Reproducible Builds Summit V

December 3 – 5, 2019 Marrakech, Morocco

Event Documentation

Table of Contents

Agenda

Day 1

Tuesday, December 3

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.

Getting to know each other

Each participant was invited to talk with another participant they had not met yet.

Project Updates I (Session notes start on page 8)

Several facilitators hosted short discussions, sharing updates about different projects and topics.

- OpenWRT
- ArchLinux
- In Toto / TUF
- F-Droid
- RB Core, Take I

Break

Project Updates II (Session notes start on page 8)

Several facilitators hosted short discussions, sharing updates about different projects and topics.

- Reproducible research
- RB Testing
- Microsoft reproducible work
- Supply Chain
- RB Core, Take II

Break

Agenda Hacking (Session notes start on page 9)
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.

Lunch

Strategic working sessions I (Session notes start on page 15)

- Rebuilders
- Distros
- Android toolchain
- Build inputs
- Sharing build results

Closing session

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.

Adjourn

Hack time (No notes taken)

Day 2

Wednesday, December 4

Opening 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.

Strategic working sessions II (Session notes start on page 29)

- Standards for verifying Reproducible Builds
- User experience with reproducibility
- Bootstrapping I
- Maven repository formats
- Machine readability for build information
- Classification of build inputs

Skill share sessions (Session notes start on page 55)

Lunch

Strategic working sessions III (Session notes start on page 56)

- Is my package reproducible yet
- Requirements for independent rebuilders
- Maven repositories
- Building a data dictionary buildinfo
- Verification of Reproducible Builds
- Bootsrapping II

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.

Adjourn

Hack time

Day 3
Thursday, December 5

Opening 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.

Strategic working sessions IV (Session notes start on page 75)

- Testing
- Full source bootstrapping
- Bazel
- Reproducible Builds Summit VI 2020
- Testing Maven Repository Format buildinfo spec
- Rebuilder doc review

Lunch

Closing Scrumboard Session (No notes taken in this session)

Reproducible Builds Summit VI 2020 Goals (Session notes start on page 93)

Closing plenary

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

Adjourn

Hack time

Session notes

Day 1

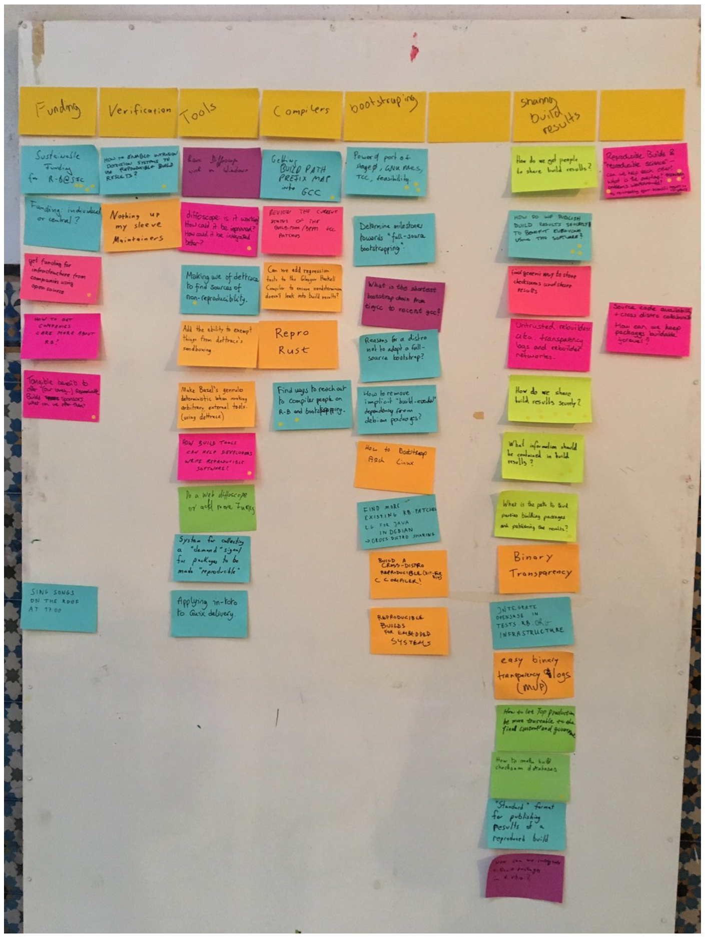
Project updates I

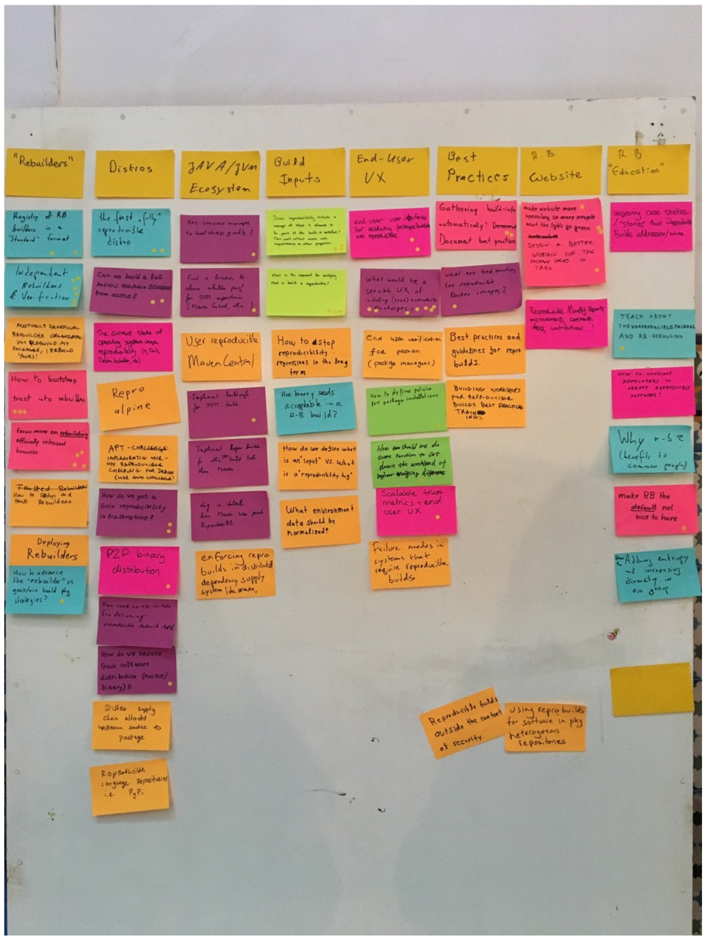
- OpenWRT
- ArchLinux
    - https://archlinux.org
    - https://tests.reproducible-builds.org/archlinux
- In Toto / TUF
    - https://in-toto.io/
    - https://theupdateframework.com/
- F-Droid
- RB Core, Take I

Project updates II

- Reproducible research
- RB Testing
    - https://tests.reproducible-builds.org/
- Microsoft reproducible work
- Supply Chain
- RB Core, Take II

Agenda Hacking





Note: numbers in parentheses indicate number of 'dot' votes placed on note

Funding

* Sustainable funding for R-B @ SFC (4)

* Funding: individual or central?

* Get funding for infrastructure from companies using open source (1)

* How to get companies to care more about RB?

* Tangible benefits to offer (our lovely...) Reproducible Builds sponsors? What can we offer them? (1)

Verification

* How to enable intrusion detection systems to use reproducible build results? (1)

* Nothing up my sleeve maintainers

Tools

* Have diffoscope work on Windows

* diffoscope: is it working? How could it be improved? How could it be integrated better? (1)

* Making use of dettrace to find sources of non-reproducibility (1)

* Add the ability to exempt things from dettrace's sandboxing

* Make Bazel's genrules deterministic when invoking arbitrary, external tools (using dettrace) (1)

* Do a web dittoscope or add more ???

* System for collecting a "demand" signal for packages to be made "reproducible"

* Applying in-toto to Guix delivery

Compilers

* Getting build path prefix map into GCC (1)

* Review the current status of the build path/BPPM GCC patches

* Can we add regression tests to the Glasgow Haskell Compiler to ensure nondeterminism doesn't lead into build results?

* Repro Rust (1)

* Find ways to reach out to compiler people on R-B and bootstrapping (2)

Bootstraping

* Power a port of staye0, GNU Mes, TCC, feasibility (3)

* Determine milestones towards "full-source bootstrapping"

* What is the shortest bootstrap chain from tinycc to recent gcc?

* Reasons for a distro not to adopt a full-source bootstrap?

* How to remove implicit "build-essential" dependency from debian packages?

* How to bootstrap Arch Linux (1)

* Find more existing RB-patches e.g. for Java, in Debian -> gross distro sharing
* Build a cross-distro reproducible (bit for bit) c compiler
* Reproducible builds for embedded systems

Sharing build results

* How do we get people to share build results? (1)
* How do we publish build results securely to benefit everyone using the software? (3)
* Find generic way to store checksums and share results
* Untrusted rebuilders aka transparency logs and rebuilder networks
* How do we share build results securely? (6)
* What information should be contained in build results? (2)
* What is the path to third parties building packages and publishing the results? (1)
* Binary transparency (2)
* Integrate openuse in tests.rb.org infrastructure
* easy binary transparency logs (MUP) (2)
* How to let top production be more trustable to the final customer and government
* How to make build checkdum databases (1)
* "Standard" format for publishing results of a reproduced build
* How can we integrate F-Droidpackages in t.rb.o?
* Source code availability and cross distro collaboration: How can we keep packages buildable forever?

Rebuilders

* Registry of RB guilders in a "standard" format (2)
* Independent rebuilders & verification (6)
* Mutually beneficial rebuilder organization. You rebuild my package, I rebuild yours!
* How to bootstrap trust into rebuilders (3)
* focus more on *rebuilding* officially release binaries (4)
* How to setup and trust Rebuilders
* Deploying Rebuilders
* How to advance the "rebuilder" vs guix/nix build pkg strategies? (1)

Distros

* the first "fullly" reproducible distro (2)
* Can we build a full Android toolchain / SDK/NDK) from source? (2)
* The current status of operationg system image reproducibility (e.g. Tails, Debian Installer, etc)
* REpro alpine
* APT-CHALLENGE Implementing one-off reproducible checking for Debian (Like Guix challenge) (1)
* How do we get a Guix reproducibility infrastructure? (2)
* P2P binary distribution (1)
* How could we use in-toto for delivering reproducible Android apps?
* How do we secure Guix software distribution (source/binary)? (1)
* Distro supply chain attacks upstream source--> package
* Reproducible language repositories i.e. Pypi

Java/JVM Ecosystem

* Has someone managed to bootstrap gradle? (1)
* Find a location to share rebuilds proof for JVM repositories (Maven Central, other) (1)
* User reproducible Maven Central
* Implement Buildinfo for JVM builds (1)
* Implement repro builds for the other FVM build tools than Maven
* dig in details how Maven was made reproducible
* enforcing repro builds in distributed dependency supply system like Maven

Build Inputs

* How to stop reproducibility regressions in the long term (2)
* What environment data should be normaliezed?
* How do we define what is an "input" vs what is a "reproducibility bug"
* Are binary seeds acceptable in a R-B build?
* Does reproducibility include a concept of what is allowed to be part of the build or installed? This could reflect source code requirements on other properties (3)

* What is the standard for verifying that a build is reproducible? (3)

End-user UX

* End user interfaces for validating packages/builds/etc are reproducible (2)
* What would be a sensible UX of installing (not) reproducible packages (7)
* End user verification for pacman (package managers)
* How to define policies for packages installation (1)
* How should we do some function to cut down the workload of enginers analysing differences (1)
* Scalable trust metrics & end user UX (1)
* Failure modes in systems that require reproducible builds

Best practices

* Gathering build-info automatically? Document best practices (2)
* What are best practices for reproducible Docker images? (1)
* Best practices and guidelines for repro builds
* Building workshops for reproducible builds best practices trainings

RB Web Site

* Make website more appealing os more people want the lights go green (3)
* Design a better interface for the package pages in t.r-b.o (1)
* Reproducible Monthly reports, improvements, comments, ideas, contributions...?

RB Education

* Improving case studies/"stories" that Reproducible Builds addresses/solves
* Teach about the unreproducible page and RB-debugging (1)
* How to convince developers to create reproducible software!
* Why r-b? benefits to common people)
* Make RB the *default*, not "nice to have (1)
* Hoto adding entropy and increasing diversity in our group

Uncategorized

* Reproducible builds outside the context of security

* Using repro builds for software in pkg heterogenous repositories

* Reproducible Builds and "reproducible science" - can we help each other? What is the overlap? terminology problems workarounds? eg. re-creating your scientific result in 10 years (4)

Strategic working sessions I

Rebuilders

What's stopping us from Rebuilding official packages?

Debian has srebuild for rebuilding a package which is a prototype and no one has picked up the work to start a rebuild. The NYU has a tool which can take a BUILDINFO and reproduce a package but there is a lack of information of where to get the list of BUILDINFO files to schedule. Debrebuild is another rebuilder tool. There is a debian rebuild setup project on salsa https://salsa.debian.org/reproducible-builds/debian-rebuilder-setup

So there are issues with publishing BUILDINFO files for Debian. (buildsinfo.debian.net and buildinfo.debian.net). This should be distributed by the normal Debian Archive. Not all BUILDINFO are signed in Debian which would be nice if it was mandatory.

Arch Linux has makerepropkg to rebuild a distributed, but no one has (yet) done the work. What is missing is a scheduler which can be based either on pacman's output or use the RSS feed to schedule packages.

Idea: put svn commit in the BUILDINFO.

Guix has no BUILDINFO files, the Guix package definition has everything required to build a package twice. Guix does not reproduce packages distributed by Guix.

There is a trust issue on the rebuilders which are rebuilding arbitrary code, but this problem can be tackled later.

openSUSE has an xml file about which packages and versions were used to produce a package and use this file to grab the packages from a non-public archive. The xml files are available via osc getbinaries and downloads it from the build service (OBS). "nachbau" is a tool to reproduce+verify openSUSE packages.
https://github.com/bmwiedemann/reproducibleopensuse/blob/master/nachbau

Only the latest version of the xml file is stored on the open build service

Openwrt has two BUILDINFO files required for rebuilding one target and not in the test infrastructure now.

Report-out points:

* Generalize the scheduler for all distro's

* Start reproducing packages from the Arch Linux in the test.reproducible-builds.org

* Debian needs to decide on a reproducer script to use on tests.reproducible-builds.org and also needs a scheduler that can drive the rebuilder

* Propose a monthly meeting for general tests.reproducible-builds.org infra

Distros

cross distros

Progress / Status-Quo

* archlinux and openwrt better integrated into testing framework
* opensuse .json results are already imported into tests.rb.o
* we send patches upstream, so they can benefit everyone
* issues and fixes mentioned in notes repo and in monthly r-b reports
* notes used on tests.rb.org to link bugs in packages
* medical industry (and others with heavy certifications) are very interested in r-b for their software

Problems

* Sharing of test infrastructure
* Sharing of patches
* No representation for flatpack/snap at the summit
* New (non-rb) tools getting more adoption means overall reproducibility in user-hands goes down
* There is a disparity between some large user-bases (e.g. Android, Ubuntu/Mint) with low r-b participation
* Lack of a walkthrough for new distros, telling them where to start and what to do

Questions

* How to get better sharing (e.g. notes repo visibility)?
* How to get more distros involved?
* What is the benefit of getting distro results on tests.rb.o?
* What distros are not involved? Why?
* Are distributions a thing of the past with containers / pypi / npm? Just moving r-b issues to another layer? But that involves more individuals and less projects - how to get them onboard? Little interest in cross-distro topics there.
* How to reach out to security-people (e.g. those attendending DefCon)

Ideas

* Better use of analogies - e.g. to food industry
* Invite "Docker" (or more generally OCI) people

http://pad.mirage/p/crossdistro

Report-out points:

* We found that there are a lot of groups / people who should be interested in r-b that do not seem involved yet
* How can we draw in more stakeholders?

Android toolchain

Debian has parts of an SDK, and their findings are summarised at wiki.debian.org/AndroidTools .

There is also an Android Rebuild project, incidentally led by a Debian developer, which provides alternative builds that can be dropped into a Google setup.

Replicant could be another source of information.

Outline

OS and SDK are one codebase, quite different to the distro package spirit...

25 GB of source code... They are spread over about 400 git repositories.

One first needs to download a file that contains the addresses. None of them are self-contained. This is handled by a tool called repo (as a layer on top of git).

The NDK has been split out and is much more clearly labelled and handled.

Release tags not obvious; version numbers:

- SDK (27)
- OS version (10)
- OS name (Q)
- Android Studio, Android gradle plugin

There are prebuilt binaries inside the source code, without instructions on how to rebuild them. (This is similar to chromium, but which does have build instructions.)

What are the components of the SDK, and how are they related?

The following are needed, but versions are not so important, except that they impact on reproducibility:

- tools

- platform-tools: adb, fastboot and a few more small tools to connect
  to the device

The following two correspond to an Android version:

- build-tools
- platforms: One of them is needed.

In Debian, there is enough to build an app, but it is relatively old.

Additionally, gradle is commonly used (bootstrapped by Debian, but with a binary seed); it might be bootstrapped from ant and a minimal gradle. People have tried bootstrapping gradle from gradle from ant, but they needed about 100 steps. Then there is the Android plugin for gradle, adding rules for creating an apk.

There is some discussion about Bazel; it can be built from source using a shell script, and it might be able to build gradle.

Torbrowser uses the prebuilt toolchain for their apk.
There is some discussion on how this could be avoided. Use binaries from Android Rebuild? One can use the (short) scripts from the project to rebuild the SDK.

There is a bug with patch in aapt2 from build-tools; having our own toolchain build would help us to port back the patch to older versions.

List of things to do:

- Rebuild versions of the build-tools, with patches backported from later
  versions.
- Bootstrap gradle from source (or from something else).
- This might require groovy.
- Package a newer version of the gradle plugin for Android for Debian;
  the blocker is kotlin.

- Bootstrap kotlin...

Closest goals:

- bootstrap gradle
- use tools from Android Rebuild for torbrowser

Build inputs

* Goals/questions we want to answer:

 * Want to clarify methodology for defining what should be an input vs not an input (input vs. reproducibility bug)

  * What should be included in the buildinfo file vs not

  * Ex: time is a reproducible bug, not an input; build path is put in the .buildinfo file even though its “just as garbage as time is” b/c it has a tranisent quality (same for username)

  * Ex: user ID gets put into .buildinfo file

 * Why is it important to include the current .buildinfo values?

  * What makes something an input vs not an input?

 * What does “input” mean (or what classes of inputs are there and how do we care/not care about them in terms of reproducibility)?

  * Potential categories:

   * Data inputs: the things that software author has direct control over, most directly affect how software works (i.e. source code, images, etc.)

   * Immutable characteristics of the environment: have to treat them as inputs because they affect the output (OS, kernel version, etc.)

   * Mutable characteristcis that we want to ignore in builds (i.e. contents of environment variables)

  * Alternative categories: Input, environment, and reproducibility bugs

 * Goal: ennumerate inputs in an attempt to come up with classifications for different inputs (see above)

  * Discuss whether immutable and mutable are suitable terms, and alternatives

  * Equivalence class terminology

   * Guix also uses this idea

   * Larger equivalence classes means smaller total inputs

 * Documenting the discussion/decisions/etc. surrounding build inputs, especially if they’ve been had before by the “elders”

* Reproducibility is constrained: reproducible build means that under the same circumstances, the same result should occur

 * Input changed means output is allowed to change

* Timestamp

 * Not something you have direct control over - making it affect the output of the build is undesirable; don’t want to think of it as an input

  * Same for other pieces of the build that you don’t have direct control over (i.e. order of thread execution, build path) -- ideally these things would not have an influence

 * Timestamp too difficult to recognize

* Source code, kernel version, OS version, etc. - information about the environment

 * Fine for pkg to be different on arm vs. x86 because the input has changed

 * If you build something with two different distros, what do you expect reproducibility-wise?

  * Cross platform build: expect to be reproducible across OSes

* Using intuition rather than a methodology makes it harder to explain

* Current buildinfo:

 * Adding something as an input was the easiest path to making a package reproducible, rather than adding it being the right thing to do (i.e. buildpath was added not because it should be an input, but because you don’t have to fix the package if you add it to the buildinfo)

* Reproducibility: given a set of inputs, expect the same outputs

 * Many complexities packaged in the word input

* Questions:

 * How to stop reproducibility regressions in the long term?

  * Right now we care about reproducibility so we patch + make packages reprodubile

  * Imagine everyone here gets busy, and in 10 years pkgs that used to be reprodubile are no longer

  * Seems like it falls under tooling, organization, etc.

 * What is the standard for verifying that a build is reproducible?

  * Deciding what is an input vs. reproducibility bug

  * Expand the question to “how can we communicate the results of the build along with its input?”

   * Standard for verifying the reproducible build results AND its inputs

   * i.e. a few rebuilders, distribute the results → should also contain the inputs and not only the outputs

    * .buildinfo file lists both inputs and checksum of binary

  * Reproducibility only makes sense in terms of multiple builds/runs

   * Theorhetically: If you knew what would make a build reproducible in terms of inputs (if it’s formalized), you could already know it’s reproducible without checking it against another run

    * In reality this is hard to achieve - easier to build the same thing twice and check that the hashes match

* Without the theorhetical view above, can only gain confidence -- motivation to have a good definition for what's an input vs. not an input
  * The current way to verify is to look at hashes, which works well for builds, but some builds produce a metadata log file (by product, not an output) -- take all outputs and crunch them together and hash them
    * Important for verifying - what output are you supposed to hash together to get the output checksum?
* What environment data should be normalized?
  * Thinking about the environment in terms of reproducibility bugs?
    * Examples of hacks/normalization:
      * Set timestamp to something static
      * Include buildpath in the buildinfo
  * Are there any reproducibility bugs that are necessary?
    * I.e. do we ever need the timestamp?
    * Install deps → time got inserted → affected the final output
  * Does this fit with the immutable/mutable classifications?
    * I.e. passing the timestamp into a rebuilder
    * I.e. intercepting the syscall and returning a fake timestamp
      * Some of the hacks like this might not adhere to POSIX semantics
    * These feel like hacks, rather than like fixing a package to make it reproducible
* Are binary seeds acceptable in reproducible builds?
  * What is meant by a binary seed?
    * Random number generator?
    * Binary input to the build?
  * For reproducibility: the answer is yes, b/c you can still have a reproducible build
  * OTOH: Tracking a binary back to source code (includes bootstrapping) is a goal for security
  * Wider question: can only auditiable things be inputs to reproducible builds?
    * Is the goal of reproducible builds to provide security, or not?
      * Focus is more on security this year than last year
      * What are some of the non security use-cases?
        * Debugability/SWE POV
        * For caching + performance optimization (i.e. in the context of Bazel)

* Corresponding source: knowing where the binary came from (could have ethical, practical applications outside of just security)

* Security is not part of the definition of reproducible builds, but more a benefit

* OS/Kernel version:

* Is this an input or reproducibility bug?

* One thought: Would be sad if binary hash differs because you upgraded your kernel version by one version

* Are there other things that come from the kernel that lead to non-determinism in the binary?

* Rely on kernel developers to make sure this doesn't happen

* Breaking change: ptrace events returned changed (whether a build would realisitcally use that, probably not, but could happen for something else?)

* Guix thoughts: project tried to answer this, came up with

* Create environment as specified by the derivation, which says name of kernel, architecture, recursive derivations it depends on

* Effectively a graph of buildinfos

* Is this effective to capture the environment?

* Approach 1: different variations of environment (timezone etc.)

* In nguix and guix: fix everything (i.e. force a particular environment)

* Ignore variety of kernel versions (i.e. Linux but version doesn't matter)

* Specific pieces of software rely on specific features of Linux kernel

* Usually ok that not all "inputs" are captured, those require special attention

* Is this documented? Guix manual explains derivatives

* What criteria is used to decide what to include or not?

* Practical choice

* Deadtrace (dettrace?), debian is different than guix approach of holding things constant → seems like it would be good to reduce the set of inputs (huge input set means reproducibility, but does that mean anything?)

* Goal: what is the minimal set of inputs that allow for reproducibility

* Do we need a standard for what counts as a build input, or is it ok to vary from distro to distro, subcommunity to subcommunity?

* The set of build inputs affects what reproducibility means for a build

* Best practices/documentation of rationale for the contents of the .buildinfo file (would help for future projects to create a .buildinfo, like for JVM)

* Minimal set of inputs makes the applications of reproducible builds (above) easier
* Two approaches: capture the env (nguix/guix), vary the env (debian)

Report-out points:

* Come up with best practices for what imputs are important for rebroducible builds
* Document, enumerate current things included in info files
* What are alternative approaches that currently exist?

Sharing build results

Separate build results exist:

* Debian has centralised data available, with json export format consumed by people
* Guix has local data on individual machines
* Arch & SUSE have official results

but no shared format, every case is really different, very distribution specific

in addition, for centralising data, idea of signing each buildinfo file means adding one file each time

then idea of having centralised log of rebuilds would avoid this cost when many rebuilders

Big high level questions, as a rebuilder:

* how can I know who already rebuilt the package, in similar conditions (distro, arch, ...) that give a chance to get the same result?
* did I get the same result than anybody?

question: does sharing a DB between distros have an interest (chance to get same result) or is it only about sharing formats and tools?

what if each big rebuilder publishes his results separately? is it usable?

Debian experience with many individuals shows issues

centralising DB could be creating a target of attacks: distributing the DB would better distribute the trust in data

need a registry of DBs, with definition of ways of searching and output format

the search key will have many fields: package name and version, but also arch, distribution, build environment, ...

need to define a shared storage of data, securing access to data published by multiple trusted parties (companies, organisations, individuals, ...)

next conversation: how to locate shared rebuild info in the next 6 months as a working proof of concept?

before trying to create something really ambitious that will remain up for years includes the question of what does the information look like

"where should the buildinfo files go?"

post mortem thought by holger/pearce: lookup buildinfos not by software+key but also by hardware model

Report-out points:

* GUIX has split (?)
* What do we want to have working in 3-6 months?

* Where did all the build info flies go?

Day 2

Strategic working sessions II

Standards for Verifying Reproducible Builds

Morning session:

This was a question in the build inputs session.

What does it mean to verify a reproducible build? This is a "rebuilt" or a "verification" build, you normalise the environment.

How is it reproducible? If the hash matches.

Debian policy bug discussion the policy should include "the packages should be reproducible". It's easy to define if a package is not reproducible, but hard to define that it's reproducible.

Do you trust yourself? If you build everything yourself, your build environment might be comprimised.

Each package could have a list describing the variations under which it's reproducible.

Who should be able to verify a build? Anyone who can install a package, should be able to verify a build.

Standards often have a MUST, and SHOULD.

Software should have been reproduced at least 3 times, by independent organisations (maybe in different places).

If you can afford it, operate your own rebuilder.

For a user, you define the rebuilders you trust:
- Different criteria, different policies

A fixed number of rebuilders could be bad, as a third party

You don't want a reprodubile build, you want a binary that corresponds to the source

How do you see reproducible builds being used, if not by the end user:

- Reproducible builds are more interesting from a distro perspective, compared to an individual user
- Reproducible builds can be compared to the technique to the washing of food before eating it. It doesn't change the taste, but it makes it cleaner. Some eaters don't care about washing food, but some do.

- Catering to a small number of end users verifying reproducible builds will help

- For Guix,

- One independently verifies what the binary provider provides. In the Debian case, we recommend you verify it yourself independently.

We want to be "reasonably" sure that something is reproducible.

If I rebuild the software, and reproduce the same hash, and then rebuild with the German locale, you get a different hash, it doesn't show it's reproducible or non-reproducible.

"Checklist", there's a concern that it could become a binary definition that.

What could be on a checklist:
- Independent verification by a user
  - Documented tool that a user can use to rebuild a package
- Independent verification by third parties * (don't exist yet)
  -

Current checklist, we don't know how it will look in the future. Current way, it's not complete.

Is the checklist, a best practice.

We're looking for a standard for best practices. Maybe an RFC.

The Tails release notes have instructions for reproducing the build.

Tor Browser and Bitcoin-qt had some description.

For furture versions of the checklist, we recommend third party rebuilders exist.

Bitcoin and Monero publish instructions on rebuilding the binaries, and if there are differences, hopefully they'll speak up.

Different types of end users require different things (policies).

Is there a world where an end user that defines a policy, and then theres a tool they run to verify things relating to the policy.

The goal is not to make a checklist for good software, but how to evaluate when the software is reprodible.

Currently, verification has to be done by end users, but ideally in the future, end users wouldn't have to rebuild the software to verify they get the same hash.

Clarify what the current things it's possible for people to do when verifying software.

It's not possible to currently independenly verify Debian, due to lack of tooling.

Recommendation 1 is everyone verifies themselfes
Recommendation 2 is everyone verifies the hash they
Recommendation 3 you find 100 organisations you trust, then you just check they get the same hash.

Follow some documentation to build the package yourself, to build all the packages you're using.

Having some tool where an end user can rebuild a package, it's not so useful. It's the next step, users and coorporations could then rebuild software and get involved.

OpenWRT builds it's own toolchain. You have a commit hash, so you can reproduce packages.

Polish the existing tooling. You can rebuild Debian, Guix and Arch things, and also OpenWRT things where you feed in a buildinfo file.

Afternoon session:

* Reprotest: build same pkg twice, problem is that it will probably be reproducible, so it tries to vary the env to try to bring out reproducibility issues (time, locale, etc.), then run diffoscope to compare two pkgs and find out where it's not reproducible
  * Unmaintained but it exists and people do use it
  * Debian has the same idea
* Overview of previous session
  * Are we talking about deb has a hash and user wants to verify, or checking if package is reproducible against myself?
    * One approach: rebuild it yourself + check if debian gave you the right package (then you don't really need to ask debian for the pkg)
      * Third party (another user) also benefits if first user makes result public
      * This is a stepping stone to the ultimate gold standard of third party rebuilders
    * Approach two: have someone you trust rebuild the pkg so you don't have to do it
* Goal of this session:
  * Don't want to duplicate work that the rebuilders group is doing: How to rebuild (setup, document, etc.), or how to distribute/publish/communicate/obtain as the user the results?
* Summary of what was talked about last year: There was discussion that independent rebuilders are required, users need to have a way to access these results
* Canonical hashes:
  * User A builds pkg, user B builds pkg, hashes differ
    * One reason: pkg is not reproducible
    * One reason: pkg is reproducible but differs b/c build env, lib versions, etc. differ
      * Pkgs are only comparable given the same environment - there is a canonical, correct hash given the build environment?
        * Fix pkg so it becomes reproducible independent of environment, or list the inputs that affect the build in the buildinfo
        * Do we want to have a stable, default environment to compare against?
          * Not practical, maintainable
        * Or require rebuilders to replicate the environment (add tooling to make it easy for users to replicate the build environment)?
          * Idea: debian (or distro) is the official source of the buildinfo, and all rebuilders recreate environment
          * Different versions of the same pkg are not meant to reproduce the same hash

* Concern: too many different environments so there won’t be enough people that verify a particular package. Two interpretations:

  * Some version of a pkg is so infrequently used that no one else is building + validating

  * End user verification uses different build environments, so the hashes don’t match → need metadata that tells you all the things to set up, we could add tooling to do this for the user

* Ultimate goal is for end user to go to pkg manager and download the pkg for their architecture, and they want to check the hash of that binary against rebuilders → buildinfo of that pkg should be the env that rebuilders replicate to check the hash

 * First pkg in pkg manager should be reference for rebuilders

* There will probably be rebuilders for different distributions, platforms, etc.

 * The physical machine where a rebuilder runs could rebuild for different distros using VMs, containers, etc.

* Goal for this session: come up with design for end user tool to automate verification of a reproducible build

 * User in capacity of rebuilding or verifying?

  * Both: end user could be trying to verify that build is correct, or someone trying to replicate the result of debian to publish the result for others

   * One perspective/use case: Fetch buildinfo file, parse env items, replicate the env, rebuild pkg, store result (sign it? What format should this be in?)

   * Other perspective/use case: User that just wants a result and they want it to be trusted

* Design questions, thoughts, concerns:

 * What is the best format for the results to be communicated in?

  * Tool should publish the buildinfo file used to try to reproduce the build - this could be the thing that’s communicated back to the verifying user (similar to in-toto metadata)

   * User needs this to verify someone else’s results (source code + buildinfo for the pkg version)

 * How does the user know to trust the results from the tool?

  * Tool can be trusted b/c you can look at its source code

  * Does the user trust debian?

  * Do we want some third-party reproducible builds authority that publishes the public keys for trusted rebuilders? That way it isn’t a central authority

 * Tool needs to set up some sort of sandbox, chroot, or something

  * Fresh VM: make sure nothing bleeds in from a previous build, preventing influence between builds from a debugging/sanity perspective at the least (previous builds can pollute the next build)

   * The goal is different from reprotest

  * The buildinfo file doesn’t list things you shouldn’t have on your system, this could have unintended consequences

  * The sandbox could be a source of untrusted ness

 * Concern: initial setup cost, not reasonable to ask users to download 10s of GiBs of toolchain to verify (although we recognize that this is not a long-term solution)

  * Requirement: Very easy for user to set up and run the tool

 * Requirement: The tool should know how to parse the buildinfo file provided and set up an environment based on that information

  * The buildinfo must have enough info for the build to be reproducible (that’s the point of it)

 * Can we have one tool for all distros?

  * One tool is ideal

   * Centralized authority

   * Unified interface for users

   * More likely for this logic/tool to ultimately be used b a third-party rebuilder

   * What logic would be shared by such a tool? buildinfo files are different for each distro

  * Concerns about one tool:

   * More difficult to audit b/c more LoC? On the other hand it could be small enough

   * Distro-specific users don’t need a tool to support multiple distros

   * How practical is this implementation-wise?

   * Who owns the tool?

  * Alternative: every distro or pkg manager has its own tool

   * Could have similar interfaces for different tools

 * Idea for the simplest case: $ name of tool + buildinfo file + src directory → get output

* MVP: CLI tool that takes a buildinfo file, source; sets up the environment; diffs/spits out the buildinfo result for comparison of hashes

 * Concern: tool has to know how to parse buildinfo, interpret that info based on the distro in order to download + install deps correctly, run things for the pkg system

  * Potential solution: encode distro in the buildinfo file

* Is the source trusted?

 * For the sake of this discussion, yes because you can audit it (whereas binary output is opaque - you don’t know where it came from)

* From rebuilders convo:

* Tooling: How easy is it to rebuild arch, debian, etc. pkg?

* Didn’t go into sharing + verifying results

* OBS: multi-distro, can add a new distro

* Is deadtrace is something that rebuilders could run?

* Not debian specific but would require a lot of maintenance, additional features to get it to a widely useful state

* Some of the lightweight ideas could be useful (guix has something like this, env normalization and similar)

* Can the tool run without the buildinfo file?

* Same tool is good for the first build, not just for rebuilds (i.e. it outputs a buildinfo file, for comparison with future builds)

* There are existing tools that do similar things (like OBS, which has two parts, a frontend and OBS build as a backend): Get pkg, get buildinfo, get same build deps, install, and rebuild

* Some of this is distro-specific, currently input buildinfo file format is different, but the idea is a good place to start (and there are other tools that are more agnostic)

User experience with reproducibility

End User Experience

We need integration in package managers.

There's a transistion period where not yet all (important) packages are reproducible.

After this we want to prevent installation of non-verified packages. Before that we shouldn't bother users with this.

OR: We could mark a subset of packages as reproducible; only these packages would need to be verified.

It's unclear if this is a good idea.

Different Users:

- * Developer
    - * Have a report for them somewhere (online webpage,...?)
    - * Provide a website to see if my software is reproducilbe (https://ismysoftwarereproducible.org/firefox)
- * System Administrator
    - * Configuration of ebuilder/Verification Policies
- * End-User (non-technical)
    - * Don't show warnings/checkmarks
        - * Prevent installation of non-verified packages

Report-out points:

* Non-technical end users should never see any (un) reproducible build issue

* Make system administrators aware how to verify the software which runs on their infrastructure

* Make Developers more aware of unreproducible software

Bootstrapping I

Bootstrapping distros

* Session 1 (morning)

** Recap

- until recently, Guix has ~250 MiB of binary seeds (NixOS about the same, using BusyBox; Debian around ~500 MiB): GCC, glibc, Coreutils, etc.
- open to a “trusting trust” attack (as described by Ken Thompson in 1984)
- janneke started looking at “addressing” GCC:
  + wrote Mes, a interpreter for the Scheme language, written in C: https://gnu.org/s/mes
  + Mes includes a C compiler written in Scheme, MesCC

** Current status

- the “stage0” project, OTOH, provides a C-like language, M2-Planet
- reduced binary seeds to 140 MiB in current Guix: https://guix.gnu.org/blog/2019/guix-reduces-bootstrap-seed-by-50/
- GCC is no longer part of the “binary seeds”:
  + instead, Mes runs MesCC, a C compiler written in Scheme
  + MesCC is used to build TinyCC
  + TinyCC builds GCC 2.95

** Goals

- further reduce bootstrapping in Guix
- reach out to other distros and devise more general solutions
- questions:

```
    - what about the build daemon and the OS kernel? why are they ignored?
    - answer: it’s not black-and-white, it’s about incrementally
      reducing the trusted computing base (TCB)
  - do cross-distro “diverse double compilation” (DDC)
    + started by David (NixOS) and janneke (GNU Guix) at the Summit
    + managed to obtain bit-identical statically-linked Mes compiled on
      Nix and Guix
    + DDC matters mostly for the binary seeds
  - on-going work: getting rid of the Bash, sed, grep, awk, etc. binary seeds
    + Gash implements a POSIX shell in Scheme (can run on Guile, but not
      on Mes)
  - Mirage
    + about reducing the TCB
    + the OCaml compiler is “not bootstrappable”: it comes with
      pre-compiled bytecode (a binary seed)
      * this binary seed is not reproducible
      * a new binary seed is committed roughly at every release
    + an OCaml interpreter in C is being developed (“ocaml-boot”, see
      Gabriel Scherer at Inria)

** Next steps

  - janneke currently working on Gash (POSIX shell) + Gash-Core-Utils
    (replacement for GNU Coreutils + sed + grep + awk)
    + this can run on Guile, but not on Mes
    + thus, it’s good for Guix (which has Guile in its bootstrap seeds),
      but not for Nix (because it does _not_ have Guile in its bootstrap
      seeds, currently)
  - thus → we should make Gash and Gash-Core-Utils runnable on Mes
    + make Mes more capable (Guile-compatible), and/or have Gash use
      fewer Guile feature
```

- currently a long chain of compilers: MesCC, TinyCC, GCC2, GCC4, GCC7
  + how can we shrink that chain?
  + TinyCC may be able to target GCC 4.x in the future
  + there’s little hope of MesCC being able to replace TinyCC in the medium term
  + GCC >= 4.8 requires C++, which is a “hard barrier”
- currently (WIP) building older versions of sed, grep, etc.
  + could we skip these older versions?
- outreach: Debian? FreeBSD?
  + FreeBSD has no GPLv3-licensed code and avoids GPL-licensed code
  + start a dialog to gauge interest and see what could be done
  + Debian has “build-essentials” and not-well-defined “binary seeds”
  + key point: reducing the TCB

* Session 2 (afternoon)

** Recap

- how Arch does it
  + example: upstream Rust provides a binary
  + Arch compiles its own version from that
  + from there, the next versions of Rust are compiled with the previous Arch Rust package
  + Arch is x86_64-linux only
  + “base-devel” is an implicit set of dependencies, always installed in the chroot where building takes place
- how Debian does it
  + similar, but with many architectures
  + “build profiles” allow you to define variants of packages: for example, a stripped-down package for Git (without optional dependencies)

  + build profiles are conditionals in the ‘control’ file
  + “build-essentials” is an implicit set of dependencies that’s
    always there

** Way forward

 - handling intermediary steps
   + Guix has “hidden” packages, used for intermediary steps but hidden
     from the user interface
   + Debian could have a “bootstrap” section, next to “main” and “contrib”
   + Arch could have a separate package repository
 - bootstrapping C (via Mes, etc.) vs. bootstrapping high-level
   language such as Rust
 - whether or not Debian & Arch implement it right away, it’s useful
   for bootstrappable folks to understand how the Mes approach could
   work for them
 - first step for Arch & Debian: package Mes (and its dependencies:
   nyacc, mescc-tools)
   + done in Debian
   + can then be used to compile TinyCC and the whole chain
   + to do: build a variant of Mes that does not depend on Guile
 - “build-essentials” and “base-devel” will have to remain
   + create a bootstrap able to build “build-essentials”
   + have a choice between taking the historical binary seeds and
     taking those compiled via Mes
   + add a field to the Debian ‘control’ file saying “do not install
     ‘build-essentials’”?
   + Arch could introduce a “bootstrap base” that only includes the
     binary seeds needed to build Mes
   + Debian: ‘sbuild’ can be told what “build-essentials” is, and thus
     one could tell it that “build-essentials” is Mes, etc.

- incentives to work on that
  + Rebootstrap people would be interested
    * rebootstrap = re-cross-compile “build-essentials” from i386 to a
      specific architecture such as x86_64
  + porters to new architectures would have an incentive to use “just”
    port Mes (+ TinyCC, etc.) and be done with it
  + benefits:
    1. security (mitigate “trusting trust”)
    2. engineering (it doesn’t “feel right” to have those binaries,
       makes bootstrapping new architectures more difficult, etc.)
    3. user freedom (the user may want to make sure they really get
       the source code that corresponds to their binaries)
- documentation
  + have a page describing how to bootstrap C (via Mes), or how to
    bootstrap a language
  + to-do: move bits from the Guix manual and blog post and from the
    Mes manual to bootstrappable.org, as a recipe for distros who want
    to do it
  + Mes dates back to May 2016

Report-out points:

* We achieved a shared understanding of what's at stake and how different distros approach bootstrapping.

* We identified the way forward for Guix, Nix and then for Debian and Arch.

* We discussed the way forward in terms of outreach and funding.

Maven repository formats

Putting the maven repository buildinfo spec into practice

http://reproducible-builds.org/docs/jvm

* theoretical buildinfo spec for maven repositories exists

* sketch for auditing spec based on buildinfo spec

* should certification be part of it?

* seems clear that certification is a later step

* maven repos are about one owner for a given slice of the repo, so no place to publish verifcation info from sources other than the official maintainer

* we have specs, we need real world experience to learn what details are essential, and which details are noise

* rebuild jtorctl with multiple build tools

* there needs to be a good hash for the source

* is a SHA1 commit hash good enough?

* we need working examples to proceed and see what details are important

* these need to be bundlable so that downstream consumers can easily see the whole picture of a library and all its dependencies

* we ran builds with bazel, gradle, and maven on https://gitlab.com/eighthave/jtorctl

* maven and bazel are self-reproducible, gradle not

* gradle vs bazel produced different bytecode though both used JDK11, but one Debian/OpenJDK and the other Zulu OpenJDK.

Report-out points:

Afternoon session

Now that jtorctl can be built with Bazel, Gradle and Maven, let's implements plugins for these 3 build tools to generate a Buildinfo file that summaries the build result

need to share to location where the resulting code wiill be shared:

    * Bazel:

    * Gradle:

    * Maven: maven-buildinfo-plugin branch in https://github.com/apache/maven-studies/tree/maven-buildinfo-plugin

```
tasks.withType(AbstractArchiveTask) {
   preserveFileTimestamps = false
   reproducibleFileOrder = true
}
```

Machine readability for build information

Requirements for Machine-readable Build Information

Meta-rule: These requirements enable users of the data to make choices and/or take actions.

Group A: Requirements on the build information itself

* Provide a schematized description of the build inputs (*)
* Include a "point in time" identity of the source-being-built (e.g. git tag)
* Build information muiltust be separable from the built artifact
* The top N existing formats of "build information" can be semi-automatically converted to this format
* Build information is human-readable as well as machine-readable
* The syntax of the information must have one or more parser packages for common programming languages (e.g. python module, golang package, C++ library)

Group B: Requirements on mechanisms used to retrieve build information

* Info is securely distributable (auditable, attributable, non-repudiatable)
* Resiliant, globally distributed availability of build information
* Must not be bound to exactly one transport
* At least one transport for retrieval must be accessible through firewalls; effectively, HTTPS must be one viable transport)

Group C: Requirements on both the build information and the retrieval mechanism

* Extensible and versioned
* Build information should be cacheable (e.g. via CDN or web proxy)
* Build information should be searchable via multiple criteria and indexes (*)
* Smallest possible overhead to participate (i.e. lowest bar of pain-in-the-neck tasks)
* Can be used by offline (non-net-connected) consumer [serializable, sneakernet-enabled]
* Must support full internationalization and localization (e.g. Unicode-ready, standard date format)
* Conformance must be rigorously testable

* Do not be "liberal in what you accept"

(*) Underspecified at this time; needs more development

Open Questions: We were very uncertain if these should be required

* Information about build inputs should be separately retrievable from information about build output

* Must be able to tie a digital identity (e.g. public key thumbprint) to a real-world identity (human being, organization)

Build Inputs II - What counts as build inputs?

Let's talk about a few things that help us classify

source code

build dependencies

the build itself

things that are generated as part of the build

byproducts, artifacts

  --> rather tools

environment

 cwd

 userid

 envvars

 linux distro

all these things have property to be configurable

next category

 build tools like shell, version compiler

 -> these are also dependencies

other things consider environment

  time

if we go that broad we should add thread scheduling as input

scheduling is reproducibility bug

then time is also a repoducibility bug

umask (default file permissions)

locale

reprotest tests for

build path

user group

file ordering

home directory

exec path

timezone

ASLR

domain host

hardwarestuff, number of CPUs, any hardware things, cpu info

architecture

cpu version

library versions

tool versions

what part of the build toolchain is considered toolchain and what is a build dependency

looking at debian wiki for buildinfo file

https://wiki.debian.org/ReproducibleBuilds/BuildinfoFiles

Build-Environment

--> List of all packages forming the build environment, their architecture if different from build architecture, and their version.

are these only packages that are actually used

Questions for categorization of variables

how frequent do the values change

are you the programmer in control of the values

are you okay with changing the value or not

we should be careful with the term operating system
are we talking aobut kernel version, different distros, different versions of the distro
or talking about windows, macos, linux
we should stick to linux
we agree that some things, if they change, the output changes
the idea of being able to change, e.g. fake the time, we should do it, is interesting

a test to appeal to intuition

if I change this variable do I still expect it to be reproducible
----

go through list and evaluate it with yes and no

source code
 --> we'll get back to it

build dependencies
 --> hard to answer with yes and no

 --> 3 things: tool chain, build dependencies, runtime dependencies

 we should skip that one too

 how stable of an API does it promise, should count towards whether it's an input or a bug

assumption for this discussion,

same version of program is deterministic

a compiler should be deterministic

----

input not a very good name

dependent and independent variable ??

----

let's go back to list

CWD

varying should not change the build

what is the question we are trying to answer?

Official definition

A build is reproducible if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts.

----

question:

Giving the same X, it produces the same output, is it part of X.

Should a reprudicible build be robust to changes to the following:

CWD: yes

userid: yes

envvars: no

but not all, environment variables that behave like cmd variables, or like build options, but not somehting like home

linux distro: ??

very generic name, is it what uname gives you, is it the collection of terms, not very useful

compiler version: no

shell version: yes

time: yes

thread scheduling: yes

umask: ??, intuitively yes, is this a security concern?

locale: yes, ideally

build path: yes

user group: yes

file ordering: yes

home directory: yes

PATH: yes

timezone: yes

ASLR: yes

domain host: yes

architecture: no

architecture might be overly broad, ISA might be better

cpu version/model: yes

build-time library (versions): no

build tool chain (versions): no

kernel version: yes

also quite broad. there might be features that are more difficult to be robust against.

not talking about difficulty, but whether ideally it should be robust we should come back to that and discuss why we we have a different answer for compiler version

-----

-- Recap

YES:

- CWD
- userid
- shell version
- time
- thread scheduling
- umask
- locale
- build path
- user group
- file ordering
- home directory
- PATH
- timezone
- ASLR
- domain host
- cpu version/model
- kernel version

NO:

- envvars
- command line options
- config files

SEMANTIC (yes):

- compiler version
- architecture (ISA)
- build-time library (versions)
- build tool chain (versions)

----

how do we check that builds are reproducible? we do bit by bit comparison

it would be enough to do semantical comparison, but that's impossible.

the practical decision of bit-by-bit comparision, affects the categorization above

----

proximity of items to functioning of program affects categorization?

Key take away:

rb should be robust against all items that don't directly affect the semantics of the output

but there are pragmatic concerns (see semantic group)

Skillshare session

- How to build an embedded Linux system
- How to harden services using systemd (BPF, sec comp…)
- Productively annoy upstream
- How to Guix
- How to run a functional GNU/Linux distribution? (Guix)
- How to make the Glasgow Haskell Compiler (mostly) deterministic
- How to integrate Tor into all the things
- How to build a wireless mesh network
- How to build Tor browser
- How to achieve and maintain inbox zero
- How to Rust
- How to learn Rust (the hard way)
- How to use in-toto to communicate and verify r-b results
- How to know whether a rebuilder is good/official/legal. How to group up rebuilders. How to define a good rebuilder
- How to debug RB issues
- How to use Bazel (for multilanguage projects)
- How to reproduce Mirage OS
- How to use Nix/Nix OS
- How to influence the POSIX standard (IS 9945, IEE 1003)
- How to make your Maven build reproducible
- How to build and distribute packages for GNU Guix (and Nix)

Strategic working sessions III

Is my package reproducible yet?

Problem

* There are people wanting to know the status of a package, possibly for different reasons

Personas

* Upstream
* Packager
* End-User (probably also includes sys admins and advanced users)

Purpose

* Distro people: what's the status of a package in other distros or their own distro, or even my own package
* Distro people: can I pick a patch from a different distro to make my package reproducible
* Upstream: is the software I wrote reproducible or do I need to fix anything
* End-User: I'm using these 100 packages and I want to know their status
* Maybe needs an api?

Challenges

* Getting all the data (get the data from tests.r-b.org database)
* matching different package names together is a challenge
* getting users to know about it and promote it
* which architecture do we want to default to? package might reproduce on amd64 but might be broken on arm
* amd64 vs x86_64
* how to handle suites
* how to handle versions (might not be the same everywhere, we could include the version in the output)
* figure out how to provide links to the patch sets for each distro for a given package

Solutions

* generate json files on tests.r-b.org, either for packages or one file for everything
* simply have the website consume that json (server-side probably)
* <input type="text" name="package">
* <select name="distro"><option selected>any</option><option>debian</option></select>
* output: green-yes, red-no
* promote it with badges if your packages and/or software is reproducible
* badge for firefox might be half-red-half-green if it's only reproducible in some distros
* to keep the interface simple we would have "expand" buttons to show more details
* provide links to interesting information for a given package

Open Questions

* do we want to show status over time? (maybe in version 2)
* do we want a framework or a static website?
* do we want to include "since x days" info?

Report-out points:

* possible to do, all challenges seem very solvable
* based on tests.r-b.org data
* allow self-hosting

Requirements for independent rebuilders

# Independent Rebuilders

1. We want people to be able to independently verify their binaries.
2. We want "rebuilders" to perform this reproducibility verification at scale

Requirements:

- The source is available (for f droid the binary is also required to obtain the upstream signature)
- The expected hash of the output is available
- A description of the original build environment must be available
- It must be possible to reproduce the original build environment in an automated fashion
- There must be a way to publish the build results
- There must be a way for end users to discover and query the build results

Every project needs:

- a build info file
- a tool that reproduces builds
- some output that represents the reproducibility results
- a place to upload and share those results

Sharing:

- rebuilders run a server that can serve log results
  - Certificate transparency style logs are a pretty good idea
- users need a place to discover the log servers

Status across distros:

Debian:

- BUG-774415: describing a rebuilder tool. big mess. tool takes a build info file and rebuilds a package.
- Some tooling work is required.

HuaWei:

- UK Authorities require reproducible builds for Huawei products.
- Tooling exists to reproduce the builds. Non determinism is recorded and replayed on subsequent builds.
- The tool is published.
- They have trouble with 3rd party dependencies.

F-Droid:

- There is currently no easy way to rebuild FDroid packages. Two or three people could do it manually.
- There is a tool that rebuilds all of fdroid, but it is not in a state where it could be run by
  people outside of FDroid. e.g. 250GB of disk space is required to set it up.
- Some info is recorded in build metadata files, other info lives on a dying mediawiki instance.
- FDroid needs a build info file definition. These files should be published alongside the APK.
- Building and comparing a package and comparing it to the upstream binary is relatively easy, but
  the tooling needs quite some polish.

OCaml:

- Does not distribute binaries because packages are non reproducible and a rebuilder network does not exist.
- Reproducible ocaml is possible since a few days.
- Build info file needs to be defined.

- Rebuilder tool / script must be written.

- No clear ideas about publishing build results.

Nix:

- End users can verify individual packages

- Tooling exists to verify ISO images

- No tooling exists for verifying the package set on an individual system

- Discussions are ongoing around sharing results and exposing them to users

OpenWRT:

- The build info files exist

- There is a script that takes a version tag and will rebuild the specified image and check reproducibility

Guix:

- tooling exists to let users reproduce

- guix challenge compares your local store path against the binary caches

- guix has multiple build farms (main one in berlin, and a second one in france that is a little flakey)

- tooling is being developed to allow comparison of these build farms

Arch Linux:

- There is a tool to reproduce a given Arch package called 'makerepropkg' which is included in the default installation (soon)

- there is yet another tool which reproduces Arch pakages which is theoratically Distribution independent called archlinux-repro

- Rebuilder tool / script must be written

- No clear idea about publishing build results https://github.com/archlinux/devtools/blob/master/makerepropkg.in

https://github.com/archlinux/archlinux-repro

Maven repositories

Testing Maven Repository Format buildinfo spec

2 use-cases:

* FDroid wants to rebuild FDroid apps dependencies as part of automated buid, then write a script that downloads a buildinfo, launches a rebuild then checks the results for every dependency
    * links to build structure to be shared
    * hard part is that it is automated, but build environment is perfectly known (Debian stable)
* Apache Maven want to have people voting on new Maven components releases not only vote on the source release but also confirm that they got the same binary output
    * Maven release process http://maven.apache.org/developers/release/maven-project-release-procedure.html
    * example of a current vote https://lists.apache.org/thread.html/09d84424d9fbc1270b688619e33aea2394ff855bf28727421e3b33f4%40%3Cdev.maven.apache.org%3E
    * notice that build environment is wide open: every voter rebuilds on his own machine, with his own OS (some even use Windows...)

working on jtorctl,

having official published artifacts available on:

* https://jcenter.bintray.com/net/freehaven/tor/control/jtorctl/
* https://jcenter.bintray.com/org/briarproject/jtorctl/

SNAPSHOTs for our buildinfo tests should be published to JTor GitLab repository: https://gitlab.com/eighthave/jtorctl/-/packages

Future multi-module tests could be done on Jackson or Mockito

Building a data dictionary buildinfo

Building a machine-readable format for reproducible build inputs

* Goal: record enough information for someone to be able to compile software
  on different distributions (Debian, CentOS, etc.) such that you end up with
  the same output hash. Want to have enough information to provide to a
  Docker container that could reproduce the build.

  For now, only come up with descriptions that things that *should* be allowed
  to influence the output of a build if they were change. Can we come up
  with a consensus for these things? (try to take inspiration from the
  "classification of build inputs" session from earlier today)

  There are many possible things that one *could* include, but each user will
  have different things they want to include. We will find a common base
  and make the format extensible so that users can add other things
  if they so wish.

  Examples of "in-the-wild" build inputs:

  - Arch Linux: http://ix.io/23xB
  - Debian: https://manpages.debian.org/buster/dpkg-dev/deb-buildinfo.5.en.html
  - OpenSUSE: https://gist.github.com/RyanGlScott/65d1f8678611ea651efe6de5d9d64581

  Here is the classification of build inputs that should not be allowed to influence
  the output in a reproducible build system:
  http://pad.mirage/p/RBVAgendaNotes-build-inputs-II

  Mattia notes: what separates "build tool chain" from "build-time library"?
  Many Linux distributions simply lump both things into a single category.

  * Q: Should "compiler version" have a single value or multiple values?

  A: Even for a single package (or single object file!), multiple compilers
    may be used to produce it! So it has to be a multi-valued answer.

Similarly, multiple "build tools" may be used (e.g., compilers for both
C and Go in a single package).

It's not enough to capture *just* the "build deps" and "version".
You need to look at the whole toolchain!

* The Principles of Versioning in Go
  (https://research.swtch.com/vgo-principles):
  Versioning is really hard, and you have to think about it hard
  to get it right. How do you properly resolve versions?

Insight: It's likely that not all packages will adhere to whatever
format we come up with. In such a scenario, it may be simpler to
change the package to fit the format, not the other way around.
- Such a thing happened with fontconfig, which had to be
  changed substiantially to become reproducible

Some tools cannot have different versions coexisting in the same
build (e.g., autotools, which will break if you have two different
versions in use at the same time).

Sometimes, even relying on a particular version number is not enough.
e.g., `gcc-9.1.3` can mean different things depending on the Debian
patch level.
- Perhaps we should have "blessed versions" of packages
  that are intended to rebuilding?
- Not sufficient to just record the compiler version. You need to
  track the full provenance of the compiler. That additional

information can be necessary!

Thought: could we teach tools to dump out this information?
`gcc` does this in a limited capacity, perhaps we could teach
other tools (e.g., `ld`) to do so as well.
- This imposes a potential cost for embedded developers,
  since the overhead and including all of this information would be high. We
  might need to have an option to strip out build information
  if desired.

We could keep as much information as possible and stripping out
unneeded information where necessary. On the other hand, this might
be prohibitively costly in terms of size.
- Holger notes that adding a buildinfo file to every Debian package
  caused the number of inodes to run out! There are often unexpected
  consequences like this.

Even old software can be challenging to support properly.
e.g., fixing a bug in `tar` caused the order of files to change.
And `tar` bugs were being fixed as recently as this year!

Report-out points:

* We have not produced the machine-readable format that we originally envisioned.

* This session was a "successful failure", because we have gained a much deeper understanding of the problem space and the challenges we would have to overcome in order to create the format.

Verification of Reproducible Builds

Standard for verifying reproducible builds

Morning session:

This was a question in the build inputs session.

What does it mean to verify a reprodubiel build? This is a "rebuilt" or a "verification" build, you normalise the environment.

How is it reprodible? If the hash matches.

Debian policy bug discussion the policy should include "the packages should be reproducible". It's easy to define if a package is not reproducible, but hard to define that it's reproducible.

Do you trust yourself? If you build everything yourself, your build environment might be comprimised.

Each package could have a list describing the variations under which it's reproducible.

Who should be able to verify a build? Anyone who can install a package, should be able to verify a build.

Standards often have a MUST, and SHOULD.

Software should have been reproduced at least 3 times, by independent organisations (maybe in different places).

If you can afford it, operate your own rebuilder.

For a user, you define the rebuilders you trust:
- Different criteria, different policies

A fixed number of rebuilders could be bad, as a third party

You don't want a reprodubile build, you want a binary that corresponds to the source

How do you see reproducible builds being used, if not by the end user:

- Reproducible builds are more interesting from a distro perspective, compared to an individual user
- Reproducible builds can be compared to the technique to the washing of food before eating it. It doesn't change the taste, but it makes it cleaner. Some eaters don't care about washing food, but some do.

- Catering to a small number of end users verifying reproducible builds will help

- For Guix,

- One independently verifies what the binary provider provides. In the Debian case, we recommend you verify it yourself independently.

We want to be "reasonably" sure that something is reproducible.

If I rebuild the software, and reproduce the same hash, and then rebuild with the German locale, you get a different hash, it doesn't show it's reproducible or non-reproducible.

"Checklist", there's a concern that it could become a binary definition that.

What could be on a checklist:
- Independent verification by a user
  - Documented tool that a user can use to rebuild a package
- Independent verification by third parties * (don't exist yet)
  -

Current checklist, we don't know how it will look in the future. Current way, it's not complete.

Is the checklist, a best practice.

We're looking for a standard for best practices. Maybe an RFC.

The Tails release notes have instructions for reproducing the build.

Tor Browser and Bitcoin-qt had some description.

For furture versions of the checklist, we recommend third party rebuilders exist.

Bitcoin and Monero publish instructions on rebuilding the binaries, and if there are differences, hopefully they'll speak up.

Different types of end users require different things (policies).

Is there a world where an end user that defines a policy, and then theres a tool they run to verify things relating to the policy.

The goal is not to make a checklist for good software, but how to evaluate when the software is reprodible.

Currently, verification has to be done by end users, but ideally in the future, end users wouldn't have to rebuild the software to verify they get the same hash.

Clarify what the current things it's possible for people to do when verifying software.

It's not possible to currently independenly verify Debian, due to lack of tooling.

Recommendation 1 is everyone verifies themselfes
Recommendation 2 is everyone verifies the hash they
Recommendation 3 you find 100 organisations you trust, then you just check they get the same hash.

Follow some documentation to build the package yourself, to build all the packages you're using.

Having some tool where an end user can rebuild a package, it's not so useful. It's the next step, users and coorporations could then rebuild software and get involved.

OpenWRT builds it's own toolchain. You have a commit hash, so you can reproduce packages.

Polish the existing tooling. You can rebuild Debian, Guix and Arch things, and also OpenWRT things where you feed in a buildinfo file.

Afternoon session:

* Reprotest: build same pkg twice, problem is that it will probably be reproducible, so it tries to vary the env to try to bring out reproducibility issues (time, locale, etc.), then run diffoscope to compare two pkgs and find out where it's not reproducible
  * Unmaintained but it exists and people do use it
  * Debian has the same idea
* Overview of previous session
  * Are we talking about deb has a hash and user wants to verify, or checking if package is reproducible against myself?
    * One approach: rebuild it yourself + check if debian gave you the right package (then you don't really need to ask debian for the pkg)
      * Third party (another user) also benefits if first user makes result public
      * This is a stepping stone to the ultimate gold standard of third party rebuilders
    * Approach two: have someone you trust rebuild the pkg so you don't have to do it
* Goal of this session:
  * Don't want to duplicate work that the rebuilders group is doing: How to rebuild (setup, document, etc.), or how to distribute/publish/communicate/obtain as the user the results?
* Summary of what was talked about last year: There was discussion that independent rebuilders are required, users need to have a way to access these results
* Canonical hashes:
  * User A builds pkg, user B builds pkg, hashes differ
    * One reason: pkg is not reproducible
    * One reason: pkg is reproducible but differs b/c build env, lib versions, etc. differ
      * Pkgs are only comparable given the same environment - there is a canonical, correct hash given the build environment?
        * Fix pkg so it becomes reproducible independent of environment, or list the inputs that affect the build in the buildinfo
        * Do we want to have a stable, default environment to compare against?
          * Not practical, maintainable

* Or require rebuilders to replicate the environment (add tooling to make it easy for users to replicate the build environment)?

* Idea: debian (or distro) is the official source of the buildinfo, and all rebuilders recreate environment

* Different versions of the same pkg are not meant to reproduce the same hash

* Concern: too many different environments so there won't be enough people that verify a particular package. Two interpretations:

* Some version of a pkg is so infrequently used that no one else is building + validating

* End user verification uses different build environments, so the hashes don't match → need metadata that tells you all the things to set up, we could add tooling to do this for the user

* Ultimate goal is for end user to go to pkg manager and download the pkg for their architecture, and they want to check the hash of that binary against rebuilders → buildinfo of that pkg should be the env that rebuilders replicate to check the hash

* First pkg in pkg manager should be reference for rebuilders

* There will probably be rebuilders for different distributions, platforms, etc.

* The physical machine where a rebuilder runs could rebuild for different distros using VMs, containers, etc.

* Goal for this session: come up with design for end user tool to automate verification of a reproducible build

* User in capacity of rebuilding or verifying?

* Both: end user could be trying to verify that build is correct, or someone trying to replicate the result of debian to publish the result for others

* One perspective/use case: Fetch buildinfo file, parse env items, replicate the env, rebuild pkg, store result (sign it? What format should this be in?)

* Other perspective/use case: User that just wants a result and they want it to be trusted

* Design questions, thoughts, concerns:

* What is the best format for the results to be communicated in?

* Tool should publish the buildinfo file used to try to reproduce the build - this could be the thing that's communicated back to the verifying user (similar to in-toto metadata)

* User needs this to verify someone else's results (source code + buildinfo for the pkg version)

* How does the user know to trust the results from the tool?

* Tool can be trusted b/c you can look at its source code

* Does the user trust debian?

  * Do we want some third-party reproducible builds authority that publishes the public keys for trusted rebuilders? That way it isn’t a central authority

 * Tool needs to set up some sort of sandbox, chroot, or something

  * Fresh VM: make sure nothing bleeds in from a previous build, preventing influence between builds from a debugging/sanity perspective at the least (previous builds can pollute the next build)

   * The goal is different from reprotest

  * The buildinfo file doesn’t list things you shouldn’t have on your system, this could have unintended consequences

  * The sandbox could be a source of untrusted ness

 * Concern: initial setup cost, not reasonable to ask users to download 10s of GiBs of toolchain to verify (although we recognize that this is not a long-term solution)

  * Requirement: Very easy for user to set up and run the tool

 * Requirement: The tool should know how to parse the buildinfo file provided and set up an environment based on that information

  * The buildinfo must have enough info for the build to be reproducible (that’s the point of it)

 * Can we have one tool for all distros?

  * One tool is ideal

   * Centralized authority

   * Unified interface for users

   * More likely for this logic/tool to ultimately be used b a third-party rebuilder

   * What logic would be shared by such a tool? buildinfo files are different for each distro

  * Concerns about one tool:

   * More difficult to audit b/c more LoC? On the other hand it could be small enough

   * Distro-specific users don’t need a tool to support multiple distros

   * How practical is this implementation-wise?

   * Who owns the tool?

  * Alternative: every distro or pkg manager has its own tool

   * Could have similar interfaces for different tools

 * Idea for the simplest case: $ name of tool + buildinfo file + src directory → get output

* MVP: CLI tool that takes a buildinfo file, source; sets up the environment; diffs/spits out the buildinfo result for comparison of hashes

 * Concern: tool has to know how to parse buildinfo, interpret that info based on the distro in order to download + install deps correctly, run things for the pkg system

  * Potential solution: encode distro in the buildinfo file

* Is the source trusted?

 * For the sake of this discussion, yes because you can audit it (whereas binary output is opaque - you don't know where it came from)

* From rebuilders convo:

 * Tooling: How easy is it to rebuild arch, debian, etc. pkg?

 * Didn't go into sharing + verifying results

 * OBS: multi-distro, can add a new distro

* Is deadtrace is something that rebuilders could run?

 * Not debian specific but would require a lot of maintenance, additional features to get it to a widely useful state

 * Some of the lightweight ideas could be useful (guix has something like this, env normalization and similar)

* Can the tool run without the buildinfo file?

 * Same tool is good for the first build, not just for rebuilds (i.e. it outputs a buildinfo file, for comparison with future builds)

* There are existing tools that do similar things (like OBS, which has two parts, a frontend and OBS build as a backend): Get pkg, get buildinfo, get same build deps, install, and rebuild

 * Some of this is distro-specific, currently input buildinfo file format is different, but the idea is a good place to start (and there are other tools that are more agnostic)

Bootsrapping II

(See Bootstrapping I, page 39)

Day 3
Strategic working sessions IV

Testing
tests.reproducible-builds.org re-design

The current state is a bunch of Python and Shell scripts which generate HTML and run reproducibility tests

Paul came up with an OpenWRT proposal for a newer build infrastructure:

docker container spinning up the openWRT tests which outputs a JSON file with metadata which tests.r.b.org

can import into the database. This makes the build "de-centralized" since the projects data get's pulled into

the test infrastructure.

Two different build types:

* CI build which builds a package twice once current and once in the future and compars
* A verification build of the released package

Matia is open to having a JSON importer for build results for the official infrastructure.

Fdroid wants to do a verification build on the test infrastructure, it is hard for Fdroid to determine which architecture the apk is when it is downloaded.

How do we reproduce ISO and how does it fit in the Database Model? Possibly a specific suite.

Presentation

Is there a need to record a queryable JSON endpoint for the tests.reproducilebuilds.org setup. An option would be to provide distro wide packages

Changes in the tests.reproducible.builds.org infra:

* Change the homepage to show all projects which are tested on the test infra
* Make the project pages all use the same template for HTML generation
* Make the page at least a bit mobile friendly
* Make a cross distro reproducubility overview
* How do we distinguish between verified builds and CI builds

Do we want badges on Github/Gitlab badges for projects to show the reproducible builds status.

Tests.reproducible.org

The main tasks:

* scheduler
* builder (reproducilbity packages)
* HTML generation
* JSON generation

Report-out points:

* Apacar propose a JSON format on the mailing lists
* Apacar proposes to work on the database to make it more flexible to support arbitrary binaries

Full source bootstrapping
extreme bootstrapping from the source

Goal: cut the TCB by removing the Linux kernel from it

http://guix.gnu.org/blog/2019/guix-reduces-bootstrap-seed-by-50/

- guix is not much there yet
- we should do something about kernel and userland and the guix daemon
- started with a remark by valgrant, "initrd is just a guile script"
- ludo showed "can we run the guix build scripts on mes?" it's only two scheme file
- combined: guile in initrd, two files to build a guix package
- we should be able to write an initrd that only depends on mes http://savannah.gnu.org/projects/mes
- "are you moving the bootstrap over to hurd? getting rid of TCB"

- bootstrapping from initrd: no build daemon (which could be malicious and inject stuff into output)

- reducing TCB and decreasing binary seeds
  - guile (binary seed), switching guile for mes -- not sure what we gain? (guile is big, but switching is lots of work)

- looked at https://bellard.org/tcc/tccboot_readme.html - 15 years ago fabrice build a bootstrappable linux iso
  - patched linux kernel
  - using tinyCC build with glibc

- guix plan:
  - bootstrap initrd with e.g. tinyccboot bootstrapped, from there build guix/etc.
  - or alternatively use hurd/mach

- the relationship between TCB (one thing bad to compromise the system) vs trusting trust (double compilation)

- current guix TCB: linux kernel, bash, guile, tcc (environment + packages around0

- with double compilation: can we go from highle implausible to impossible?

- re-compiling tcc won't solve the bootstrapping story

- "demonstrate that buliding stuff with the guix daemon on linux is the same as in initrd"

 - outcome is not that we can build everything from initrd, but there's an equivalence between initrd and linux-daemon built packages

- "only build mess and boot0 in initrd, compare checksums"

 - flag "build in a vm in initrd" instead of a container with the daemon

  - to get security, we nevertheless need to run it on real hardware

- atm "guix-initrd" is not real (but it will be soon)

  - take a derivation, topologically sort it and sequence it

  - leading to a script which can build without the daemon being run

- initrd

 - all the source tarballs

 - guile

 - binary seeds

 - not think with guile in mind, but think with mes -- it is separate

- a plan (for 2020?) to get mirageos bootstrappable

  - in a unikernel (after unix-based prototype)

  - use OCaml bytecode as binary seed (2.5MB), plus C programs: compiler, linker, assembler

  - bootstrap OCaml (requires ocaml + C compiler, linker, assembler -- maybe a different service for starters)

  - bootstrap Coq

  - bootstrap compcert, extract from Coq

  - compile opam + mirage + ...

  - compile the source of the unikernel

- back to the microkernel, building stuff on Linux vs building stuff on HURD
  - hurd could be much smaller as TCB
  - build environment: nothing but mach, needs file system service (or key-value)
  - current state: the daemon uses the posix personality
  - could be more fine-grained: nothing (no fs, no processes, ...) by default
    - no network access needed
    - first one to build is gnu make, runs configure (forks a lot), build script (file system + processes)
    - need fork() from the beginning
    - mes + mescc could compile in a single process, mescc c library could provide fork() etc.
      - mescc forks to run the assembler (third-party, but part of mescc)

- first: needs to do a package build in initrd
- now we explored that the builder should have a flag to build in a vm (initrd/qemu)
  - it is trivial! ;)

- bootstrapping mescc on FreeBSD, likely needs some configure tweaking

Bazel

Bazel and DetTrace

Dettrace: Tool for automatic reproducible code.
- Uses container to sandbox enviornment
- Enforces determinism at the system call and CPU instruction level.

Bazel: build system that structures rules for building.
Also does some lightweight sandboxing, statically knows the dependencies for a build and all files of the build.

Genrules: Allows calling arbitrary commands outside the build system.
    Bazel cannot enforce determinism for genrules.
    This is where DetTrace comes in?

Idea: DetTrace + Bazel to automatically fixing nondeterminism.
    Bazel allows DetTrace to run in parallel at the "leaves"
    since Bazel enforces that parallel builds can only access
    their files, and not other parallel builders in the build.

Problem with DetTrace + Bazel:
- DetTrace does not support Java or other languages with threaded runtimes.
  Omar recommendation: Allow threads to run in parallel in DetTrace for DetTrace

How should DetTrace allow selectively turning off an on features?
- Strace already has some categories for selectively turning off features.

DetTrace overhead:

- 2.5X to 3.5X for builds
- Overhead of DetTrace is linear to the number of system calls
  intercepted.
- Omar expects DetTrace would be used with far less system calls
  intercepted than they are now, so the overhead for the
  common use would be far less.

Ryan asked what is the best way to combine Bazel with DetTrace?
Ryan's current approach is hacky.

Bazel builds may not work because different versions of Bazel
are not compatible for builds.
Bazel is not compatible due to many breaking changes, lots
of changes in the last few years. Bazel expects to make
fewer breaking changes in the future, but not promises of
no more breaking changes.

Bazelisk automatically downloads the correct version of
Bazel.

Package that tries to be unreproducible.
https://github.com/bmwiedemann/theunreproduciblepackage

What is the advantage of Bazel + DetTrace:
- Better guarantee of reproducibility.

Newer build systems inspired by Bazel:
- Bucks (Facebook) and Pants (Twitter)

Build System comparison:
https://ieeexplore.ieee.org/abstract/document/8255774

Bazel provides a non-turing complete, deterministic language.
https://docs.bazel.build/versions/master/skylark/language.html

DetTrace Publication:
Reproducible Containers ASPLOS 2019

Paper draft will be posted on RB General soon-ish
Code will be found on github, link not available yet.

Timestamps are the biggest source on unreproducible builds
in Guix. Other people agree that DetTrace should do a far
minimal tracing for the "biggest" known sources of unreproducibility
in packages.

People interested in packaging Guix with DetTrace.

Reproducible Builds Summit VI 2020

Misc points:

- Venue was a bit difficult year.
- Maybe Mexico next year, it's close to the US, but not the US.
- Get away from the winter schedule
- Maybe 2 events in a year

Would be good to have a hacking event, then another event with talking and hacking.

Having two events would fragment participation, as it would be hard for people to come to both.

A decentralised hacking event organised around a bigger event might work well.

A hacking event in Morocco (this venue), the venue could be sponsored, not the travel, which reduces costs.

Having a venue where academics could publish a paper would enable participation from those people.

3 days before FOSDEM, there's a Mini Debconf, but it was organised a bit too late for 2020.

Back when the X Windows system had it's own conference, it was in Boston (USA) in January. January in Boston may be offputting. It made it easy to get a venue.

Maybe more structured working next year.

Maybe structured working, "Scrum board"

Doing more than 8 days is a stretch.

Scheduling an event in the winder may not fit in with US holidays.

Thinking of an event in October/November, avoid December.. Thanksgiving is a little earlier next year in the US.

There's convinience in December as people are less busy.

Competition in October/November is fierce.

18 to 20 of November is the Aspiration conference.

Mazfest is in October, which takes 2 weeks of Aspiration time.

There are good event calendars for free software events, so we should figure out what other events there are, and can we get on them.

Loved stuff:

- Location in Marakesh, Morocco
- Connecting people on the mailing list was good
- Staying at the venue, no need to travel here every day
- There was two women, builds solidarity

Challenge:

- Other stuff to think about, worry about travel tips, traveling as a woman
- Traveling solo
- Holger doing too many things

Suggestions:

- More online coordinatino in advance
- Coorinating meeting up at the venue, and traveling to the venue
- Having a checklist for private bathrooms... would be good
- Getting the pre-event logistics emails out sooner, two months ahead

Action item: Spreading the organisation of event more, getting Mattia involved in the spreadsheet stuff.

Hard to get other people involved.

Hopefully having a bigger team will allow having a checklist for the organisation.

These summits are very intense, people didn't hack in the evening in Paris. People spreading out and heading to resturants.

Several Debconfs, accomondation was in the venue.

Mexico, a Debconf was offered by someone working in a a university there

Costa Rica is another option, good infrastructure, easy to get to, Aspiration has local partners. Really convinient location in Latin America, reachable from South and North America. Direct flights from London/Spain.

Iceland, expensive, crazy expensive.

China could be an option. The Great firewall could be an issue. There's maybe services you can pay for in China to get less restricted access. Beside the Internet problems, many people are based in Europe, China is far away.

Many people don't want to go to the US, the visa and immigration issues are bad.

Coordinating with a company? Visiting Hua-wei, there could be an introduction of what Hua-wei does.

Some hotels in China had different access to the internet, with less restrictions.

Writing requirements would be good:

Staying in the venue would be good

Unrestricted access to the internet

Venue and event checklists would be a good idea.

Outreach, it would be good to have more women attending. Super willing to have a follow up conversation about outreach.

There may be women in accedemia that might attend. It's harder for men to do outreach. Emails were sent to ~20 women for this event, but emailing doesn't achieve much.

Sage Sharp, has a consultency, reaching out to them could be useful in.

Equity is better than diversity, giving people a stake.

Other perspectives present can allow better analysis of the points made.

Reproducible builds folks hang out on the mailing list, 95% of the mailing list contributors are white men, and most people working on reproducible builds were white men.

In protest organisations, if you want people to come to your events, you should go to their events.

There are sysadmin organisations involving women, maybe getting involved there could be constructive.

Building relationships beyond trying to get people to come to your event is good.

Asking if there are folks in their communitites that could benefit from being in a space to talk about Reproducible Builds, and then when they're here, get involved in the event including leading sessions.

As a women and person of colour, reminding people about participation guidelines is important. Bringing pepole here and making sure they're OK is really important.

The other conversation that would be good to have, what is the way that we explain the importance of reproducible builds to other groups. How to explain to a human rights activist about the importance of reproducible builds.

It can be difficult to get the elevator pitch to be understandable.

We had sessions on explaining reproducible builds at previous summits, but we should have more discussion about the benefits of reproducible builds to other groups.

Different kind of equity conversation, who can we invite to hold space to talk about the benefits of reproducible builds to people they know about.

There are lots of people who work on reproducible builds who work on it from a hard problem. The bootstrapping problem is hard, and people can work on it just because it's fun to work on.

Tor is an example of good tension between the technical and the political/real world effects of the work.

We should translate the website to other languages to make it more accessible.

There is no translation of the website currently.

There's the proecss convo and the translation convo.

One thing done in California, everything is facilitated in English, but the sesions take place in Spanish. Naturally people will settle on a language.

There are lots of companies, including ones in China, engineers from those could be invited.

Localising the website is a tactic, holding the event in multiple languages is more difficult.

China could be a priority because of the mistrust.

The US government funds lots of work on Tor, and Reproducible Builds

Testing Maven Repository Format buildinfo spec

2 use-cases:

* FDroid wants to rebuild FDroid apps dependencies as part of automated buid, then write a script that downloads a buildinfo, launches a rebuild then checks the results for every dependency
    * links to build structure to be shared
    * hard part is that it is automated, but build environment is perfectly known (Debian stable)
* Apache Maven want to have people voting on new Maven components releases not only vote on the source release but also confirm that they got the same binary output
    * Maven release process http://maven.apache.org/developers/release/maven-project-release-procedure.html
    * example of a current vote https://lists.apache.org/thread.html/09d84424d9fbc1270b688619e33aea2394ff855bf28727421e3b33f4%40%3Cdev.maven.apache.org%3E
    * notice that build environment is wide open: every voter rebuilds on his own machine, with his own OS (some even use Windows...)

working on jtorctl,

having official published artifacts available on:

* https://jcenter.bintray.com/net/freehaven/tor/control/jtorctl/
* https://jcenter.bintray.com/org/briarproject/jtorctl/

SNAPSHOTs for our buildinfo tests should be published to JTor GitLab repository: https://gitlab.com/eighthave/jtorctl/-/packages

Future multi-module tests could be done on Jackson or Mockito

Rebuilder doc review

Rebuilder doc review (2019-12-05 ~10:30)

Goal of session: review documentation and notes from prior session in this summit and past summits.

May not be the best use of time at this time; possible better to coordinate in-between summits.

Develop a plan of summarizing and strategizing between summits?

Coordinate summary and plan of next steps on a per-distro basis in github/gitlab/etc? A place with revision control and mechanisms for comment and review.

prior summits:

https://reproducible-builds.org/events/paris2018/report/
https://reproducible-builds.org/events/paris2018/report/#Toc11360_331763073
https://reproducible-builds.org/events/paris2018/report/#Toc11374_331763073
https://reproducible-builds.org/events/paris2018/report/#Toc11432_331763073

https://reproducible-builds.org/events/berlin2017/ReproducibleSummitIIIEventDocumentation/
https://reproducible-builds.org/files/ReproducibleBuildsSummitIIReport.pdf
https://reproducible-builds.org/files/AspirationOTFCommunityLabReproducibleBuildsSummitReport.pdf

moved on to talking about next steps to move towards making NYU rebuilders experimentally usable:

- upload apt transport .deb to debian experimental
- explore why so few packages are built
- solicit more builders?
- document, iterate and fix!

explore other rebuilder setups

irc meetings about deploying and implementing rebuilders

Get some people from NYU to come to other universities, schools, etc. and do workships on setting up rebuilders and documenting the process!

- connect with people at google to set up rebuilders and do something similar

Generate the documentation by doing more implementation!

Existing rebuilder and rebuilder verificaiton for Debian + in-toto

https://salsa.debian.org/reproducible-builds/debian-rebuilder-setup

https://github.com/in-toto/apt-transport-in-toto

Closing Scrumboard Session

No notes taken in this session

Reproducible Builds Summit VI 2020 Goals

* Write a script that shows % of installed pkgs that are independently verified

* Make the [core] repository of Arch Linux Reproducible and Verifyable by end users

* Have a working setup for an automated rebuilder (Debian) that is documented well enough and known so that everyone with resources can run it

* Debian: upload all pending r-b patches by RBS6 and get unstable to 90%

* 6 Rebuilders of different distros by RB6

* Independent Nix rebuilders running + sharing results

* Debian Rebuilders with 100% test coverage for stable & testing

* Enable Guix users to require reproducible builds from multiple build farms for the packages they download

* Implement something to collect Tor Browsers build results and publish them

* Guix master's bootstrap binaries are (Guile+) 500 bytes hex0

* Mirage OS unikernel with minimal binary seed which compiles itself ~ bootstrap! < 5MB!

* Have the Guix Data Service up and running with several (1,3) build farms, giving the percentage of reproducibility

* Guix supports user policies on binaries and providers several build farms

* Send out RB6 invitations by Augst 1st 2020

* Have a public Debian rebuilder running at & by Microsoft, Google, $elsewhere

* Get connection with any other rebuilder/system from Huawei to get share buildinfo

* Make better tools in Bazel to diagnose / fix RB issues

* Convince Aache board to require RB checks

* Talk about hosting a rebuilder at Google, have a plan of action towards this

* Ship independently reproduced / verified packages to F-Droid client and verify them there

* Deploy RB-enforcing proxy on F-Droid production builders

* Start mapping the OSTREE reproducibility problems (flatpak, system upgrades)

* By RB6, there at least 3 rebuilders regurarly reporting via tests.rb.org

* Prototype scalable resilient buildinfo distribution