fully reached, but we can get to very small (maybe infinitessimal) values of trust needed.

How to distinguish trusted bootstrap binaries from other binaries?

tools/compilers that depend on themselves:

gradle

ghc(Haskell)

rust

maven

identify important next steps:

collect list of bootstrappable and non-bootstrappable tools

convert notes to proper doc

identify high-importance targets to achive bootstrappability

more to be discussed

What documentation is still to be created for developers who are new to reproducible builds?

newbee docs
===========

- entrypoint of https://r-b.org good.
- webpage alone is not enough
- debian wiki is needed
- found a theoretical explanation
- missing: hands-on-guide, good examples
- example -> `gzip` vs. `gzip -n`
- reprotest is helpful

- guide:
  - 1st: build a binary reproducible
  - 2nd: build the debian package reproducible

- good examples:
  - the unreproducible go packages
  - the unreproducible hello-world.c
  - .py .ruby [...]
  - should cover each type of reproducibility and each language
- we need documentation for every language including all specialities
  also build systems should be explained e.g. autotools, cmake, [..]

- what's reproducible?
  - append the "definition of r-b" with a paragraph "confusions about the term reproducible" and then explain that

- new document (wanted): "how to contribute"

- wiki should go away? yes!
- only one source of documentation

- link the definition on the r-b.org page
- create a docker container with all tools

- video tutorial how to make a package reproducible? the easy packages are getting rare, so it's hard to learn. (maybe an ascii video)

Day 2

Working sessions III

Improving reproducible builds in Java

Java Bootstrap path --> working for guix

Bazel has less problems because they always use bazel to build

Bootstrapping maven and especially gradle is still challenging

bootstrapping maven from older versions of maven, requires a few steps.

For gradle it's harder because it depends on really recent (nightly) versions of gradle.

--> idea to bootstrap with groovyc

BUT: Gradle since version 3.0 depends on groovy >= 2.0 which depends on gradle.

Grovy 2.0 depends on gradle 1.0, which depends on groovy 1.8.4 which has a maven build-system.

--> same idea with maven but easier

Problem with java projects which have tests which depend on the parent project. --> dependency cycles

----

Javadoc --> produces unreproducable output based on filesystem order

-> Fix at source when read from filesystem

There might be a debian patch: javadoc-sort-enum-and-annotation-types.diff

Gradle - groovy bootsrap chain:

maven 3->groovy

groovy ->	gradle
1.8.7	1.2
2.0.5	1.12
2.3.6	2.3
2.4.12	4.3

Gradle depends on over 300 previous versions of itself.

Try to go the if we can compile ourself corretly after skipping trough several steps. Git tags seems to be good indicators.

Bootstrapping: Mapping the problem space

- two possibilities: minimize the binary seed; eliminate the importance of the binary seed (diverse double compilation)

- usually diverse double compilation is only feasible for C compilers (because other languages and build systems often only have a single implementation)

Timothy showed his build-dependency graph, an informal generalisation of diverse compilation to projects with more than one bootstrapped dependency. Different/diverse "seed sets with no shared binaries".

- debian currently doesn't declare an exact set of bootstrap dependencies; f-droid assumes many binaries from debian; projects should be specific about these assumptions in general. Ricardo showed us the Guix bootstrapping diagram, a formal explicit bootstrapping specification. Seed set includes things like "mkdir", could be made further abstract e.g. pointing to POSIX spec

- reducing binaries is a goal in itself: the number of binaries we need to carry around to get started should be small; it is not necessarily about increasing trust (there are other ways to get there: diverse double compilation, long chains of rebuilding, etc)

- Definition of "diverse", i.e. not coming from the same authors. Golang 1.4 is not "diverse" from golang 1.6, for DDC purposes

- how can we encourage compiler writers to keep around a bootstrappable implementation? there are disincentives for developers: maintaining a bootstrap path is expensive. by setting a good example of a pretty bootstrap we could demonstrate the benefits

- sharing bootstrap paths, e.g. from a C compiler, lowers the bootstrapping cost across the whole ecosystem, reduces attack surface. harder for compiler writers however, who want to write in their own language.

- one strategy is to find a binary-reduced bootstrap path to a C compiler that can build GCC; any language that can be built with GCC would thus benefit (including Guile, Hugs, etc); another path is to build a C compiler in another language that itself has a short binary-reduced bootstrap path; another path is to revive ancient compilers (but that's hard)

- brief digression into the fact that GCC does not distribute its own binaries
  - maybe makes it harder for compiler writers to bootstrap themselves
  - OTOH, independent distros supply "independent binaries", maybe diverse enough for DDC (unsure if totally solidargument)

- identify "checkpoints" (i.e. verified binaries), possible ways to verify a binary:
  - DDC with multiple independent bootstrap compilers
  - human verification that binary matches source code (for small enough codebases)
- allows us to confidently use the binary to bootstrap future stuff
- can also be used to verify future checkpoints more directly, using 1 single normal build
- (assumes all these processes are reproducible)
- keep a record of which checkpoints generated which other checkpoints

Rest of the session was about constructing the build-dependency graph between compilers

Partial dependecy graph of a ghc bootstrap path, early boostsrapping information at: https://elephly.net/posts/2017-01-09-bootstrapping-haskell-part-1.html

8.2.1

8.0.1

7.8.1

7.4.1

6.12

6.6

5.0.4

One idea to consolidate the approach in the link was proposed, is to use nhc98 to build a 32 version of ghc, then crosscompile ghc to x86_64.

----- Transcription of the poster

A node in this graph represents a particular compiler. Edges are "can build" relationships.

Short chains are preferred.

Checkpoints are binaries with a specific hash---the graph above and including the checkpoint could be replaced with that very binary. Ambitious projects may want to increase the number of checkpoints that are reached trhough compilation from source.

[Graph:

In th emiddle there is a node gcc-4.7 and annotations "Diverse double compilation";

can be built from and can build may other compilers, in particular clang and tinyCC ("25,000 loc").

from gcc-4.7 there is an edge to ggc-5. From gcc-5 there are many outgoing edges: cpython, ruby

interpreter, jikes, hugs, llvm. golang can self-build and also be built from gcc-5.

3 nodes are marked in red (indicating "not cool"): ocamls requires gcc-5 *and* ocaml,

ghc requires ghc itself to build, rutst requrest clang, llvm, python interpreter *and*

rust itself.

For ghc there is incoming edge from hugs, marked with a quesiton mark.

For tinyCC there is an incombing dotted line chain indicating a possible bootstrap path:

slow simple lisp -> messcc -> tinnyCC. Another dotted-line bootstrap path drawn is

slow simple lisp -> mes -> guile -> nyac + mescc.]

This graph may require

- "or" inbound links (somewhat rare... but desirable! Tends to suggest a good checkpoint)
- "and" inbound links (typical: srcs and compiler, etc)
- probably use a different color for textual sources
- optonal useful node annotaitons: binary size (smaller tends to suggest a good checkpoint)
- should we highlight loops? (again: probably good checkpoints)
- mabye a giant elipsis for where $language has a whole huge tree (we mostly care about compilers here)

Best practices and open issues in regards to engaging upstreams

. follow upstream processes

  - difficulty finding .any. place to contribute

  - difficulty finding the .right. place

  - github, gitlab, sourceforge, devel mailing list, bugtracker cover 90+% of cases

  solution: ask on IRC or via direct mail on how to contribute

. difficulties in relying on package maintainer to forward etc

  - ask distribution maintainer to forward your patch upstream

. patches rejected

  . they want to keep their timestamps

  . suggestion: "ask why the current status quo exists", not just assume they are °wrong°

  . discuss how to solve the problem differently in a way the maintainers like

  . rejected (or not merged) due to lack of ability to test code paths or regressions

    . add tests

  . find a different reviewer with a saner view of the world

  . try to be friendly and not push too much - might trigger defensive behaviour of maintainers

. gentle pings

  . "what can I do to help you merge this" in the case of unmerged or untouched patches

  . wait 1 month at least, exponential backoff to not be annoying e.g. wait 1, 2, 4, 8

  . "I will do it on the weekend" or "next month" - follow up after that

  . keep track of patches to ping - e.g. keep URLs in an etherpad, script to send emails

   - If there is a patch in debian that is not forwarded to upstream, when forwarding upstream mention that Debian already has that patch and you want it in your project as well to signal that there is more than 1 interested party. for bugs.debian.org update forwarded status

   - link back to debian/other distribution's bugreport in upstream one

problem: deprecated branches

- when patches do not apply to master because it has been rewritten

- try to get an review/approval from upstream - makes it easier to get it merged into distributions

- adopt in https://github.com/distropatches/

problem: patch sent during deep freeze, so patch is ignored/delayed

solution: need to ping at the right time

idea: keep and share template-snippets of well formulated commit messages or bug report, include reference to documentation so it can be more verbose and useful to upstreams than when writing from scratch every time.

e.g. https://wiki.debian.org/ReproducibleBuilds/Contribute#How_to_report_bugs

problem: have to sign CLA

- argue that it is not copyrightable
- declare patch to be MIT licensed so it can be integrated anywhere
- bots can be very strict, so signing the CLA is often the easiest solution

e.g. Qt, google, GNU, facebook, python

- send bad patch, so maintainer has to rewrite it and it is his copyright

How can policies help the end user to define what they want in terms of reproducibility?

How can the end user define what they want in terms of reproducibility?

# Policies

* there is no one size fits all policy
* can you verify using a checkbox that the software is the one we want to install?
* but we want the computer to verify that many people have reproduced the same software.
* how to handle exceptions to the policy?
* providing these policies to users is a different question than to provide this kind of information to developers/researchers
* after the installation, we should also be able to verify the software we installed on the computer

* question: what if the news arrives via Twitter, should this be part of the policy?

Questions:

- What is the range of reasonable policies?
- How can we communicate these policies to machine adminstrators?
- How can we implement this in the backend.
- What are the downsides of these policies?

# Example policies

EXAMPLE 1

```
REBUILDERS              BUILD INFO FILES
==========             ================
---------------
| rebuilder A | -------→ pkgX matches digest N
---------------
```

```
---------------
| rebuilder B | -------→ pkgX matches digest N
---------------
---------------
| rebuilder C | -------→ pkgX matches digest N
---------------
                     |
                     |
                     v

                     MACHINE
                     =======
                     policy configured to
                      - fetch build info files from three rebuilders
                        for example Debian, ACLU, CCC, some army :)
```

local admin could pick a set of rebuilders and pick build info files from the 3 and check if they match.

If the three build info files don't match we could

- not install the package
- try to rebuild it ourselves

→ This is what we would need to define in the policy.

- how many rebuilders do we want to match (m out of n)?
- where does the list of rebuilders come from in first place?
- think big, maybe we need 500 rebuilders, not only 10. like universities for example.
  → we could have a rebuilder consensus
  → it would be harder to attack such a network of rebuilders
- How is the redistribution gonna work?
- Backup mechanism for policies

- there is a time when packages would appear as being unreproducible, for ex. when security updates are published, tthe rebuild will take time

- have an admin specified rebuilder, for example a local rebuilder

- have a weighted builders (i trust my friends more than XYZ)

  - we want to be able to say that some rebuilders cannot block consensus for us

- how do we ensure that our failure modes are not failing?

- what if corroboration fails, is missing, is insufficient?

→ install anyway

→ don't install

→ local log

→ external log: report

  - on a distributed architecture, how and where do we report it?

- rebuild it ourselves

  - and then? it could match one of the rebuilders, none, the majority, the minority

  - which buildinfo file will we use?

→ what if the OS rebuild does not correspond to the consensus' checksums/buildinfo files?

- policy for rebuilds: always rebuild and then check the consensus, or never build because i don't have resources, or build only in specific cases, like if checksums don't match

Rebuilders

- architectures?

  → for some archs, m out of n will be different

- for n out of m:

  → default = "the majority", ex: for 10 known rebuilders the strict majority would be 6. majority of potential rebuilders

  → time -> "not having rebuilt" does not count towards your majority, unless you have a local on demand rebuilder

- we don't challenge the idea that we download the buildinfo files from somewhere.

  - for ex, we could be able to specify the download from a local machine, in case of a company.

- right now, for user policy, the question is not that much where we get the buildinfo files from, but what we do with it.

- could be a sudo rebuilder daemon which publishes buildinfo files (proxy policy)
  - let's put aside the question where we get the info from right now
- currently admins have to have /etc/apt/sources.list or a similar file
  → we could have a rebuilders.list
    - question: m out of n → is this per package, per repository, per combination?
      - policy per repository
        - rebuilders then have to commit to rebuild the contents of the entire repository
      - weight
      - what do to if corroboration is insufficient
      - log/loglevel
      - what if my rebuild fails
      - exception handling
  → default policy
      - maybe we could turn this on with sensible exception handling at least
        → for example report externally, warn/log
      - exception handling for packages which are currently known the be unreproducible
        → example GCC
      - turn it on for packages which are supposed to be reproducible (since n times)
  → have one default policy for Debian/etc repositories, one per PPA
  → repositories themselves should state what their default policy is
    - "sensible default"
    - don't blindly trust changes of this policy / compromise
    - policy shipped in a package
    - there should be a list of rebuilders
    - but admins should also be able to manually select the rebuilders
  → user specified policy

- this is still very complicated, we should break it down to 3 options.
- GUI where we could check "Debian as rebuilt by the CCC"

EASY GUI:

Policy Chooser per repository

----

x default (none)

----

o Debian

----

o Debian stricter

Advanced button

ADVCANCED GUI:
REPOSITORY X

Trusted rebuilders	INCLUDED	REQUIRED
Rebuilder A	x	
Rebuilder B		
Rebuilder C	x	

Number of rebuilders needed for consensus (m out of n): 2

→ by default weight is binary (0/1), and...
..in advanced mode we could use non-binary weights.

Consensus fail policy
Log policy

It seems doable! But what's the cost/benefit ratio of it?

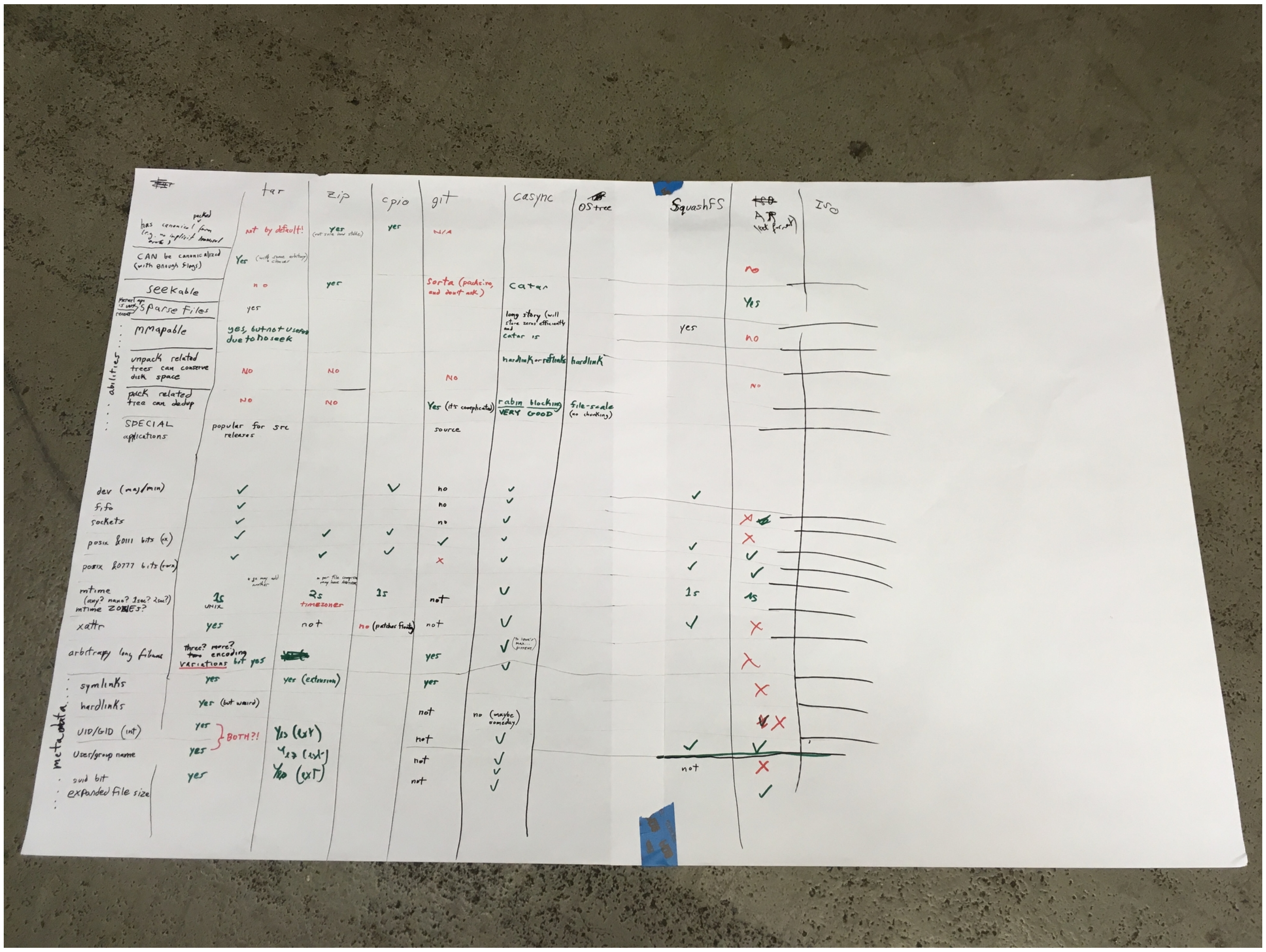
- User stories
  - "Running Debian as rebuilt by the CCC"
- Snapshot maintenance
- Rebuild service -> just send a queue to a local or whatever rebuilder
  not as part of the installation process

→ Let's extract the smallest useful subset of this in another session!
  <3 Minimum shippable piece

(outside of policy questions: how do we ensure we don#'t add more privacy concerns for users with this)

Working sessions IV

Mapping out archive formats



	tar	zip	cpio	git	casync	OStree	SquashFS	AR (ext format?)	ISO
abilities									
has canonical packed form (e.g. no implicit ~~[illegible]~~ order)	not by default!	yes (not sure how stable)	yes	N/A					
CAN be canonicalized (with enough flags)	Yes (with some arbitrary choices)							no	
seekable	no	yes		Sorta (packs: no, and don't ask)	catar			Yes	
Sparse Files (kernel api is very recent)	yes				long story (will store zeros efficiently and catar is		yes		
mmapable	yes, but not useful due to no seek							no	
unpack related trees can conserve disk space	NO	NO		NO	hardlink or reflinks	hardlink		NO	
pack related tree can dedup	NO	NO		Yes (it's complicated)	rabin blocking VERY GOOD	file-scale (no chunking)			
SPECIAL applications	popular for src releases			source					
metadata									
dev (maj/min)	✓		✓	no	✓		✓		
fifo	✓			no	✓			✗ ~~✓~~	
sockets	✓			no	✓			✗	
posix &0111 bits (+x)	✓	✓	✓	✓	✓		✓	✓	
posix &0777 bits (rwx)	✓	✓	✓	✗	✓		✓	✓	
mtime (any? nano? 1sec? 2sec?) mtime ZONES?	* gz may add another 1s UNIX	* per file compression may have another 2s timezones	1s	not	✓		1s	1s	
xattr	yes	not	no (patches floating)	not	✓		✓	✗	
arbitrary long filenames	three? more? ~~two~~ encoding variations but yes	~~[illegible]~~		yes	✓ (to linux's max... different)			✗	
symlinks	yes	yes (extension)		yes	✓			✗	
hardlinks	yes (but weird)			not	no (maybe someday)			~~✓~~ ✗	
UID/GID (int)	yes } BOTH?!	Yes (ext)		not	✓		✓	✓	
User/group name	yes } BOTH?!	[illegible] (ext)		not	✓		not	✗	
suid bit	yes	[illegible] (ext)		not	✓			✓	
expanded file size					✓				

Building a system image from existing binaries

Building a system image from existing binaries
- binaries = build dependencies, like in static linking
- source code = scripts used to create the image, independent of specific
  versions of the external packages that are used

Downloading binaries from remote places
- Bootstrapping processes need a similar thing, download upstream bootstrapping
  binaries
- Declaring binary dependencies is the main thing, c.f. buildinfo files
- What one does with it during the build is not material, as long as it's
  reproducible

Build environment
- distribute description of a VM, e.g. gitian
  similar concept to buildinfo, which describes build-dependencies
- doesn't have to be the exact same VM or environment (and shouldn't be) but
  similar enough to express the build dependencies, and reproduce the build.
- if it's easier for users to recreate the build-environment -> more
  reproducers, better for everyone
- ideal for each project to have a one-liner shell script to run that outputs
  "is this reproducible"

FS images
- mk_ext4, deterministic tool to create a reproducible ext4 filesystem
  mkfs for BSD should be reproducible
  - might be dependent on ordering of input operations, etc, but still
    reproducible if that is reproduced
- isofs, reproducible upstream
- squashfs, have patches, but upstream uncooperative

- libguestfs, run a image in a VM, but not reproducible

Setting SOURCE_DATE_EPOCH / timestamps
- SOURCE_DATE_EPOCH can be set to the latest modtime of all packages (i.e. build-dependencies), communicates some info about how recent the image is

Cache files / other post-installation products:
- drop if not necessary
- fix if necessary to be static
- generate at boot time, if necessary to be random or system-specific (like UUIDs or RNG seeds)

Other sources of unreproducibility
- DNF has dependency resolution depending on configured priority of remote repos => need to store this config with the source code
- debootstrap should be more or less reproducible, testing required

Use hash information in buildinfo to reproduce d-installer output (images)

How to preserve software integrity in different legal contexts?

Where we're coming from

- dkg works for ACLU (but is not a lawyer)
- concerns: government forces devs to ship malware, with gag orders (not allowed to talk about it)
- solution: make systems where it's impossible for devs to comply (every change has to be visible and accounted for)
- we have at least four legal jurisdictions represented in our small group (US, Brazil, European countries)
- one way to market RB is to mention the mafia, but in some places law enforcement can be perceived as the mafia and can be used instead

What can law enforcement force you to do?

- example: Apple vs FBI (https://en.wikipedia.org/wiki/FBI%E2%80%93Apple_encryption_dispute) but doesn't really work because we do not have the source code
- the same example could apply to the maintainer of tails though (baseball bat kind of threat to flip a bit)
- another example mentioned; gag order against archive.org rescinded (https://archive.org/details/GagOrderRescinded)
- depending on context we might agree willingly (catching child abducters...)
- typical gag order does not allow you to talk to your lawyer (probably not true)

Side question:

- what is the prior art of gag orders? (eg before computers and Reproducible Builds)

How can we defend ourselves?

- employment (google...)
- 501c foundations for our OSS projects, either directly (NetBSD Foundation) or indirectly (FreeBSD Foundation, SPI for debian...)

- foundations like ACLU (1.5M members in the USA)
- not being the single point of failure
- having a lawyer, or going to a journalist? (if not fully detained)
- revoking our own accesses (private keys, destroying second factors...)

Dealing with jurisdictions is very difficult, and many of us live in countries with a passport issued in another, making it even more complex.

Take from the session

Are Reproduble Builds really a deterrent?

Can we technically make it difficult or expensive for threats against us thanks to Reproducible Builds?

Can we ask foundations what would happen if a developer is under (legal) pressure to insert a backdoor, and reaches out to them?

Can we come up with a guide gathering ways we have to defend ourselves?

Report

Very complex topic, which raised more questions than answers - for now at least. We focused on which situations we can individually be pushed to do things detrimental to the integrity of software projects, and ways to defend ourselves. Besides being employed by big companies, foundations may manage biggest projects, but not always. We need to ask them which steps they would and could take for us. With this we could or should come up with a more general guide on how to defend ourselves?

Marketing: Why is it valuable to support the reproducible builds work and who is our audience?

1st brainstorming session: what are the reasons why you think marketing is a good idea (or not)? (see first post-it board)

we identified four clusters that marketing relates to: users, developers, management, and the free software community. There is agreement that marketing is important to become visible, thereby moving the burden of discovering RB from the audience to us. It is also important to increase diversity in the RB community.

2nd brainstorming session: who would be the targets for marketing efforts? (see second post-it board)

observation: there's a lot of overlap e.g. NPM (the company), NPM (the package manager developers), NPM (developers publishing JavaScript packages via NPM), etc.

Interestingly, only few votes went to individual users, because their individual power is limited. Targeting organisations might be better, because *they* can reach many more people that are all similar. It makes more sense to target "user groups" (e.g. via Amnesty International, or HPC users) than individual users (through individual user stories).

what is the message we have for them?

"Compiler builders" are a target, because compilers are crucial to all software. The message to them is: you don't want your compilers to produce random results. Appeal to their pride in creating something beautiful and accurate that produces deterministic results.

"Software distributing companies": reproducible builds generates user trust. Give your users a reason to trust you even more. Also tell them that the compiler builders are helping us to get deterministic results and increase trust; pass that on to your users.

“Academics” includes scientists who want to fully describe their software environment in a reproducible fashion in order to get published (e.g. to comply with journal requirements); it also includes scientists / reviewers who need to reproduce or verify published results, which includes reproducing the software environment.

Journals have reproducibility requirements that are sometimes satisfied by shipping a binary blob (e.g. a Docker image). Repro builds is a more principled and thus more scientific approach to satisfying these requirements. Strategy: publish an academic paper introducing best practices that include a motivation for reproducible builds.

People often follow scientists, so publishing something on RB in journals might help motivate compiler builders as well.

Transcription of posters:

Marketing

Poster 1: WHY?

- Users:
  - End user needs to bge made aware of the issue
  - I don't want to hide what were doing
  - It feels "very important" yet hard to discover
  - Users seem to be unaware that reproducibility even exits, we should show how we can benefit
  - get more users exited / use our work
  - The world needs secure software, the world needs to know this
- Users / Developers / Management:
  - Increase diversity use/contributors/areas of usage
  - Take responsibility for how easy it is to find out RB importance

+ It can be very hard to explains the importance of RB

- Developers:
  + Sustainability of R-B
  + Get more developers involved
  + Needs more contributor -> progress
  + There is a lot of software with RB problems -- We need to encourage more people to care and help
  + Change developer mindset
  + Improve buy-in from other devs
- Managemnent:
  + Get upper management care more about R-B so devs can work on it
  + Encourage funding
  + We need more "management buy-in"
  + We should be able to aquire resources more easily if we are r-b ...: build farm
  + Publicity can provide a "professional" context to motivate projects dedicated to RB (e.g. academic publications)
- Free software community
  + our collaboration is fun & productive, other areas of free software can learn from us
  + RB can strengthen the free software movement / effort

Poster 2: WHO?

- Scientists depending on reproducible software environments for reproducible research. (5 votes)
- HPC users e.g., chip-?-manfacturers
- Banks and other businesses that are critical in our society (1 vote)
- Industry (e.g. aeronautics)
- ... who (as a user) would greatly benefit from RB?

- Computer sciences / academy people

- Upstream developers, especially if we already have a simple patch.

- Compiler builders (2 votes)
  - gcc developers (1 vote)
  - go developers
  - Free software (upstream) devs so them get the relevance for their whole project
- Free software developers (1 vote)
- GNU maintainers
- npm developers
- developers who want to be sure to have the exact same build environment across a team

- Free software user (1 vote)
  - Security conscious floss users
  - User who are enthusiastic about free software, who want to be sure that the sources they have really correspond to their binaries (1 vote)
  - Users of medical devices
  - Security conscious users who want to avoid running backdoored software
  - Political activists

- Funders who support repprobuilds work (1 vote)
- Free Software foundations and companies

- Distributions (both os and source binary software ones) (2 votes)
  - Canonical (1 vote)
  - RedHat Management (1 vote)
  - Microsoft, Oracle, Apple
  - Github, Gitlab, ... companies

TOP THREE AUDIENCES TO FOCUS ON:

1. Compiler builders
2. Software distributing companies
3. Academics

Day 3

Working sessions V

Defining terminology: reproducible, bootstrappable, reliable

Pad1
====

Directed graph we came up with in dot:

```
digraph concept_relations {
    reproducible -> practical
    reproducible -> tamper_proof
    reproducible -> zero_trust
    bootstrappable -> zero_trust
    bootstrappable -> tamper_proof
    reliable -> practical
    reliable -> tamper_proof
    reliable -> bootstrappable
    reliable -> build_failure_is_tampering
    build_failure_is_tampering -> isolate_external_failures
    practical -> bootstrappable
    practical -> reproducible
    zero_trust -> security
    zero_trust -> scientific_soundness
    zero_trust -> resource_sharing
    zero_trust -> proven_good_result_without_rebuilding
    resource_sharing -> practical
    proven_good_result_without_rebuilding -> practical
}
```

We had concepts:

reproducible: bit by bit identical results

reliable: build instructions either work or fail consistently

boooststrappable: bootstrappable from source, or bootstrappable to identical binaries trough different bootstrap paths.

We identified goals:

security

zero trust

scientific soundness

practicality

tampering detection

isolation of external failure causes

Pad2

====

Reproducible, deterministic:

- binary identical outputs

Replicability:

- strict replicability, get the same result even using different methods
- does this exist in build systems?

Reliable

- the build instructions always work

Trust

- Good: Not needing to trust, zero trust, trust less
- Bad: needing to trust

Bootstrappable

- reduce the set of blindly trusted inputs

SOURCE_DATE_EPOCH specification: Overview and improvements needed

```
SOURCE_DATE_EPOCH
=================================================

- current spec talks about "current time of system";
  however generated files that end up in final output
  --> clamping: old files stay old, but newer files
  get set to source date epoch

  This is existing practise already, so we should
  update the spec to reflect this.

  + Note that this is can also be considered as following
    from the currect specification; but not everyone agrees,
    so clarification might be helpful.

  + Discussions on if the following one-sentence mathematical
    summary is useful as guiding idea

    "The output of the build should look as if the build
     had happened instantly at SOURCE_DATE_EPOCH."

    --> Will use that sentence as motivational summary
    in the introduction (but not as part of the definition)

  + git not setting the time of checked-out files to that
    of the last commit that changed them (---> clamping
    is a problem here; forbidding clamping would make
    this a non-problem)
```

- Debian and Suse take SOURCE_DATE_EPOCH from dates in
  Changelogs; Archlinux doesn't have souch a canonical
  source.

  What they do is take current time, but store the value
  in their buildinfo file; so the buildinfo file effectively
  becomes part of the source file.

  Discussion on whether this seems fine (the spec allows
  a buildinfo file to be considered source; but the majority
  of this discussion group considers it preferable to take the
  time from version control).

- discussions on the overwriting change.

  + "newer" value should not be in the future for clamping
    to still work

  + the example could be made more clearer

  + bazel drops all timestamps and environment variables to
    have better caching. OK? Does the spec have to allow
    it? (Technically, a suprocess does not see a value
    set to a parent process, nor a newer value)

  + subprocesses with modified SOURCE_DATE_EPOCH not
    properly insolated

- Setting SOURCE_DATE_EPOCH in the presence of newer dependencies

  --> set to the maximum of all dependencies?

Setting up build environments for reproducibility

Complex Build Environments

fdroid is building android apps. requires lots of software to
build. for every different app it needs a different set of sdks. each
app defines it's own build environment requirements. metadata gets out
of sync. provides a build server VM that includes most of the
potential versions that each app might require (~30GB
image). loosening strict dependencies might be too much work, and more
divergent from upstream could lead to specific bugs. app itself may
not specify all necessary build dependencies; fdroid meta-data is
needed to specify this additional information.

automated integration tests against newer versions compared against
the specific versions.

fdroid currently uses a single build server VM.

goal: reduce the set of implicit dependencies in the base image. move
more into the app-specific metadata. this could reduce the size of the
base image, and have more people using it to test builds.

build a minimal build environment and require more explicit build
dependencies in the fdroid metadata. this allows to detect implicit
dependencies when builds fail.

explore containers instead of virtual machines.

some of options such as debian have a base environment in a chroot, which installs build-dependencies at build time(sbuild, schroot, pbuilder). guix/nix have a similar concept by design.

containers are good for doing the environment setup, and virtual machines are a possible implementation, but most expensive. the software provided by the virtual machine may not need to be the same software used in the build environment.

issue of isolation for security vs. clean build environment are separate issues.

get rid of a single monolithic image.

containers make it easy to do builds with network isolation.

definiting a "seed set" for the virtual machine by splitting parts of the build environment into different "packages".

fdroid has manifest metadata for packages. can tell gradle what flavor of app to build, fields tells if there are any pre-built binaries in the build, some binaries in the source tree may need to be ignored (verified built from source elsewhere).

focusing on gradle would handle most of the packages in fdroid; it's easy to extract various metadata using gradle plugins. only a handful use maven. quite a few use ant.

need a way to get a list of build-dependencies for an offline mode. this is difficult for android applications, because the dependencies are gotten from gradle.

build environments should be a minimal build environment indepdendent of the host system. It is recommended to have no network access, in order to ensure network inputs do not result in irreproducibility.

What is needed to run rebuilders?

What do we need to have/run rebuilders?

Who does rebuilds already?
==========================

- One attendee runs a rebuilder for GUIX (~1k packages).
- One attendee has rebuilt individual packages.
- One attendee has rebuilt individual Debian packages using the published `buildinfo`.
- Tails developers rebuild Tails ISO images before they are released.

Example scenario
================

What's the simplest, cheapest and useful rebuilder?

- Documentation:
  - required hardware resources (per architecture)
  - constraints on rebuilders (e.g. to protect against malicious code in the build rules)

- It should be as simple as (for a Debian rebuilder):

```
apt install debian-rebuilder
```

- By default: try to rebuild Debian unstable (⇒ actionable feedback), generate & publish `buildinfo` files.

  We need canonical ("official") `buildinfo` files published by

Debian and we'll use them to rebuild.

We will likely need to use snapshots.debian.org to get the desired version of the build-dependencies.

In the future building stable might be good too but right now it's not useful (we know it's not reproducible and we can't fix it).

- What if I have N systems I want to turn into rebuilders? Will they coordinate to share the work or duplicate it?

- Is it OK to broadcast by default the fact you're rebuilding Debian? Actually we don't need this: apart of the signing key and stuff in the `buildinfo` we don't leak much.

What does the concept of rebuilding mean?
==

For example, do we try to reproduce a previous build in a build env. that's as close as possible to the previous one, or do we "just" rebuild and compare the results? Or do we do the latter and then if it doesn't match, we retry with a build env. closer to the `buildinfo`?

Pie in the sky option: encode in the source (e.g. `debian/control`) the requirements for building reproducibly? E.g. having source package X declare `X-Repro-Build-Dep-Same: libfoo` specifies that a rebuild of this package with a different version of `libfoo` than what's in the `buildinfo` file likely won't be reproducible.

How do we rebuild?
==================

Do we want a centralized/generic rebuilding infrastructure?
Or specific to the project we're rebuilding?

How/where do we store results of rebuilds?
==

(both matching and non-matching)

Prior art:

- https://buildinfo.debian.net/ received `buildinfo` files from
  some Debian package developers & automated builders
- Fedora has a system where developers can confirm a package works
  (e.g. on N architectures); this is not about build reproducibility
  but it is an example of multi-party validation

Do we want a shared database of rebuilds? Standardize? Centralize?

- SPOF is a problem. Each rebuilding org could publish their own
  `buildinfo`.
- For complex queries like "what package was built by package X"
  we need access to all such databases?

How do we lookup reports?

Lookup by hash? This only tells us that someone reproduced a build,
but it doesn't tell if someone got a non-matching build.

Lookup by project namespace + project-specific info, e.g. distro + (package, arch, version)?

Marketing
=========

How do we convince orgs to run rebuilders?

- Have a few high-profile rebuilders (e.g. CCC, Wikimedia, ACLU) who can each explain their own reasons.
- The argument can be tailored to every potential rebuilder.

Bonus features & ideas
======================

- Reuse this for the "I want to rebuild all packages locally before installing" policy.
- Reuse this for the "if rebuilders don't agree, then rebuild locally" policy.
- Publish the metadata for a partial mirror that only contains packages I could successfully reproduce. But the actual packages would not be hosted here, they would be fetched from regular mirrors.
- Is it OK to rebuild only subset of the packages? In the discussion about end-user/sysadmin policy we simplified and assumed rebuilders would build all packages. OTOH every rebuilder will start from having rebuilt 0 packages.
  Build every package once, or potentially more than once (e.g. building random packages)?
- Allow rebuilding only package uploaded signed by a specified list of DDs.
- When local build does not match, publish binary artifacts or at

least diffoscope. They could be garbage-collected based on how much disk space was allocated to the rebuilder by the sysadmin. F-Droid rebuilder publishes diffoscope output for non-matching builds.

Working sessions VI

Mapping out our short- and long-term goals

(by decreasing order of priority)

One year
========

 - Reproducible hackaton #2
 - Reproducible summit #4
 - `.buildinfo` files published by Debian
 - tools to compare & aggregate `.buildinfo` files
 - cross-distro <https://tests.reproducible-builds.org/>
 - cross-distro `issues.git`
 - The Unreproducible Package™ covers more than 42 issues and 10 languages
 - we have a logo, stickers and t-shirts

Two years
=========

 - rebuilder run by recognized organizations
 - Debian Policy says "packages MUST be reproducible"
 - 99% of Debian Buster is reproducible
 - we are recognized as a trusted organization by official projects
 - OS installation images are reproducible
 - Qubes OS is reproducible
 - users can set their r-b policy in a Debian release
 - easy to rebuild packages from Debian stable
 - Debian wiki pages are obsolete
 - easy to answer "where was this binary built" from, by whom, how?

(souce code, build environment)
- change of paradign of the "free software" definition → software is only free if it can be rebuilt reproducibly
- the reproducible builds project is very well-known in the FOSS community worldwide
- FLROSS ?
- GPL v4 requires reproducible builds

Distant goals
=============

- 100% reproducible packages (verifiable)
- proper bootstrapping i.e. guaranteed reproduciblity from scratch without binary blobs
- a commercial interest in reproduciblity builds (expressed as requirement for procurements)
- reproduciblity non-free sofware?
- expand reproduciblity builds to more distros and systems, more adoption, reproduciblity packages are the default, being non-reproducible is considered as a bug by a broad community
- secure building networks / decentralized and community building
- no talk with "reproducible" in the title at FOSDEM
- reproducible Debian in space

Unknown
=======

- reproducible Lineage OS

How to onboard new contributors

related: https://pad.riseup.net/p/reproduciblebuildsIII-newcomerdocumentation

What are the goals of a newbie?

What problems do newbies have to get started?

problem: spread out resources ML, git, alioth, wiki

want: web-page with "gzip" in red and "gzip -n" in green

want: find the reproducibility status of packages in other distributions => is it reproducible, what patches are required?

=> make rb status pages more discoverable

=> needs UI designer to make tiny hidden links more visible

=> FAQ how to find information about status of a package in a distribution ; also past rb bugs + patches relating to that package - if possible also tools patches

=> more advertising of toolchain patches

For developers, wanting to get reproducible results from make

=> describe your build environment

Q: how to make my software reproducible?

How to check if my software is reproducible?

  reprotest, tests.r-b.o, reproducibleopensuse tools, describe manual process

How to find out why not?

  look at diffoscope output for relevant strings and use them to locate relevant source: grep -r "Build date: %s"

  investigate where output files come from and if any of the inputs differ, recursively

  check for other sources listed in theunreproduciblepackage

How to fix?

  refer to README files in theunreproduciblepackage and other rb docs

Maybe: How to make sure that it remains fixed?

  (e.g. re-test every month)

How to ask for help with the above?

  ML, IRC, useful information to include

=> add call for action

=> add "for developers" / "how to get involved" page to r-b.org

"Resources" is wrong for ML => Community/Communication channels

People without own software:

What can I do to help?

=> work on r-b tools, fix debian packages and FLOSS upstream projects, work on test infrastructure

=> use "reproducible-check" on your Debian system

Good: want to discover about the subject with direct links to videos, link to event page

News blog under debian.org instead of r-b.org, news on r-b.o is not updated, can appear orphaned

=> move ?

separate marketing updates from developer information?

r-b.o/docs are read and somewhat useful, but maybe too theoretical or not well-ordered.

Next steps: add "get involved", "for developers", "for distributors" page

Tools section: links to debian package of reprotest, disorderfs, strip-nondeterminism instead of generic project website

=> de-debianize, have tarballs discoverable # pypi is outdated anyway and not as useful

Identifying next actionable steps for marketing outreach

Overall marketing goal: Get more people to embrace, support and implement reproducible builds

Audiences we had identified in the Marketing I session:

- Compiler builders
- Software distributing companies
- Academics
- Developers (added post-session)

we expand on the audiences we previously identified:

- toolchain developers
- companies that produce and distribute software
- universities / academics
- developers in general
- governments and policy makers
- powerful users (software consumers)

For each audience we want to identify:

- Why reproducible builds are valuable to them, from their perspective
- Example of success stories we want to see happen
- Where to reach them, both online (lists, etc.) and offline (conferences, etc.)
- Next steps and action items

Developers in general

=========

Developers:

- those who develop their own software

- those working with other software

Values:

Here is my software, it build reproducibly, and I am proud of it.

Higer quality, interest of the user and interest of other developers. User pressure is a good makreting value point.

I am a dev I didn't know how to prove that my work is rb. With reprotest I can achieve that.

Action items:

- get documentation to support devs to know how to use reprotest in this regard.

- documentation re sending patches to upstream

Toolchain developers

=========

They have 2 key issues:

- toolchain is not reproducible

- it doesnt work deterministically

Values:

- trustworthiness from other developers

- it's easier to debug a toolchain that is well defined

- reproducibility improves resource sharing (because all resources are equal)

- benefit to their users

Example of success story:

- continuous integration systems

Action items

- Find instances in which something bad happened and could have been prevented. Search for bug reports to find these cases and write a post-mortem of the bug report.

- GUIX: direct outreach with upstreams. reach sevral language communities

- Conferences: FOSDEM, also small ones

    Talk: good for ppl who already know about rb

    Table: even better espec. for folks who don't know rb yet

- Have a logo and get a banner for booths

Academics
=========

We're focussing on scientist, and particularly on disciplines whose
published results rely a lot on software.

There's an ongoing "reproducibility crisis" in science.

Some leading journals (e.g. Nature) now require software that a paper
relies upon to be FOSS.

These people already have a concept of "reproducibility", which is
good. Sadly it doesn't match ours. How do we bridge these 2 concepts?
We have some scare stories that prove that non-reproducible software
can lead to non-reproducible papers:

 - benchmarks are only reproducible if the benchmarking software can
   itself be built reproducibly

- a version string is not enough to describe what software one shall
  use to reproduce results: e.g. two R binaries built from the same
  source, but with different versions of build-dependencies, can
  produce different results

ACTION: write "The Unreproducible Paper" (similar to "the
unreproducible package") that shows how so-called reproducible
research results can't be actually reproduced if they rely on
non-reproducible software.

Where to find interested people who can have some leverage: computer
science, ACM, fields that already care a lot about FOSS
and reproducibility.

Companies
=========

We focused on companies that produce and/or distribute software.

Value of RBs to them

- Save costs and time to remediation for security problems
  (interestingly, these arguments work for internal and proprietary
  software as well)

  - reduce scope of QA upon incremental changes: why are you testing
    what has not changed?
  - faster builds and automated tests ⇒ shorter feedback loop for

developers ⇒ more enjoyable and productive workplace

- Increase trust existing & future consumers can put into your software i.e. competitive argument vs. other FOSS vendors (this requires the software to be FOSS in the first place)

  E.g. RHEL vs. CentOS: why should I trust the RHEL binaries if I can't reproduce them?

- Lower the incentive for adversaries to target your build systems and the people who run them. Incidentally, this allows you to allow less trusted people (e.g. junior sysadmins) to manage your build system.

- Companies that produce/distribute software also consume some ⇒ the marketing speech tailored for companies that don't produce software works here too.

Success stories

- We're told that Microsoft has shortened a lot their feedback loop and time to remediation thanks to RBs.

  ACTION: quotable reference needed

- Bazel (Google) and the many companies that use it

- Turn high-profile organizations (NASA, Tesla, Google etc.) into success stories / advocates:

ACTION: find out if they use reproducible builds; if yes, why?
⇒ we get a success story and probably new selling points;
if not, then it's a good case for arguing in favor of RBs.

Funding reproducible builds work

Funding for RB - Mapping Session

3 steps: Mapping the status quo of resources currently flowing into RB; identifying what is working well and where there is room for improvement, finding solutions to the needs that need to be addressed

Funding consists of 4 major groups:

material and infrastructure (sponsored hardware: servers/cpu, free tools: CI, repositories)

money: one-off funding and sustained funding, travel support for conferences and meetings.

Time: Volunteer work; paid interns through Summer of Code programs, Outreachy etc; convince your boss to let you spend some of your working hours to work on RB; fix a client's/company's problem and get paid for it

Community: Working on documentation, advertising & spreading the word, mindshare; offload work from Debian maintainers to upstream and get them involved in the idea

Material and infrastructure is working well.

Money is somewhat erratic, it's not certain how sustained funding will and can be.

Time is the most scarce resource, and it's closely interconnected to Community, to which there hasn't been a large effort so fart.

Solutions:

Material/infrastructure:

get the donor to maintain

Money:

form a task group to focus efforts for fundraising > frees up developers' time

funding might be easier to get for a formal organizational entity/structure like a foundation - lots of work

Bounty Hunters

Time & Community

delegate

call for volunteers > explicitly ask for time (vs. implicit in the project)

ask employers to donate time explicitly and implicitly (e.g. through making their software reproducible)

focus on practical benefits for RB

spin software scandals caused by non-RB software practices, make use of public exposure of the topic

find a person to coordinate volunteers, build a community (esp. GSoC interns, they need more than being allowed to work to support you afterwards)

Sell RB to government advisory groups - do the beer drinking hand shaking thing

Explain RB - don't say what it is technically, but what it wants to achieve and what its impacts will be

badges for projects to display their reproducible pride

be present at events, do meetups, to engage a wider audience

Working sessions VII

Enabling cross-distro reproducibility

repro-builds

DDC: Diverse Double Compilation

We have a failing test case for reproducibilly build GCC via DDC on guix

The ultimate DDC would be being able to do cross distro DDC

Why do we want to have the same binary for gcc on debian?

Golden bootstrap compiler that everyone agrees on. This "golden bootstrap" compiler becomes the root of the software compilation DAG.

Other binaries may be bit identical universaly. For example a bit identical TOR would be lovely. However, gcc is a good start.

Recomendation: gcc 4.7 (which doesn't require C++ support to compile)

For a canonical TOR, you need to agree on library versions.

Goal:

For equal build infos do we get equal binaries? We should have a script for comparing buildinfo files and determining if they are equal.

Current state:

At the moment, buildinfo is not a meaningful cross distro data format.

Goal:

Rather then declaring build-info the defacto shared standard. We should define the semantic space that any given build-info like format needs to store and the create tools for converting between the formats.

Current state:

Buildinfo currently specifies some of the data, but not all of the data, required to define a build environment.

Current state:

Things are worse then they seem cross distro. Even if we specify the build environment percisly in terms of which library version was used. We still need to normalize the builds of all build dependencies.

Current state:

At some point Suse had a non-polished reproducibly building gcc patch. This patch was not submitted.

New goal:

Don't solve the build dependency comparison problem yet, because we don't know how to solve it. Solve the SIMPLIST case. Try to create a golden tinycc.

How much work can we share in a canonical build process for a bootstrapped binary identical tinycc?

Should we have a shared build system or a shared recipe and multiple build systems?

This recipe should work with any c compiler/multiple compilers.

We want to specify only the parts of the recipe which actually effect the output. But not specify the entire "hash" of the exact system used to build.

```
-------------------------------------------------------

This is DDC (Diverse double compilation)

|-------------|     |-------------|     |-------------||tinycc source|  == |tinycc source|  == |tinycc source|
|-------------|     |-------------|     |-------------|

      ↓                   ↓                 ↓

|-------------|     |-------------|     |-------------||any c compile|  != |any c compile|  != |any c compile|
(any c compiler)
|-------------|     |-------------|     |-------------|

      ↓                   ↓                 ↓

|-------------|     |-------------|     |-------------|
|tinycc bin 1 |  != |tinycc bin 2 |  != |tinycc bin 3 |
|-------------|     |-------------|     |-------------|

-------------------------------------------------------

|-------------|     |-------------|     |-------------||tinycc source|  == |tinycc source|  == |tinycc source|
|-------------|     |-------------|     |-------------|
      ↓                    ↓                 ↓

|------------|     |------------|     |------------|
|tinycc bin 1 |  != |tinycc bin 2 |  != |tinycc bin 3 |
|------------|     |------------|     |------------|

       ↓                    ↓                 ↓

|------------|     |------------|     |------------|
| tinycc bin  |  == | tinycc bin  |  == | tinycc bin  |
|------------|     |------------|     |------------|
```

We want to create a reproducible build path for tinycc which works on any POSIX environment.

We want to make the paths hyperdiverse. Basically fuzz testing on this process.

Try on many computers even running the same system.

End goal: Create a common bootstrap method and start comparing hashes.

Create a bootstrap@home project where anyone with some spare CPU cycles can run the script and report back the hash they got.

Exploring reproducibility for Mac and Windows

Cross-building reproducibly

For Windows

For Windows, it is possible to use the mingw-w64 toolchain. There are packages in debian (and more distributions), but it is quite old and therefore more

recommended to build it from source (from the Git repository).

possible action item: update the debian package

This toolchain is still actively maintained, and notably used to build Firefox for Windows. It is possible to build against the official MSVC run-time from Microsoft, which can be freely downloaded from the Windows website.

Instructions can be found at:

https://git-rw.torproject.org:builders/tor-browser-build.git

(there is a README file)

There is a configuration file for each component, written in YAML. A tool called "rbm" parses the YAML and takes the steps required to build (and is already capable to create containers).

(I can picture some sort of integration with Salt to auto-create build VMs)

This toolchain can apparently also be built through pkgsrc (the NetBSD packages) on a number of different platforms (cross/mingw).

For Mac OS X and iOS

Apple doesn't seem to care about Reproducible Software. They use their own toolchains internally, which they do not distribute.

However, it is possible to:

 * cross-compile for Mac OS X with clang;

 * use Xcode to build reproducibly for iOS (except when signing)

In the same repository as above, there are also instructions for cross-building for Mac OS X. It downloads and leverages the official SDK for Mac OS X.

It is technically possible to run Mac OS X in a VM, although it is apparently illegal. The Mac OS X installer can be leveraged to generate bootable removable media to install it. VirtualBox supports it explicitly: there is a host profile for Mac OS X on Intel 64-bits.

A container technology is also available for Mac OS X.

General notes

The path to the toolchain used can end up in the final binaries (eg GCC).

Reproducibility can be useful for performance, eg if people can more easily share binaries because they know they are generated identical (caching). The

real-world use-case brought up involves computer clusters shared among many users.

What does code-signing means in terms of reproducibility?

code signing
==

General area of this session: source code
signing, e.g., signing tar balls, git
commits, etc.

Summary of ideas that came up; no definite
conclusions of the session unfortunately.

- Goal: know that this code comes from upstream
  (i.e., I have the same code as upstream)

- besides upstream there are also the steps
  by packaging; also an upstream has more than
  one developer

- F-droid is building from git commits, so
  commit/tag signing does a good job there

- reproducible builds is about getting things
  in the open (backdoors now have to be backdoors
  in source), so provinence could be a next steps

- Packages can ship upstream tarballs and upstream
  signatures; should that be part of the buildinfo
  file?

  + useful for a rebuilders if they detect malicious
    things in the code

- Management of key? (I.e., what do all the signatures
  buy me if I don't know how are those persons?)

  + at least maintainers etc can show where they got
    the code from

- Using it in the update process? (I.e., only automatic
  updates for upstream signing releases?)

- Useful for sharing the load of maintenance (e.g.
  maintainer hand-over explaining in a formalized way
  what the upstream signatures are supposed to look
  like)

- trust chain from upstream to the binary on my machine
  - reproducible builds bridges the gap between the binary
    and the source package
  - code signing can bridge the gap from upstream to the
    source package

- cross distro comparision? "Is the upstream source of
  XYZ version ABC the same as that in all other distros"

- only trust keys that have been around some time (new
  keys a cheap, so people can always restart a new identity
  under a new key)

- non-distributable binary blobs make distribution upstream signatures useles

- upstream tarball rerolls... *sigh*
  (should be a hit on upstream reputation!)
  (Maybe an argument to distribute signatures in source
   packages)

- whole different project: chains of signatures (i.e.,
  releases signatures should also sign the signature
  of the previous release)

- buildinfo are already necessary for forensics; so
  why not add the remaining things used for forensics
  (singatures allowing to get source code provinence)
  Optional field in buildinfo files for the
  content of an upstream signature?

Prioritizing the minimum viable set of tools needed for end users

- user-provided list of reproducers (represeted by URL, name, signing key)

  - a reproducer provides signed statements for each package they have built reproducibly

- apt queries each reproducer with these bits: origin, name, architecture, version (this is unique across all versions of debian)

  - lookup by hash is problematic because we cannot distinguish between "failure to reproduce", "FTBFS", and "not tried/built yet"

- reproducers return either a signed statement of "not tried/built yet" or a signed hash of the output (e.g. their buildinfo file)

Even nicer:

  - upload the .buildinfo file, because it contains origin, name, architecture, version, and hash; this avoids the problem of looking up by hash alone. Downside: more bandwidth used. ¯_(ツ)_/¯

OPEN QUESTIONS:

  - can we mirror the signed buildinfo files to avoid centralising this?

  - What happens if the reproducibility status changes over time?

  - can we (eventually) avoid these rebuilder servers by looking up .buildinfo files on buildinfo.debian.net that are signed by entities the user trusts?

ACTION ITEMS:

  - mock up a simple web server that always returns the hash + reproducibility status; this requires either read-access to the binaries or access to a database with records of binaries + hashes (e.g. the debian archive) ---> do this on localhost with access to the local deb cache (obviously on localhost that's going to be an exact match, each time)

  - set up a second web server that returns a random hash once in a while

  - write an apt-hook that contacts the simple services and verifies their responses --- unless the package is in the blacklist.

  - design the API the web server ought to implement and through which the client communicates with the service.

  - do this over TLS so that installation details are not leaked to the limited number of reproducers.

- for proper testing we really should have an actual rebuilder, but until then we can download the buildinfo files from coccia.debian.org (FTP master mirror) and serve them when queried by hash (or buildinfo)

Transcription of poster notes

day 3 PM

User Policy/Implementation

(@left top)

n = 3 trusted rebuilders

k = 2x signed buildinfo

```
(@middle)                  (@right top)
Signed .buildinfos           coccia.debian.org (ftp master mirror)
==================      <--- buildinfo.debian.net?
* signed statement: (?)
* reproduced by: 6
* dissent by: 2
* not found/not built yet 2
 ^
 |     X NO!
 |
 |
 |
 |
```

(@left corner)

buildinfo query

===============

* by hash - only finds _agreements_, not dissent
* by domain-specific unique ID:
  - debian: $pkg, $vers, $arch
* by buildingo! (contains the above)

```
(@right corner)                                 (@also right corner)
Debian main                                     Rebuilders
===========                                       ==========
archive/mirror                                 * ccc
.deb ------------->     * k of n agree on hash? <--- * nsa
                    (n = fixed number)            * aclu
                   * all must agree?
                                         * signed .buildinfo
```

Discussing current status and potential improvements in regards to .buildinfo files for rpm and iso

fedora has multiple .buildinfo files which can be fetched by a webservice

opensusebuildsystem has a file with list of installed packages and their version. "buildenv"

.buildinfo files are to new, so rpm build utilities doesn't create it.

There is an upstream of RPM, but every distribution applies additional patches.

rpm should create .buildinfo.

There is a starting point at https://github.com/woju/rpmbuildinfo as shell script which

produce similiar files than debian.

Debian .buildinfo can be found at https://manpages.debian.org/unstable/dpkg-dev/deb-buildinfo.5.en.html

Task: find someone who do RPM development or someone from redhat, fedora, opensuse

Task: show `rpmbuildinfo`

Add checksums (sha256) of the inputs into the .buildinfo files.

We discussed about where to put .buildinfo file. We suggest to have a seperate file, but it could be controversal for the rpm project.

iso

###

The iso contains of:

- "classic" Build Environment
- Packages which are part of the installer (bootloader, installer binary, dependencies of the installer)
- Packages which are on the disk as part of the Repository, which the installer will use to install.

means -> .buildinfo must contains all 3 sets

Brainstorming the reproducible builds logo design

Transcription of notes on poster

Style guide

Font ?!

Stickers

tshirt (Hoodie)

Logo "urgent" for 34C3

CSS, styleguide not so urgent

CSS

fav ico

color palette, r-b.o vs tests.r-b.o

Logo on webpage

(un)reproducible Logo/Icon

0% - partial - 100% (differents logos for grades of reproducibility)

black background

Dec 1st 2017, "nice to have"

free license

good logo more important than 34c3

timely feedback

Budget for tshirts

Budget for Logo

Announce steps via email, include announcement for IRC 30min meeting 3 days later

3 step process:

- First ideas
- Changes/Choice

  - Final

1st Step:

  3-4 choices

  1 list of deliverables

Reviews by Open Design Community too