Reproducible Builds Summit IV

December 11 – 13, 2019. Paris, France

Event Documentation

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

Table of Contents

Agenda

Day 1

Tuesday, December 11

10:00 Opening session (No notes taken in this session)

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

10:30 Getting to know each other

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

10:45 Sharing knowledge about reproducible builds (No notes taken in this session)

Participants gathered in small groups which included both folks who were at previous Reproducible Builds Summits and newcomers.

The talking points suggested for their discussion were:

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

11:05 Break

11:20 Project updates (Session notes start on page 9)

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

- Sustainability of the Reproducible Builds project
- Reproducible openSUSE
- Tool sets
- Boostrap ability
- How to achieve user trust and adoption?

- Testing framework
- Reproducibility by design

12:30 Lunch

13:30 Agenda brainstorming (Session notes start on page 12)

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.

14:40 Break

15:00 Strategic working sessions I (Session notes start on page 20)

- Tooling
- Compilers
- User verification
- Terminology
- Bootstrapping
- Rebuilders
- Mapping out hardware level issues
- Java reproducibility

16:10 Break

16:25 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.

16:55 Adjourn

17:00 Hack time (No notes taken)

Day 2

Wednesday, December 12

10:00 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.

10:20 Strategic working sessions II (Session notes start on page 41)

- User stories
- Rebuilder security
- Bootstrapping II
- Java and JVM languages
- Hardware II
- Cross distribution participation
- Trust models
- Tools matrix

11:45 Break

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

- How to comment your code correctly
- How to <anything> Alpine
- How to analyze (non-existant) open source business models
- How to Rust
- How to write Haskell
- How to use a SAT solver to resolve dependencies
- How to stay on top of email follow ups
- How to <verb> Nix(OS)
- How to user IPFS?
- How to do budget forecasting

- How to conduct blameless postmortems
- How to automatically test GUI (OpenQA, dogtail)?
- How to ... OPAM? Ocaml package manager
- How to organize an event
- How does a cheap wifi router work?
- How to use the autoprovenance tool
- how to package Arch Linux
- How to develop an OS with minimal trusted code base
- How to Compile Scala
- How to leverage EMACS for Guix packaging Helm mail (Mu4E) everything
- How to M-x magit shell
- How to reconstruct npm's logic from scratch (& generate npm_modules)
- How to talk about Tor to non-technical folks (and its political impacts)
- How to hack Ruby Gems for reproducibility
- How works RPM OSTREE (for R-B)?
- How to use Guix?

12:45 Lunch

13:45 Group picture

14:00 Strategic working sessions III (Session notes start on page 60)

- Reproducible file systems
- IPFS distribution
- User stories II
- Alpine
- Tools matrix II
- Bootstrapping III
- Trust models II

- Test infrastructure
- OPAM reproducibility

15:30 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.

16:00 Adjourn

16:05 Hack time (Notes start of page 77)

Day 3

Thursday, December 13

10:00 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.

10:20 Strategic working sessions IV (Session notes start on page 79)

- Bootstrapping IV
- Trust models III
- Cross distribution infrastructure
- MirageOS
- IPFS II
- Rebuilders III
- PGO Profile-guided optimization
- Generative sources
- Terminology issues II
- Debugging for reproducibility issues

- Getting openSUSE on testing infrastructure

12:30 Lunch

13:30 Strategic working sessions V (Session notes start on page 95)

- Strategic working sessions V continue (see notes on pages 79–94)
- Qubes OS reproducible hacking
- Transparency log for Debian and Tor Browser

15:30 Closing plenary

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

16:00 Adjourn

16:05 Hack time

Session notes

Day 1

Project updates

Sustainability of the Reproducible Builds project

Discussion focus: Governance, where the project is, sustainability.

Initially the project didn't need to be an organization. The Linux Foundation funded a few members of the core team for the first few years, CII provided support too.

Recently the Reproducible Builds became a member project of Software Freedom Conservancy: this helps to manage funding, and it helps image-wise since they are a good organization with good reputation. Also, they support providing useful connections and networking opportunities.

Reproducible openSUSE

Discussion focus: scripts to help make openSUSE builds reproducible

https://github.com/bmwiedemann/theunreproduciblepackage

The Unreproducible Package is meant as a practical way to demonstrate the various ways that software can break reproducible builds using just low level primitives without requiring external existing programs that implement these primitives themselves.

It is structured so that one subdirectory demonstrates one class of issues in some variants observed in the wild.

https://github.com/bmwiedemann/reproducibleopensuse

Reproducible openSUSE tools

https://github.com/bmwiedemann/reproducibleopensuse/blob/master/autoprovenance

autoprovenance

Toolsets

Discussion focus: Diffoscope, strip-nondeterminisms, tools for debugging.

https://reproducible-builds.org/tools/

Boostrap ability

Discussion focus: Bootstrap ability Ask me Anything (e.g. Is my favorite distro bootstapable?)

https://bootstrappable.org/

Connect through mailing list and/or the IRC channel #bootstrappable

https://gitlab.com/janneke/mes

Reduced Binary Seed Bootstrap http://joyofsource.com/reduced-binary-seed-bootstrap.html

Wish: every language developer to tell their bootstrap able success story.

How to achieve user trust and adoption?

Discussion focus: Once we have reproducible builds, how can we help users trust them and use them.

Project: How to protect Debian users from backdoors? Response: Reproducible builds can help with that!

So we need to have a system to ensure that all new packages are built reproducibly.

Also needed along the line: operated diversity.

Open questions, specifically about update processes: question of user privacy, trust between organizations (sharing infrastructure, agreeing on APIs etc.)

Debian APT archive. See https://www.debian.org/distrib/archive

Testing framework

Discussion focus: How we test distros for reproducibility

https://tests.reproducible-builds.org/

Current needs:

– Compare against what the distro are releasing.

– More rebuilders

Reproducibility by design

Discussion focus: Looking at distros which have elements of reproducibility by design

Agenda brainstorming

Specific projects/ languages

- Making Java reproducible
- rebuild OpenWrt
- What to do about Javascript?
- Discuss bootstrapping options for JVM languages
- Rebuild coreboot
- Make a plan for making Alpine r-b
- Get openSUSE on test.r-b-o
- Best practices for building (older) Go projects. For distro packaging

User

- Hoe users can verify binaries
- Debian is 94% reproducible in theory and 0% in practice. How to fix that?
- Conflict resolution/trust. How tdo we help people make good decisions when independent builds do not reproduce?
- How can we make it possible to know who and how many people have been reproducing a build

Specific problems

- Should we avoid archive for sources?
- How do relative buildpaths in binaries work with debuggers
- Making sure there is no rebuilder-software monopoly

Bootstrapping

- Port the Reduced Binary Seed bootstrap to other distros (from GNU Guix)
- What is/would the minimal binary needed for building a Linux From Scratch! (LFS)?
- Should open software be validated on open hardware as a baseline?
- Study how/if execution of binaries will impact the build. How do we trust HW? Openstandards for validation methods (?)
- Clarify the role of hardware in reproducibility and bootstrapping

Reach out

- How to reach out to compiler/language devs? (GCC, Rist, Go, Java,...)
- How to reach those who don't know about r-b or are not doing it?
- Let's have another r-b hackathon!
- Confusion about "reproducible builds", locked dependencies and reproducible binaries
- Develop methos to include HW design/ differences/ dependencies. Raise public awareness. Does it take an equivant(?) of Spectre/Meltdown before reproducibility is taken seriously?
- Help more people to use r-b tools

Funding/sustainability

- how to ensure sustainability for RB + bootstr. B
- How to reach-in; stay connected more between R-B Summits

Doc

- Imrprove documentation for r-b testing dashboard
- Find, refine, document core concepts: rebuilders, buildinfo
- Keep all the notes/etherpads?
- Create a document page "State of r-b in X project"
- Create a siplified sheet to see the progress of r-b in difference project-specificBest practices for reproducible Docker builds

Out of scope

- How do we learn from each other's independent projects?
- should we have more "vagrants" (devs of multiple distros)?

[Column with blank label]

- Universal measure of progress for build reproducibility
- How reproducible is enough to call something "reproducible"?
- Talk about what reproducible means. Levels of reproducibility

- Stardardized, comparable approach for testing uild reproducibility
- Achieve evaluation-time reproducibility in Nix
- How reproducible builds can help cross-distribution, cross-platform, cross-integration
- Figure out how to do deterministic encryption (required for reproducibility when storing secrets in the Nix store)
- How to snapshot a Fedora repo

[Column with blank label]

- Sharing, processing, comparing buildinfo files
- How to store build metadata and what should be in it?
- How to get from reproducible packages to a reproducible distro/project (distributing .buildinfo, rebuilders,...)
- What to do as a community about binary artefacts in source materials?
- How do you define the smallest trusted subset of packages in a distro?
- What's the rule of the build system in reproducibility?
- I want to get a mapping about the diffrence files and the programs which the diffrence caused.

[Column with blank label]

- verifying reproducible builds
- End-user tools – make sure to install packages reproduced by at least N trusted rebuilders
- How do we build a Web of Trust?
- How to have a trustworthy check of a Reproducible Building

[Column with blank label]

- How to use ds-build to reproduce rpm, deb,...?
- "buildinfo" format used by ds-build – is it different for different packages?
- How should rebuilders be operated

[Column with blank label]

- Hashing all the things

- Are IPFS-style hashs a solution to script-based program reproducibility?
- Getting to content-addressable builds (i.e. putting reproducibility to good use)
- Offline-first in packge managers
- Deprecating centralised mirrors (decentralising package repos)

[Column with blank label]

- Reproducible games :)
- "Port" reproducibility to all knowledge

[Post-its not belonging to any column]

- Handling signed artifacts (rpm,...) – strategy for distributing signatures
- Reproduce system images – make post-install scripts reproducible
- Status of reproducible Debian installs

User verification

- How does a user understand and trust an audit trail for an artifact or build process?
- User package reproducible verification (framework)
- How reproducible buildsmake your life easier?
- How can users help with reproducible builds?
- Develop tools for users to check they're using reproducible packages
- Expose reproducibility to users. How to know/install only reproducible packages
- How can users verify a build without having to rebuild it themselves? (efficiently)

Cross distro

- Distribution issue sharing of not reproducible packages
- Can we share built binaries across distros?
- What are most complicated non reproducible builds issues?
- Cross-distribution reproducbility
- When we have reproducible packages do we still need build time?

Language issues

- Can we continue to ignore NPM?
- We can reproduce OCamp packages
- Should projects randomize their dependencies?
- Define buildinfo for (Maven) Central repository + test IRL
- Add reproducibility reporting for java libraries + process to extend the tested perimeter
- I have a reproducible builds issue with Ruby Gem. Who could help me? S7I/We/jy/me/we
- OCaml specific diff tools
- Gcc optimization and instrumentation set specific to a processor
- Reproducible GHC

Bootstrap

- Bootstrap GHC (Glasgow Haskell Compiler)
- Bootstrap GHC & OCaml
- How are reproducibility and bootstrappability related?
- We need to talk about Java... [can the Java world be built from source alone?]
- Make bootstrapping best practice among compiler hackers
- Reach out to compiler writers to avoid bootstrapping problems
- Spread the word about bootstrapping

Developers Developers...Developers!

- How do we associate properties of source code (e.g. code review status) in a standard way to build artifacts?
- How to attach non reproducible metadatas to artifacts? ex: the build data as a Docker image label

Sources

- We have to reproduce generated sources too"
- How do we trust the source was legitimate?

Trust & Delivery

- What constitutes a proff of reproducibility?
- How do we distinguish or attest that an artifact was built on trusted hardware or in a trusted environment?

Things to do here

- learn about rep.b. tooling
- Have a Nix & Guix BoF/exchange/meeting
- Find out about common build system patterns

Build infrastructure

- Really simple repro-testing in CI-systems (like Nix (?))
- How to define a standardized matrix of variables to test if a package is reproducible
- Define reproducibility by a consensus of different build forms
- Can we guarantee reproducibility? (eg syndbox)
- Set up redundant build farms & package archives

Android/Java

- Verifying java/ Android libraries
- Find and hopefully fix Android SoK bug that makes everything unreproducible
- Bootstrap gradle
- Figure out how to do offline builds with gradle

[Post-its not belonging to any column]

- Discuss cross-compilation and build reproducibility
- How to improve cross-distro collaboration for reproducible build issues and patches?
- Security patching. How to avoid rebuilding the universe?
- Story to deal w/ specific hardware (optimizations)
- Discuss potential milestones for operating system reproducibility
- Are dynamic links ever advisable (will I always need to build statically linked everything?)

- Discuss build sandboxing properties
- Can we have reproducible development environments

Metadata standards

- Database/storage of reproducibility information/ .buildinfo archives
- Agree on standard for hash representation
- Could buildinfo be defined to be more easily bulk-comparable and indexable?

Communications

- Rigorizing a definition of the benefits of diverse compilation
- Promote reproducibility (verification) as part of the broader software supply chain

Rebuilders

- Talk about the current state of rebuilders (w/ NYU)
- (Think about) trust management forrebuilders
- Reproducing packages in the Debian arcive
- How to ensure rebuilder operator diversity and create trust in their claims
- locate/discover/enable more horsepower to work on reproducible builds
- Discuss what is needed for distributed package rebuilding
- Discuss how to involve third party people to rebuild packages
- Add openSUSE in test.reproducible-builds.org

End user/ Interfaces

- How can distributions/projects cooperate in running log servers?
- Verify reproducibility with package manager clients
- Make verification of reproducibility accessible toend users
- How should the user-facing part of reproducible builds look like and what should it contain
- How do I prove to others: This did/did not build reproducibly?
- How to represent reproducibility problems to OS users
- Discuss interfaces/user stories for "end users"

- How can I trust this content is correct? (Do we need CA's for packages?)

[Post-its not belonging to any column]

- Reproducibility on level of package managers (opown?)-success stories?
- What to do next with B_P_P_M build path prefix map
- 3rd party dependencies: Will I always need to pin everything or can e have deterministic resolvers?

Strategic working sessions I

Tooling

Tools in r-b:

diffoscope

strip-nondeterminism

disorderfs

reprotest

autoclassify (partially specific rpm/OBS)

autoprovenance

filterdiff? https://github.com/bmwiedemann/reproducibleopensuse/blob/master/filterdiff

buildinfo.debian.net

https://github.com/bmwiedemann/theunreproduciblepackage

obs-build for cross-distro buildenv setup

rebuilders ?

missing tools:

  trace back issues to source line

disorderfs:

  problems when applying it to whole build-chroot

diffoscope:

  running 30 minutes for an ISO image - e.g. diff sub-parts

  hard to see the interesting parts / changes in large diffs

tool interconnect / integration - similar to github+CI

r-b test aaS

- involve travis-CI

- repeatr?

packaging for tools
    matrix: what is packaged for which OS

Where to maintain/develop the tools - outside salsa?
    git repo, bug tracking, mailing lists, documentation, how to contribute
    go through the "good OSS project" checklist
    how to get more people involved
    - de-Debianize
    - add more use-cases - e.g. security for diffoscope
    - language-specific use-cases (Go, C++, Java)

doc: structure for different personas
    single dev
    doc: +cookbook do this to test that your software builds reproducibly
    distribution maintainer
    product/business manager wanting to learn about the benefits

rebuilders:
    how to incentivise running rebuilders
    what software is it? apply OSS checklist
    prepared public cloud rebuilder images

Nix / OBS:
    could quantify money or CO2 saved by build-tree-cutoff from r-b

Compilers

BPPM = BUILD_PATH_PREFIX_MAP

Specification defined in https://reproducible-builds.org/specs/build-path-prefix-map/.

Why do compilers embed buildpath? Do they need it?

Go compiler: stores _both_ buildpath and debug symbols.

Example package: seqtk (?) differs only in build path embedding.

Compiler option -fdebug-prefix-map.

GCC8 -ffile-prefix-map, used to manipulate __FILE__

Why BPPM needs to be a map (and not just "trim the part before the project root").

Clang has debug-prefix-map, but not file-prefix-map.

GCC rejected the BPPM patch: too many env ars, don't want any more.

Old idea: use this variable only if a special flag is provided? Modify debug-prefix-map or file-prefix-map.

On Nix/Guix (?) just use the same virtual path. Why not in Debian? Because we want full reproducibility.

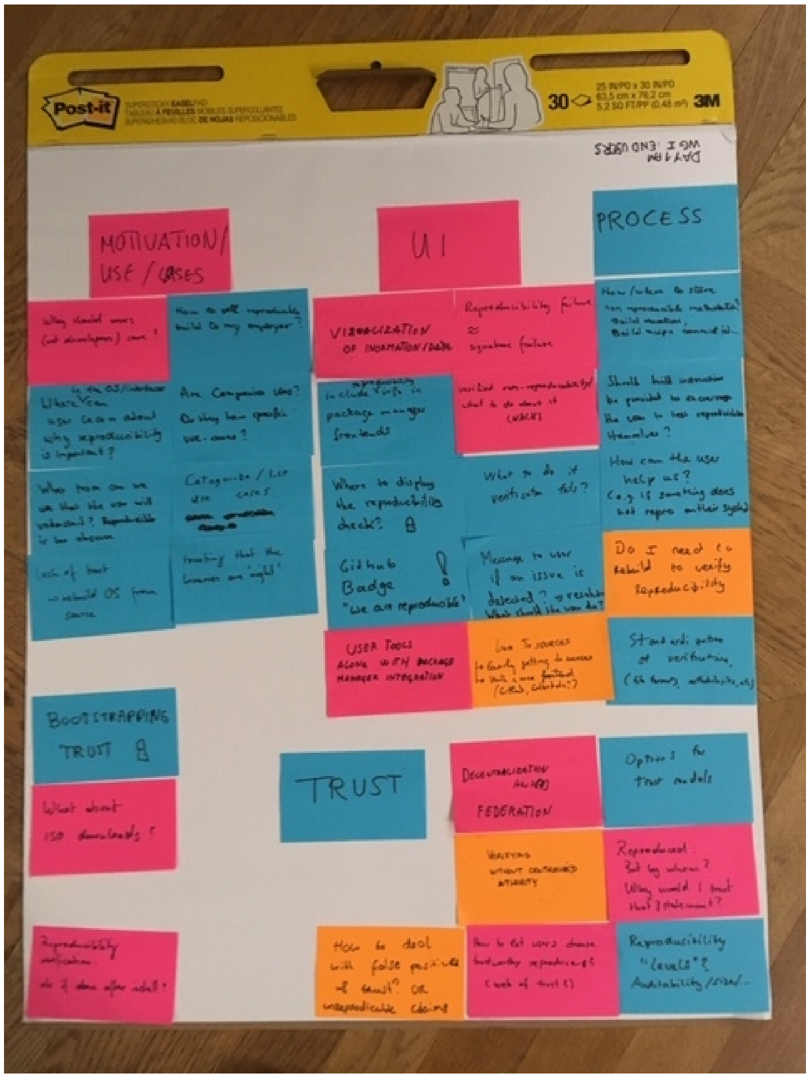
On Go: problem with static libraries.

Do we want to go with BPPM or not?

- Push for adopting it as-is?
- Work around by using build flags to accept this variable.

Do we want to rely on build flags instead?

User verification


Post-it
MOTIVATION/ USE/CASES
UI
PROCESS
VISUALIZATION OF INFORMATION/DATA
Github Badge "we are reproducible"
USER TOOLS
BOOTSTRAPPING TRUST
TRUST
DECENTRALIZATION AND FEDERATION
Options for trust models
Reproducibility "levels"?

# End user verification

## Motivation / Use cases

- why should users care?
- where can users learn why reproducible builds important?
- what term can we use that the user will understand? Reproducibility is perhaps a bit obscure
- how to sell reproducible builds to my employer?
- Are companies users? Do they have specific use-cases?

- Categorize / Use cases
- lack of trust => rebuild OS from source
- trusting that the binaries are 'right'

Tasks:
- Identify user stories

UI

- visualization of information/data
- include reproducibility in package manager frontends
- where to display the reproducibility check? (ssl key)
- Github badge? "are we reproducible"
    *
- user tools along with the package manager integration
- link to sources, easily getting to sources, with a nice frontend
- reproducibility failed == signature failed
    * is this possible?
    * verification before the repository
- verified non-reproducibility / what todo about it
- treat reproducibility failure like a signature failure?
- what to do if verification fails?
- Message to user if an issue is detected? Resolution, what should the user do?

Tasks:
(Before discussing UI, we should discuss the user stories)
- Github badge, how can we solve solve/make this happen? (For example go binary
  downloaded from Github or have someone verify the build from
  r-b.orgs and show the task color)

## Process

- how / where to store, non reproducible metadata (like build logs)? Build duration, build commit id
- should build instructions be provided to encourage the user to test reproducibility themselves
- how can the user help us? I.e. if something does not reproduce on their system
- do I need to rebuild to verify reproducibility?
- Standardization of verification (the formats, methodologies enabling automatic verification)

Tasks:

- how do I file a bug report and (automatically) provide useful information (diffoscope?)
- How do I verify a particular build? Before installing a package, in a standardized fashion.
- how to verify if a reproducible build is "trusted", how do this automatically? (From user perspective, how can he benefit)

# Bootstrapping trust

- what about ISO downloads?

Tasks:

- How do I verify if something I install is reproducible, e.g. a distro ISO

# Trust

- verifying without centralized authority
- reproducibility "levels": audit ability / size. How many reproducible build levels?
- decentralization and federation
- options for trust models
- It was reproduced, but by whom? Why would I trust that statement?
- How to let users choose trustworthy reproducers? (web of trust)
- How to deal with false positives & trust? Or unreproducible claims

- user expectation re reproducibility/verification: Ok if done after pkg install?

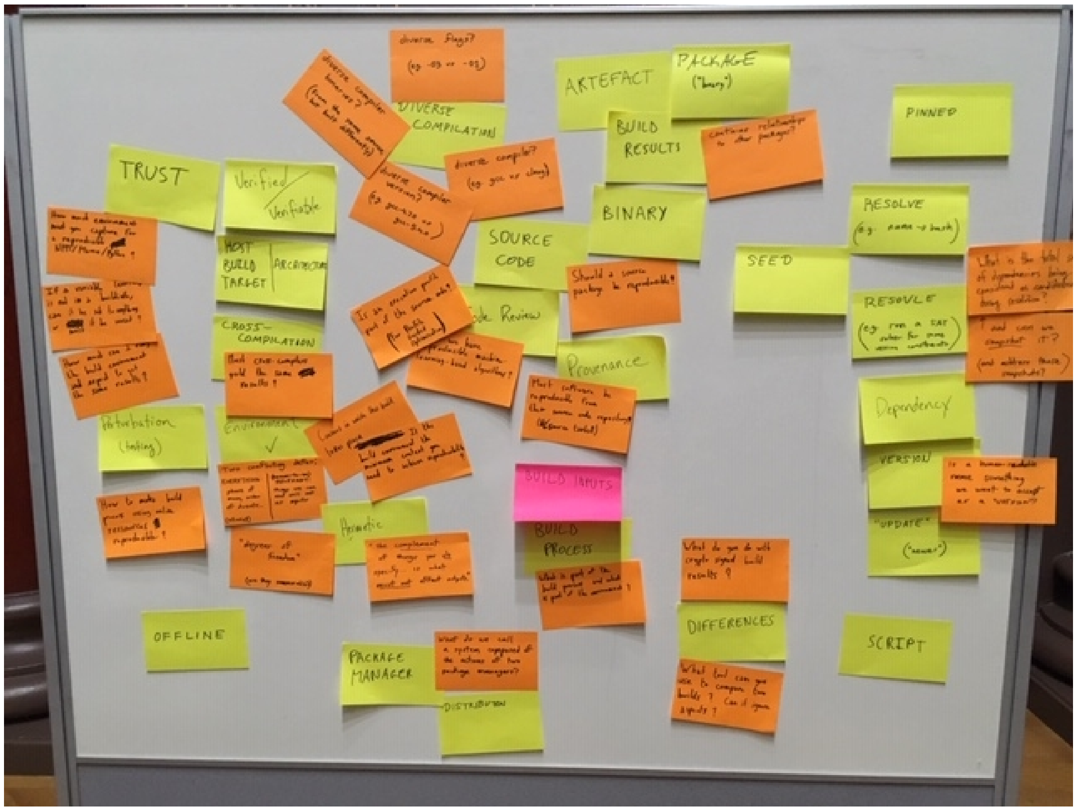
Tasks:

- Define different trust models and architectures?

TODO:

- Identify user stories
- Define different trust models and architectures?
- standardised way of verifying reproduciblity
- Identify user stories
- Github badge, how can we solve solve/make this happen? (For example go binary downloaded from Github or have someone verify the build from r-b.orgs and show the task color)

Terminology



# Vocabulary

Words that we might want to define or that felt relevant for our work:

* trust
* verified
* verifiable
* host/build/target architecture
* cross-compilation
* perturbation (related to testing)
* offline
* environment
* hermetic

* package manager
* distribution
* build inputs
* build process
* diverse compilation
* artefact
* package (the "binary" kind)
* build results
* binary
* source code
* code review
* provenance
* differences
* script
* pinned
* resolve (e.g. name → hash)
* resolve (e.g. run a SAT solver for some version constraints)
* seed
* dependency
* version
* update ("newer")

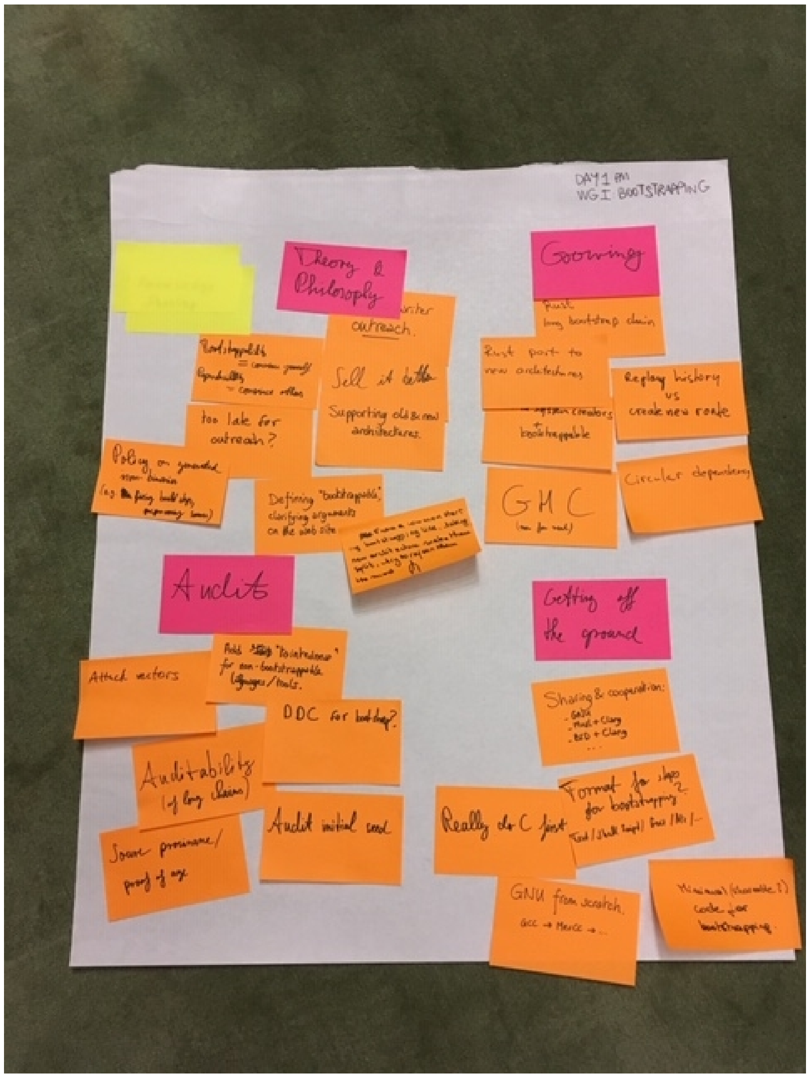
Questions that arised from looking at these terms:

* How much environemnt must you capture for a reproducible NPM/Maven/Python (which are multiplatforms so the issues might be different than with distributions)?
* If an environment variable is not in a .buildinfo, can it be set to anything or must it be unset?
* How much can I vary the build environment and exepect to get the same results?
* Must cross-copmilers yield the same results than same architecture compilers?
* How to make build process using online ressources reproducible?

* Environment has two conflicting definitions: everything (which might include phase of the moon, order of directory entries) vs. (known-to-be) revelant (things we can and wil be set as inputs)
* "degrees of freedom" (are they enummerable?)
* "the _complement_ of things you do specificy… is what _must not_ afect outputs."
* What do we call a system composed of the actions of two package managers?
* What is part of the build process and what is part of the environment?
* diverse flags (e.g. -O3 vs. -O1)
* diverse compiler binaries? (from the same source, but build differently)
* diverse compier (e.g. gcc vs clang)
* Is an execution profile part of the source code? (for Profile Guided Optimization)
* Could we have reproducible machine-learning-based algorithms?
* Must software be reproducible from their source code repository (and/or source tarballls)?
* Does build results contain relationships to other packages?
* WHat is the total set of dependencies being cosidered as candidates during resolution?
* … and can we snapshot it? (and address those snapshots?)
* Is a human-readable name something we want to accept as a "version"?

/!\ There should also be a photo of the post-its board somewhere /!\ (see above)

Bootstrapping



All things bootstrapping (<2018-12-11 Tue 15:15>)

- what should be done when the bootstrap chain becomes too long? (example: Rust)

- porting a bootstrap chain from one architecture to another (example: Rust)

- compile writer outreach: ask compiler writers to do the right thing, so that we have less work. We need to find a way to convince compiler writers, to sell the idea.

- build system creators. Example: maven depends on plugins whose dependencies later switched to maven. Not really the build system authors' fault. Is it already too late for outreach?

- standardize "GNU from scratch": GCC -> MesCC -> …

- sharing a minimal base system & cross-system cooperation:
    - GNU
    - Musl + Clang
    - BSD + Clang

- identify alternative bootstrap paths that allow us to share previous bootstrapping work

- implement diverse double compilation (DDC) for the early bootstrap (for verification)

- is a source bootstrap with long dependency chains really better than trusting a single binary? Probably, because having the source code could enable audits, which would be difficult to do with just a binary blob.

- clarify arguments for bootstrapping on the website (https://bootstrappable.org)

- define bootstrappability

- list attack vectors that can be prevented with bootstrapping

- what ways do we have to proof that source archives are "genuinely old"? Can we proof/verify source provenance?

- supporting old and new architectures: do we cross-compile from old architectures to the new ones? Or do we port the bootstrap to the new architecture?

- *how* do we share the bootstrap; in what format? Should it be an executable shell script? A Nix derivation? A Guix Guile builder script…? Should it be “human-executable” text instructions?

- how do we deal with dependency cycles? Either by unrolling dependency loops to chains (i.e. replaying history), or by creating a new route (e.g. by writing a new minimal implementation barely satisfying the dependency)

- what is a reasonable policy for generated stuff (that are not binaries)? Example: generated build instructions to get rid of a circular dependency on a build system; or generated C “source code”. What can we accept and what needs to be replaced?

- audit the very early C bootstrap: have we really thought of everything wrt Stage0?

- does the early bootstrap really have to reach C first? Is there a better first “real” language?

POSIX systems need to have an early C bootstrap. On top of that we have other languages that each need their own bootstrap story that may assume the presence of a POSIX system.

Outreach: should we add a “tainted” metric to packages that directly or indirectly depend on binary blobs? Produce a ranking of languages and their “ecosystems” wrt to the amount of “taintedness” or the cumulative size of the binary seed?

In some aspects bootstrappability is orthogonal to reproducibility, but on some axes there’s a clear connection. Reproducibility is important for auditing purposes, but if there’s a dependency on an opaque binary blob, the audit cannot be complete.

Bootstrapping allows you to convince yourself of the trustworthiness of a binary. Reproducibility allows you to transfer trust to others.

Trascription of post-it notes

Theory and philosophy

- Compiler writer outreach
- Bootstrappability=convince yourself. Reproducibility=convince others
- Too late for outreach?
- Sell it better
- Supporting old & new architectures
- Defining "bootstrappable" clarifying arguments on the website
- Policy on generated non-binaries (e.g. fixing build steps, prep+using source)
- From a common starting bootstrapping line, taking new architecture make them split, try to rejoin them the most

Growing

- Rust long boostrap chain
- Rust port tonew architecture
- Replay history vs create new route
- build system creators + bootstrappable
- Circular dependency
- GHC (now for real)

Audit

- attack vectors
- Add "taintedness" fornon-bootstrappable languages/tools
- DCC for bootstrap?
- Auditability (of long chains)
- Source proviname/proof of age
- Audit initial seed

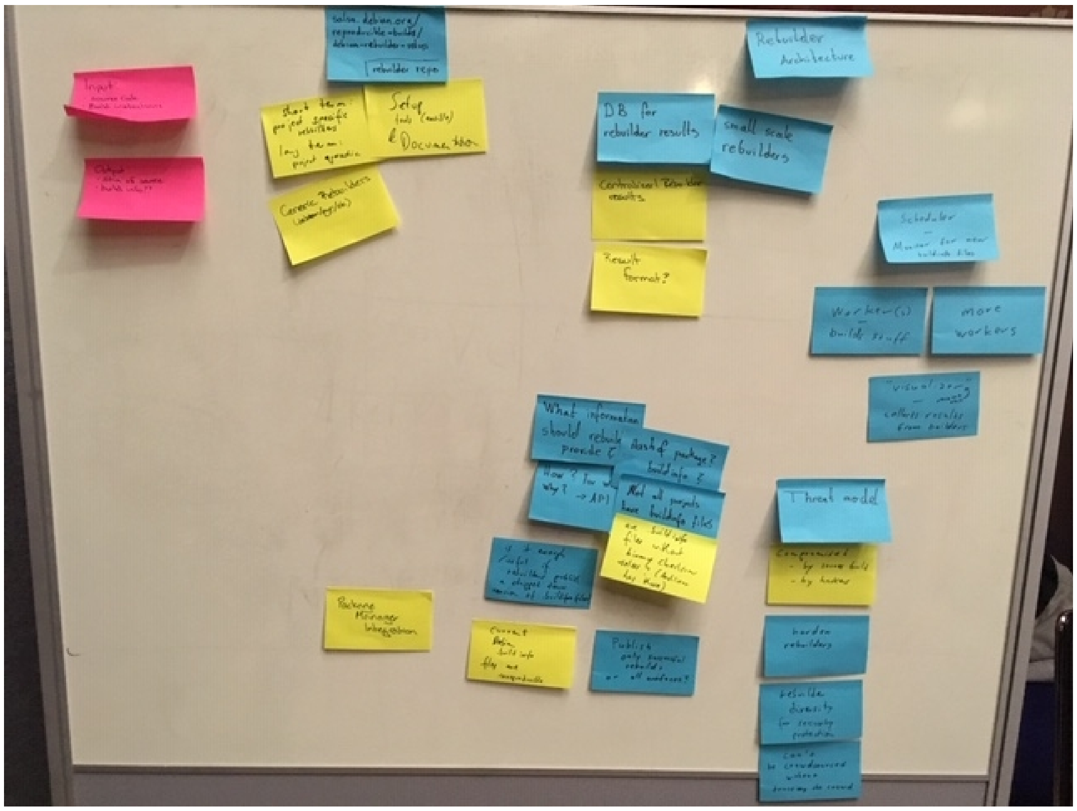
Getting off the ground

- Sharing & cooperation: GNU; Musl + Clang; BSD + Clang; ...
- Format for steps for bootstrapping? Text/ shell scripd/ Guix/ Nix/...
- Really do C first

GNU from scratch. GCC->MesCC->...

Minimal (sharable?) code for bootstrapping

Rebuilders



Input: source code; Build instructions

Output: sha of source; build info??

Rebuilder repo: salsa.debian.org/reproducible-builds/debian-rebuilder-setup

Short term: project specific rebuilders. long term:project agnostic

Generic rebuilders (debian/etc)

Setup tools (available) & documentation

Rebuilder architecture

DB for rebuilder results

Small scale rebuilders

Centralized rebuilder results

result format?

Scheduler – monitor for new buildinfo files

Worker(s) – buildstuff

More workers

"visualizer" (-> rename this) collects results from builders

What information should rebuilders provide?

How? From whom? Why? -> API

Hash of package? buildinfo?

Not all projects have buildinfo files

are .buildinfo files without inary checksums useless? (Archlinux has those)

is it enough / useful if rebuilders publish a stripped down version of .buildinfo files?

Current Debian .buildinfo files are unreproducible

Publish only successful rebuilds or all artifacts?

Package manager integration

Theat model

Compromised: by sources build; by hackers

Harden rebuilders

Rebuilder diversify for security protection

Can't be crowdsourced without trusting the crowd

Mapping out hardware level issues

There are more and more architectures (x86, ARM, GPUs, FPGA, machine learning chips, DSPs, ...) with interesting aspects that cause irreproducible builds:

* Instruction set differences (e.g. does the build machine have SSE4?)

* CPU microcode may change instruction semantics

* Drivers not open source, black box (e.g. CUDA drivers)

* FPGA (no knowledge of placement algorithm etc.)

* Random build failures because of too much memory, too many cores.

* ARM memory ordering giving different results (ARM is less forgiving than x86)

* Hardware failures (random bitflips, cosmic rays, ...)

* HW has changed, current top-end micro-processors have > 20 threads/socket. A two socket system will have 40 threads.
  Observations are that a fair amount of number of bulilds fail with high thread counts, main reason are incorrect Makefiles. The current work around is to limit the number of threads to 1 - which yields in a max 2% utiliztion in the above example. In many cases builds are somewhat timing critical (to release fixes etc) so having an efficient and scalable parallel build environment is important

* Current technologies are using strace to trace exact execution of builds. This probably will not work on multi-node builds as clocks are not running lockstep (do we need a distributed strace facility moving forward?)

Also, how can we trust the hardware? Somebody mentioned backdoors in FPGAs. (Could there be a Thompson attack on hardware? E.g. could the software used to build the hardware (VHDL compiler) be backdoored to insert/propagate a backdoor in the hardware?)

Solutions:

* Develop best practices for dealing with various irreproducibility issues. For example, for races in builds, switch to build systems that provide proper isolation between build steps so there are no undeclared dependencies. For those stuck on older build system, can we provide tools that detect concurrency issues preemptively? (E.g. use strace to discover undeclared dependencies in Makefiles.)

* Split software into deterministic and non-deterministic part?

* Balance performance and reproducibility. For example, profile-guided optimization is hard to reconcile with reproducibility, since profiles are typically not reproducible, so you may need to turn it off.

* For the trust issue: use open hardware to build software. Can then trust that the hardware is doing what it's supposed to be doing. Maybe there should be a reference platform (say, based on RISC-V).

* Disable hardware features (e.g. can we make features like SSE unavailable to build processes?)

Random shoutout to MIPS and Cray-1 instruction sets.

Java reproducibility

objective: verify that dependencies are corresponding to source code

not only Java (jars) but also Android (apk)

source code as tarball, run build

but build is "transparent"/"opaque" => don't know all the inputs: dependencies are pulled from (Maven) Central repository or others as binary

how to check everything that was pulled?

Bootstrapping issues in addition

Maven is now available in Gix as bootstrap, but not plugins

Gradle in progress, but more complex since even more dependencies

Java ecosystem uses Maven Central as repository

how to rebuild Central repository content in a separate location to check against original?

sources required in central are not for build, but for IDE users = -sources.jar for .jar

see https://central.sonatype.org/pages/requirements.html#supply-javadoc-and-sources for reference

but only sometimes -source-release.zip or .tar.gz (= source tarballs that we need to rebuild) are provided, since not defined as checked requirement to publish

Android is globally using Gradle and download from many repositories, mainly "Google's Maven Repository" (https://dl.google.com/dl/android/maven2/index.html)

some code in there is proprietary

reproducibility requires to store which JDK version is used, since JDK 7, 8, 9 ... don't generate the same bytecode

=> absolutely need to record JDK version used to be able to later rebuild by someone else

recording platform is questionable

how can we move forward and help future artifacts published to Central to be really reproducible and rebuilt?

record:

- jdk version used for the reference build
- build tool (with version) used for the reference build
- url of source tarball
- exact transitive dependencies used (at least for version ranges)
- eventually a flag to explain that the build is known not yet reproducible
- OS?

=> need to talk about publishing the build recipe (as Buildinfo...)

then how to verify dependencies at buildtime, because they might come from other repository, or changed in the middle

exists Gradle witness plugin that records in Gradle buildfile hashes of dependencies used, and when hash is already present checks that the calculated hash is the same

but does not the work for plugins,

and some people complain that build is slower

Day 2

Strategic working sessions II

User stories

End User Stories (Dec. 12, 2018)

Companies or Enterprise
Upstream Package maintainer (Author)
Distribution package maintainer
System Administrator
End User
Non-distro user
Provenance tracking
Reproducible Builds Project

* End users

** Distro and package manager users

- User just wants “good” software artifacts/packages.
- Providing a globally consistent view on what is the “correct” binary for some source code can help ensure we have “good” binaries.

- Users want to make sure the binary they have is the same everyone else is running.
- User installing a package through a distro convinces themselves that a package is trustworthy:
  + they can “challenge” package providers (“archives”) by comparing the contents of the packages they serve;
  + they can click a button (e.g., in GNOME Software) or run a command to rebuild locally and compare the contents, etc.
  + they can set a policy to install a package only if it was rebuild by K out of N trusted builders;
  + they can choose between different package providers for their distro;
  + they can run a package provider for their distro;
  + they can build everything.
- Users push information back to the community by rebuilding and reporting issues to the distro and developers.
- Distros could push reproducibility beyond packages to whole systems.

** Users installing non-distro software off the Internet

- Many users don’t use package managers much; instead they use, for instance, images from DockerHub or statically-linked Go binaries posted on GitHub. How do we address the needs of these non-distro users?
  + The Go binary was built by Travis, is hosted on GitHub, etc., which creates a long chain of entities that users of the binaries must trust. To address this, upstream projects that provide Go binaries could subscribe to reproducible-builds.org, which would rebuild the software through a Dockerfile; when the build matches, they would get a green binary reproducibility badge to show on GitHub.
- Reproducible builds make diverse double compilation (DDC) a feasible

thing; this allows security auditors to look for backdoors in binaries.

** Company employees installing internal software

- For internal company software, being able to check the source/binary correspondance helps debugging (“reproducibility of bugs”) and ensures you’re running the right software.
- Reproducible builds can help make license auditing because you can trace an artifact back to source. Put another way, reproducible builds provide good provenance tracking.

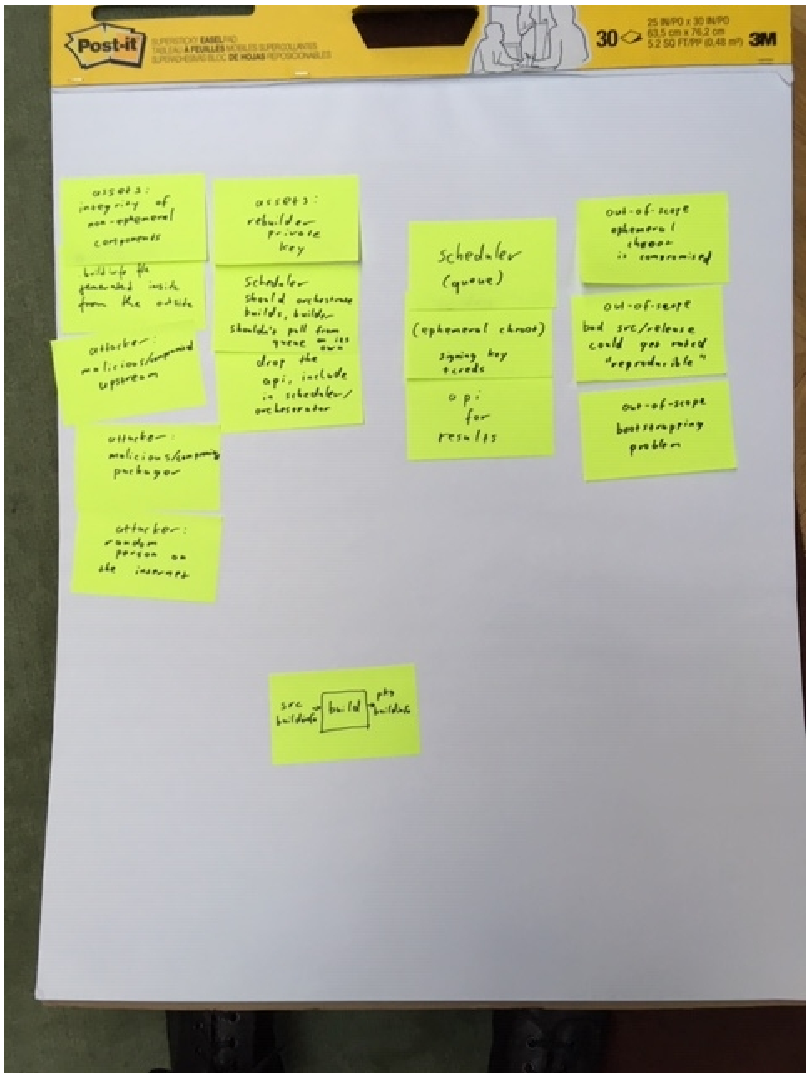
* Upstream software maintainers

- User has a bug and wants to reproduce the misleading artifact for debugging.
- Software developers that distribute binaries (usually distro developers, but also upstream developers that provide Flatpaks, etc.) want to make sure their software has no backdoor, and want to make sure they are not themselves an attack vector.

* Distribution package maintainers

- How do distro developers learn about reproducible builds? How do they learn about best practices and tools? For example, Nix and Guix enforce policies such as no network access in the build environment and provide tools such as “build a package several times and compare.”
- For distro maintainer, R-B make enable verification of the binaries they provide so users can build “trust” in the software binaries they obtain.

Rebuilder security


assets: integrity of non-ephemeral components
buildinfo file generated inside from the outside
attacker: malicious/compromised upstream
attacker: malicious/compromised packager
attacker: random person on the internet
assets: rebuilder private key
Scheduler should orchestrate builds, builder shouldn't pull from queue on its own
drop the api, include in scheduler/orchestrator
Scheduler (queue)
(ephemeral chroot)
signing key + creds
api for results
out-of-scope ephemeral chroot is compromised
out-of-scope bad src/release could get rated "reproducible"
out-of-scope bootstrapping problem
src buildinfo
build
pkg buildinfo

scheduler

- schould have authority over the build queue - builders (the thing inside chroot) should not have direct access to it

builders (use fresh chroot)

API for results

- should be part of scheduler, not separate thing buildinfo can be uploaded without any permission - increased attack surface

assets:

- rebuilder private key
- integrity of non-ephemeral components (everything outside of chroot)

attackers:

- malicious/compromised upstream -> out of scope
- malicious/compromised packager -> with reproducible builds we care about malicious binary package
- random person submiting buildinfo file

out of scope:

- ephemeral chroot compromised
- bad src/release could get rated "reproducible"

- bootstraping problem

Debian generate buildinfo inside (potentially compromised) chroot

todo:

- verify the buildinfo file generated inside, from the outside
- use something stricter than chroot for sandboxing the build

compromised binary package could compromise further builds on all rebuilders

rebuilders should use only packages confirmed to reproduce

maybe: have "unstable-confirmed" with only confirmed packages for (re)builders to use

- (re)builders should verify this before installing (with buildinfo files)

Bootstrapping II

Summary of state of bootstrapping in Nix/Guix.

Classic bootstrap tools set in Nix/Guix: gcc, binutils, bash, coreutils, ... (250+ MB of opaque binaries)

Current state in Guix: replaces gcc, glibc, binutils (roughly 150 MB of binaries) by linux-libre kernel headers, mescc-tools, bootstrap-mes (https://www.gnu.org/software/mes/). mescc-tools provides among other things a Scheme interpreter that is used to build a patched tinycc, which in turn builds gcc. mes also contains a small C library. The Scheme interpreter is currently delivered in binary form, so the next step is to bootstrap *that* from source.

Should be fairly straightforward to use this in the Nixpkgs bootstrap. However, currently no ARM support.

Have to use old gcc (pre 4.something) because newer gcc requires a C++ compiler.

in guix just now:

mes (scheme interpreter / C compiler) -> tinycc -> ancient binutils (2.2.0.18) -> ancient gcc-core (2.95) that doesn't depend on (most of) glibc -> ancient glibc (2.2.5) -> full ancient gcc -> gcc 4.7.4 (last pre-C++ gcc) [all of the previous steps are i686-linux] -> crossbuild of current version of gcc for x86_64-linux

Kernel dependency? Bootstrapping on bare hardware without an OS?

Overhead of the new bootstrap approach: 3 hours or so?

Mes bootstrap tarballs: http://alpha.gnu.org/gnu/guix/bootstrap/i686-linux/20181020/

Java and JVM languages

JMV languages bootstrap

1. situation on guix:

    1.1 bootstrapping

    1.2 JVM: bootstrapped

    1.3 jdk: version 11

    1.4 JAVA: bootstrapped

      1.4.1 ant: bootstrapped

        1.4.1.1 have build system

    1.5 clojure: bootstrapped

      1.5.1 have build system

      1.5.2 missing leiningen

*** unbundle asm and lang

** groovy: bootstrapped

** scala: compiler bootstrapping problem

* tools

** maven bootstrapped

* scala compiler

** keep dependency count low

*** antlr

*** bcel

*** ant

** grammar for scala

*** ast done

** empty classes build

** inhertance all done

** ignore type checking -- as far as we can

** target something recent

** write it in compcert style

** keep it uptodate

*** involve other people when done

* scala no other dependencies than itself

** twitter might be interested

* gradle

* maven

** too much dependencies

* kotlin

** depends on itself

** fials to build the dependency list

* android sdk?

** google's repo, combine all repos, build in one round

* check mockito on guix

Hardware II

- [mfl] Increasing number of cores, from single or dual core system to 100s or thousands of cores in a SSI (Single System Image, i.e. instance of a Linux kernel)

- Spreading out builds to multiple nodes can expose new problems. These problems probably expose today, but given the limited amount of parallel execution the race conditions may not be visible.

- big cluster systems might have new big problems

- wrong calculations, hardhare have bugs. r-b can find serious bugs

[mfl] I think incorrect results, or in other words undetected user level data corruption are extremely low probability. All CPU vendors are putting super efforts into validation as wrong results are an absolute no-go. The cache and memory sub-systems are probably most exposed to random bit-flips, but are protected through SECDED (Single Bit Error Correction Double Bit Error Detection). In addition the majority of external memories has a chip-kill feature, i.e. the ECC bits are laid out in way accross the multiple memory chips that an entire chip can fail.

RAS (Reliability Availability and Serviceability) have become common in HW and SW design,there are different techniques in predicting errors and changing corresponding components.

It should however be clear that HW is failing due to a multitude of errors and these errors are expected and handled by design. The same is true for disk drives which have a number of reserved sectors as spares, replacement is (almost) transparant to the user.

Communication interfaces (copper or fibre) are using advanced algorithms to eliminate errors, again this is totally expected.

More relevant are "new" bugs as demonstrated by SPECTRE and MELTDOWN.

But then remember the old saying "Any piece of HW can fail at any time" :)

- cpus are blackboxes, based on software (microcode)

[mfl] Certain instructions are micro-coded. Micro-code is SW (although a bit special) and the implication is that a build may yield different behaviour. At least there is no way to integrate micro-code into hashes. The sme is true for other "invisible" SW components as the BIOS and BMC

- The hardware in 10 years might be difference in the more detailed behaviour. E.g. the timing might be different, but still in the spec. But programs or build process require such behaviour

[mfl] The point here is that certain HW elemenents can have a life-cycle of decades. We need to pro-actively reserach potential pitfalls

- fpga: are not reproducible because of place and route, which usually not reproducible.

[mfl[to some extend unpredictable behaviour also occurs on HW caches. It probably can be safe to assume that a FPGA implementation is operating when the VHDL/Verilog or whatever HDL has synthesized and placed&routed succesful. Any error there should probably be seen similar to a compiler error generating wrong code.

Need to evaluate if it is feasable to create hashes for bitstreams.

- r-b become reproducible calculations. because some hardware does not guarantee reproducible calculations.

[I'm lost here which HW doesn't guarantee reproduceable results? There are a lot of customers actually validating bit-wise correct results on many different platforms.

- everybody loves bitflips

[mfl again I don't see bitflips as soemthing which will show up at the user level. If there is other proof on this we obviosly need to re-consider

- it probably would be helpful to include low level HW details into build information, including except CPU architecture and revision, specific uCode information, BIOS and BMC level as well as other low-level information. I would even go so far to consider to include environmental conditions as modern CPUs have very advanced power management feature to control TDP, so a warm-running CPU will behave differently than one running at constant temperature. One key feature is brown or black silicon where parts of the chip are agressively powered off to optimize. Same for other features such as Turbo Mode etc. This has been a major problem for High Performance Benchmarks as timings can greatly vary across runs without obvious reasons. If timing is monitored for r-b having a sudden increase of 10% or 20% of wallclock time could become a problem

Cross distribution participation

representatives from Archlinux, openSUSE, Guix, Nix, Alpine Linux, Debian

"package finder tool" interface for searching infomation about packages, links to distribution trackers, etc.:

https://maintainer.zq1.de

sharing between distros:

https://salsa.debian.org/reproducible-builds/reproducible-notes/blob/master/ideas_on_sharing_notes_between_distros

- export distro-specific information via json and incorportate into tests.reproducible-builds.org
- add more distros to tests.reproducible-builds.org database

what is needed to create a database across distros to compare reproducibility across distros:

-

Source reproducibility, upstream may replace tarballs with minor fixes without changing the names.

- pressure upstream as a cross-distro effort to upstreams to always
- github tarballs may not always produce same output
- Use software heritage archive api to backup/mirror
  https://www.softwareheritage.org/archive/
- Document best practices for reproducible builds
  Debian documentation: https://wiki.debian.org/UpstreamGuide

repology as source of information about packages in various distros: https://repology.org/

PostgreSQL database dump of the reproducible status of Debian, Arch Linux and ?

https://tests.reproducible-builds.org/reproducible.sql.xz (MB)

Schema https://tests.reproducible-builds.org/reproducibledb.html

- debian-specific assumptions in schema (e.g. suite)
- adding a distriubtion field to the reproducible build database (mattia)

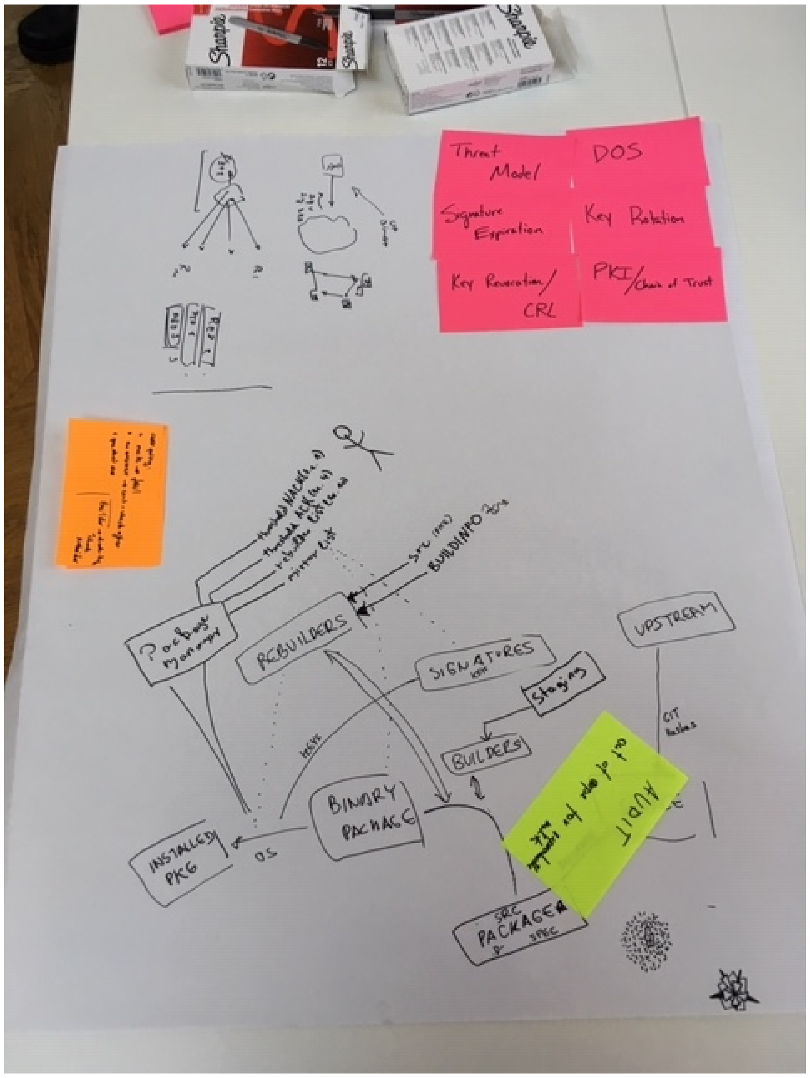
issues around go packaging:

- downloads dependencies at build time, fixed in some distros by downloading dependencies and making available via normal distro channels

database of patches submitted upstream to avoid duplicate work

- git grep on reproducible blog
- reproducible builds issue tracking system for patches?

Trust models



# trust model for reproducible builds and rebuilders

- upstream (the developers who did a release) release a package (tarball)
- needs an audit that there are no malicious things in there
- the packager captures the checksum of the tarball, writes a spec file, develops patches for the release -> sends to the builder(s)
- the builder (automated infrastructure) builds the package (resulting in a binary package), and signs it

- the user installs a package (via package manager)

the trust is in the builder (which holds the key)

now, adding rebuilders to the picture, who rebuild the package, and should be checked

want to remove the blind trust in the authority!

requirements:
- negative results?
  - if one can't be fetched, it's a failure
- DoS?

solution:
TUF/notary
- two trees, one for each rebuilder

adjustments to package manager
- needs to know a rebuilder list

process:
- user gets binary package
- wants to verify binary package
- uses a magic box for that: signature of "upstream" + signatures of rebuilders
- magic box aggregates rebuilder results
- box could be package manager, could be around each rebuilder
- who defines the treshold

verification:
- either client has to query a list of rebuilder, and check checksums
- or rebuilders are federated and client does a single request

- not totally trivial to solve, DoS, evil rebuilders
- how to choose which rebuilder checksums to use
- certificate transparency log

if one of the signing keys gets compromised, with a CT you can see whether some pakcage was signed out of the official loop

for the n+1 threshold (n should be chosen by the user), the negative matters as well

whenever you query the rebuilder: three-state:

- i rebuilt it, here's the hash \
- i rebuilt it, but failed hash / there's only a single result, the rebuilder does not need to know the hash of te builder
- DoS/unreachable

rebuilder signs with their key the signature of the package

client retrieves a list of checksums, and verifies that it meets

distribution of hashes vs trust model

"score system" for rebuilders?

- some are more trusted than others
- context-dependent
- from the POV of the client, want to mark some rebuilders as non-trusted

policy vs long-term audit

"official" mirror list + "official" rebuilder list

reputation is part of the model

malicious rebuilders could get official checksums, and publish it to get high reputation

another model: builder only publishes package when some rebuilders have successfully rebuild
- race condition: builder published, rebuilders have not yet gotten the new package
- staging: only publish result if you manage to reproduce package several times
 - ideally use the same model and tools
 - if gets hacked, rebuilders may find out

- rebuilder depends on results of builder (build-info file)

what should the user configure?
- rebuilder list (gets a default one), add/remove some
- threshold of acknowledged, non-acknowledget
- non-acknowledget
 - signature verification failure

non-nacks:
- communication failure?
- not yet rebuild?

- single non-reproducibility should lead to failure
 - may be a hardware failure, issue in rebuilder

timing and monotonicity:
- package is released at point X
- rebuild by threshold X+1
- can install now!
- another rebuilder nacks at X+2
- cannot install anymore
- what happens in (a) already installed on machine (b) not installed

(a) alert, have to audit

(b) don't install

what are the types for request/response for rebuilder?

- single signature or set of signatures?
- request: combined for multiple packages or fetch the entire database

for a given package name and version, what should be the result?

- a one-to-one connection between package+version to single hash+signature
- now if it rebuilds multiple times with varying results, what should be the answer?

TUF (the update framework)

- captures snapshot of all the packages at a given time
- provides this signed

what we're getting here

- apt using tuf, gets snapshot
- apt install a requires b {<1.2}
- concern: treshold for 1.1 is not there, but 1.0 is there
- first resolve all the packages, then install something

Tools matrix

Package Matrix for https://reproducible-builds.org/tools/

- Repository management tools
    - http://maintainer.zq1.de/ gives you a nice overview of what is packaged in which version on which (major) distro, and also links to various resources for that package (e.g. repology)
    - https://repology.org/ is the prime source for all kinds of data on versions, packages and metapackages.
- We started creating a Matrix to see which of the tools are packaged
    - First on paper (http://profpatsch.de/stuff/img/reproducible-tools-matrix.jpg), boxes are unpackaged
    - The next step would be to query the repology API and generate the matrix automatically, e.g. https://repology.org/api/v1/metapackage/diffoscope gives you all necessary information about diffoscope on various distros.
    - Could write a snippet of JS that queries repology (asynchronously) on the bottom of https://reproducible-builds.org/tools/
        - Update: there's badges provided by repology! But you either get a very long list of (old) repositories or a suboptimal link for each package and version. See https://github.com/repology/repology/issues/762
- nixpkgs
    - P packaged/updated strip-nondeterminism https://github.com/NixOS/nixpkgs/pull/51905
    - JB (nomeata) packaged reprotest for nixpkgs, making it compatible with non-debian systems. TODO: links to PR(s)
- Alpine
    - N packaged diffoscope & found a bug in Alpine's libmagic when fixing a failing test.
- Arch
    - disorderfs was updated to the latest version & a test fixed (https://vdwaa.nl/arch-reproducible-build-summit-18.html)

Strategic working sessions III

Reproducible file systems

# Use cases

System images include e.g. cloud images, live media, installation media.

# Solved already

- GPT: rather easy, see upcoming technical report from Tails.
- iso9660, SquashFS: fixed last year or two, see previous technical
  report from Tails.

⇒ we'll focus on non-solved.

# FAT filesystem

## Overview

2 ways to create:

- mkfs
- mtools' mformat

2 ways to copy:

- mount + cp
- mtools' mmd + mcopy

Until today, Qubes OS and Tails did not manage to get 100%

reproducibility of FAT filesystems.

## WIP

- Qubes OS: https://github.com/QubesOS/qubes-installer-qubes-os/pull/26
- Tails: https://redmine.tails.boum.org/code/issues/15985
- Debian Installer: https://salsa.debian.org/installer-team/debian-installer/merge_requests/3
  (Thanks!)

## Creating filesystem

Solved!

As of 2018-12-12, on Debian sid, this generates 2 identical FAT32 filesystems:

```
#!/bin/sh

set -eu

SRC_DIR="$1"

[ -d "$SRC_DIR" ] || exit 2

export MTOOLS_SKIP_CHECK=1

TIMESTAMP='2008-12-24 08:15:42'

for i in 1 2 ; do
   IMG=$(readlink -f $i.img)
   rm -f $i.img
   mkfs.msdos --invariant -v -i 1234ACAB -C $i.img $((1300 * 1024))
```

```
  (
    cd "$SRC_DIR"
    find ! -name . -type d -print0 | LC_ALL=C sort -z | \
       xargs -0 -n1 -I_ faketime "$TIMESTAMP" mmd -i $IMG ::_
    find -type f -print0 | LC_ALL=C sort -z | \
       xargs -0 -n1 -I_ faketime "$TIMESTAMP" mcopy -i $IMG _ ::_
  )
  [ $i -eq 2 ] || sleep 61
done
```

IPFS distribution

Using IPFS to distribute package binaries (Dec. 12th)

* How does IPFS work?

- A “peer” in the IPFS network is a machine running ‘ipfs daemon’.
  Each peer in the IPFS network has a “peer ID” (the hash of a public
  key) and connects to other peers. Together, the peer form a
  distributed file system. The file system can be mounted using FUSE.
  Content is unencrypted; applications who need it are expected to do
  it by themselves.
- IPFS uses a distributed hash table (DHT) as a value store; libp2p
  implements DHT communication. Users look up content by hash in the
  DHT, which gives them the contact information (IP address, etc.) of
  the peers that provide that content.
- Users can “pin” data to prevent it from being garbage-collected.
  Non-pinned data—e.g., data a peer fetch from other peers—can be
  garbage-collected.
- Data inserted in IPFS is chunked and stored as a Merkle tree. All
  the data is content-addressed.
- When the user adds a file with “ipfs add file.txt”, IPFS performs
  the following procedure:
  1. Split the file in several chunks, by default 256 KiB.
  2. The set of chunks leads to a Merkle tree.
  3. The end result is the SHA256 of the root of the tree.
- For some cases—e.g., video streaming—an “unbalanced” Merkle tree (a
  “trickle”) that is more appropriate for retrieving data
  incrementally.
- IPFS can chunk using a content-based chunking algorithm to detect
  identical chunks within files.

- IPFS provides an HTTP gateway: ‘ipfs daemon’ serves requests such as localhost:8080/ipfs/HASH/file.txt. https://ipfs.io provides that gateway, although it’s obviously centralized and mostly useful for testing.
- libp2p implements discovery over mDNS/DNS-SD, NAT traversal, Quick support, etc. etc.

* Sending packages on IPFS

- Instead of https://mirror.debian.org/….deb, you would have http://localhost:8080:/ipfs/HASH, which is immutable.
- If a peer is in your vicinity (possibly discovered using mDNS/DNS-SD), you’d end up downloading directly from that peer.
- Because IPFS can essentially be used to “mirror” the contents servers by an HTTP server such as ftp.debian.org, and because it exposes an HTTP interface (the “gateway”), it could be integrated in a package manager without making any modifications
- However, better integration can help, for example:
  1. you still go to, say, ftp.debian.org, to get a list of package name/IPFS hash pairs;
  2. from there you fetch the package itself directly over IPFS, using its hash.
- IPNS (“name space”) allows use get a persistent identifier and to update it every time you publish new content, with URLs like /ipns/PEERID/file-name.txt. Updates are published via publish/subscribe (“pubsub”).
- Peer IDs are base64 strings so they are not human-readable. A workaround is to add a TXT record to a DNS entry, with “_dnslink=/ipfs/HASH”.

User stories II

# User Stories

[Note: these are in rough priority order]

1. Someone uploads a malicious* package version to a package repository. End user types "apt-get install <package>". The malicious package version does *not* get installed.

* In this document we say "malicious" but this also includes honest mistakes where the source code going into the build does not match what was checked in.

1b. Same as above, but for a user who chose an alternate package repository which may be less trustworthy than the central repository.

1c. Same as above, but for a user not using a package manager. For example, downloading static binary from GitHub or a Docker image from Dockerhub.

1d. User configures the definition of what trust means, e.g. at least N rebuilders have "certified" the binary, a particular rebuilder has certified, etc. [TODO: define "certified".]

2. User wants to contribute to help build trust. They rebuild packages and publish their results so that other users can detect bad binaries.

3. End user chooses operating system based on the trustworthiness of its binaries. How can they gain insight?

3b. Distribution publishes statistics about trustworthiness, e.g. % of reproducible packages in repo, to give others trust in their distribution. Also to compete and push industry toward better place.

4. Developer builds something locally, which they trust, and remotely (CI/CD), which they don't trust. By comparing hashes they don't need to upload large amounts of data, possibly over a weak link, to gain trust in the remotely built binaries. Example use case: Tor Browser, which takes ages to build and has huge artifacts.

5. System administrator wants to verify that none of the binaries running on their system are affected by known vulnerabilities. Can use a blacklist of "bad" artifact hashes.

6. System administrator applies a security policy on the source code that run on their system, e.g. "all source code must contain sign off by another person" or "do not deploy AGPL binaries." The deployer proves that the binary If you can prove that a binary came from particular sources, you can apply the policy on the source to know whether it's OK to allow the binary.

7. Package maintainer's system is compromised. Malicious binaries built on their system are not accepted by the package repository.

8. Author of software performs necessary steps to make their product reproducible, with tests to prevent regressions. Doing so should be easy.

[TODO: any other error cases?]

## Out of scope

N. A user wants to rebuild the entire operating system from source. [If they rebuild everything, why do they need reproducibility?]

Alpine

Alpine link added in https://maintainer.zq1.de/#bash JavaScript part

* kpcyrd worked on adding Alpine to https://tests.reproducible-builds.org

* ncopa worked on packaging diffoscope. There was various failures in the testsuite.

Tools matrix II

See notes of the 2-part "Tools matrix" session on page 58.

Bootstrapping III

Goal: Use mes as binary seed for Nix

Looking through the boostrappable documentation:

- No graphs!
- some outdated information (e.g. about the state of projects)
- core idea should be presented more concisely

The website sources are available here:
http://git.savannah.gnu.org/cgit/guix/bootstrappable.git/

Looking at Debian:

- base image size (current sid): 122MB
- apt-get build-dep gcc: 322MB
- This is a probably sufficient, but definietely not fully necessary set of bootstrap binaries
- https://wiki.debian.org/DebianBootstrap
  build profiles to build minimal package variants used to simplify the bootstrap process
- https://wiki.debian.org/HelmutGrohne/rebootstrap
  cross-building bootstrap of architectures

TinyCC bootstrap (via guile + mescc) in Nix:
https://github.com/edolstra/nixpkgs/commit/800bd373c0958d3fda355df9d1d0a49eea5815c8
\o/

## Trust models II

# Trust models II

The goal was to identify possible trust models and architectures: how can we protect users from non-reproducible packages and identify the global consensus on what binary package hash corresponds to a source package. We defined two models.

Manipulations of the source code by the uploading maintainer may be allowed.

## questions

what may be malicious:

- package archive/repository
- minority of rebuilders

## models/architectures

supply chain steps:

1. upstream
2. downstream/distribution maintainer
3. a benevolent archive builds twice to make sure package is reproducible
4. create binary package
5. install on client and some rebuilder checks (to be defined later)

We now describe two models with different assumptions and security properties. In a paranoid world we could want both models simultaneously.

### based on a transparency log

goal: create a global consensus on what constitutes the "correct" binary to a source package

- add certificate transparency-style Merkle tree log server
- archive submits source code and meta data into the log server
- archive publishes inclusion promise issued by log server
- client verifies log inclusion of release and consistency of log server
- monitors verify consistency of log server and rebuild
    - investigate all releases/source packages
    - rebuild them
    - compare to the binary released by the archive
    - in case of error divergence: alert rebuilder/monitor operator

notable properties:

- security updates don't have to wait for rebuilders
- one honest monitor is enough
- reproducibility check is potentially done /after/ installation
- investigation of malicious source or binary provided by the log server

⇒ only detects, does not prevent reproducibility failures

### based on supply chain-security

- rebuilders retrieve source packages from archive
- signed results are available to clients over HTTP API
- clients require quorum of positive rebuilder indications
- rebuilder identities configured locally
- root key (public part) must be present on client, to verify configured threshold and rebuilder public keys

- configuration defaults may be overridden locally

notable properties:

- prevents reproducibility failures (all checks complete before installation)
- matches package to be installed with packages rebuilt by rebuilders
- requires number of reliable rebuilders operated independently of distribution
- reproducibility of package manager client package itself prevents malicious archive from tampering with configurations

## further considerations

### trustworthy rebuilders

How do we get reliable third-party rebuilders, NOT operated by the distribution? How do we get enough (how many?) of them?

- trust management for rebuilders?
- federated? centralized? web of trust?

### can we integrate git into the log server Merkle tree?

Not sure, we don't know the documented security guarantees of git.

### Checking reproducibility is not enough

We should get guarantees from further up the supply chain, e.g. that someone vouches for the source code that is used by all rebuilders

## Resources

[Rebuilder NYU (server)](https://reproducible-builds.engineering.nyu.edu/)

[Rebuilder University Bergen (server)](http://158.39.77.214)

[in-toto (website)](in-toto.io)

[in-toto apt transport (repo)](https://github.com/lukpueh/apt-transport-in-toto/tree/rebuild-checking)

[software transparency architecture](https://arxiv.org/pdf/1711.07278)

[software transparency at DebConf18](https://debconf18.debconf.org/talks/104-software-transparency-package-security-beyond-signatures-and-reproducible-builds/)

[On Omitting Commits and Committing Omissions: Preventing Git Metadata Tampering That (Re)introduces Software Vulnerabilities]
(https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/torres-arias)

Test infrastructure

Test infrastructure

https://tests.reproducible-builds.org/reproducible.sql.xz (MB)

Schema https://tests.reproducible-builds.org/reproducibledb.html

M started adding a distribution field in the test database model.

* How to deal with components/repos (main/contrib debian), arch (core,extra,community), openSUSE (has suites and repos: Factory, Leap:15.0, home:bmwiedemann)?

* Distros allowing multiple binary or source versions of the same package?

* Have one database for all or one database per distro?

* Two kinds of distros, one where we just import the json and the other where we reproduce stuff?

* Check for special cases when they are found

* Add a script for Arch JSON output, similiary to debian json output (bin/reproducible_json.py)

* Guix json output from guix-challenge to be coded to allow importing into the jenkins DB

* Add a detail package page for Arch, and a link to the debian page with an icon of build state

* The note distro sharing can be done independent of sharing the database. https://salsa.debian.org/reproducible-builds/reproducible-notes

    - Some Python scripts need to be rewritten to understand the new notes format, reproducible_html_notes.py

    - Script to convert old nodes to new scripts

    - misc/* in notes.git

OPAM reproducibility

opam hooks -> preinstall, postinstall

opam package install -> list of binaries and locations

local reproducibility (storing hashes on the first run and check on the second round)

check reproducibility w.r.t. hashes provided by independet rebuilders

opam should have a way to clearly specify the external dependencies in order to be able to exactly record and reproduce the

build environement, including versions.

- GCC
- make
- ld
- libgmp
- shell

independent rebuilders should be provided with buildinfo files to be able to reproduce exactly the build environment

opam2 runs on a sandbox that at the moments binds /usr in the bubblewrap chroot. The idea would be to restrict the sandbox to only the declared dependencies.

I did a bit more work on the opamrepro script. You can find my last version here :

https://gitlab.com/abate/opam-repro

Initial findings are:

- calendar.2.04 is NOT reproductible
- ocamlgraph.1.8.8 is NOT reproductible
- ppx_deriving.4.2.1 is NOT reproductible
- ocp-build.1.99.20-beta is NOT reproductible

- irmin.1.4.0 is NOT reproductible

This analysis was done on a small set of packages and ignoring *.cmt* files.

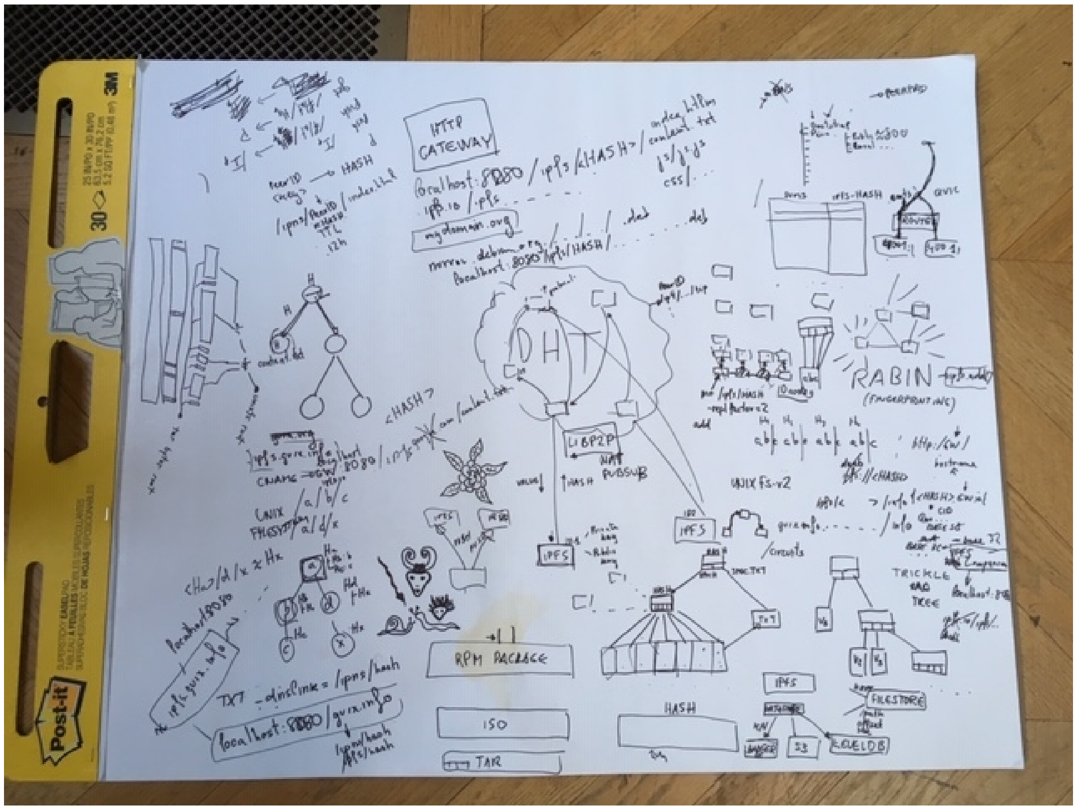
it would be interesting to generate output for all opam packages.

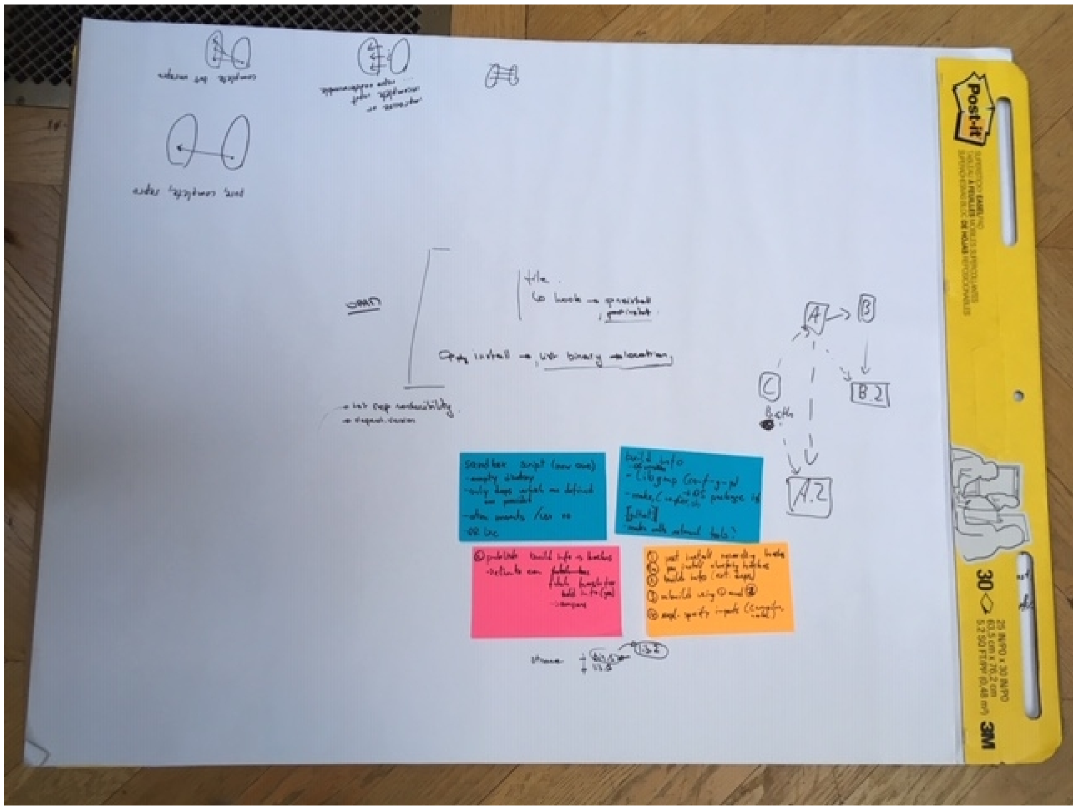
Next steps

- use a nicer way to store locally build results (instead that a text file in /tmp)
- run opam twice automatically on a package to check for repoducibility (maybe adding a command ?)
  - see https://salsa.debian.org/reproducible-builds/reprotest , a utility that modifies environment / build path / ...
- generate a buildinfo file to describe the build environment
- compare the result with a remote repro builder

https://gist.github.com/hannesm/224e777fb29cef1b1a9fdda2ea6881fe

Hack time

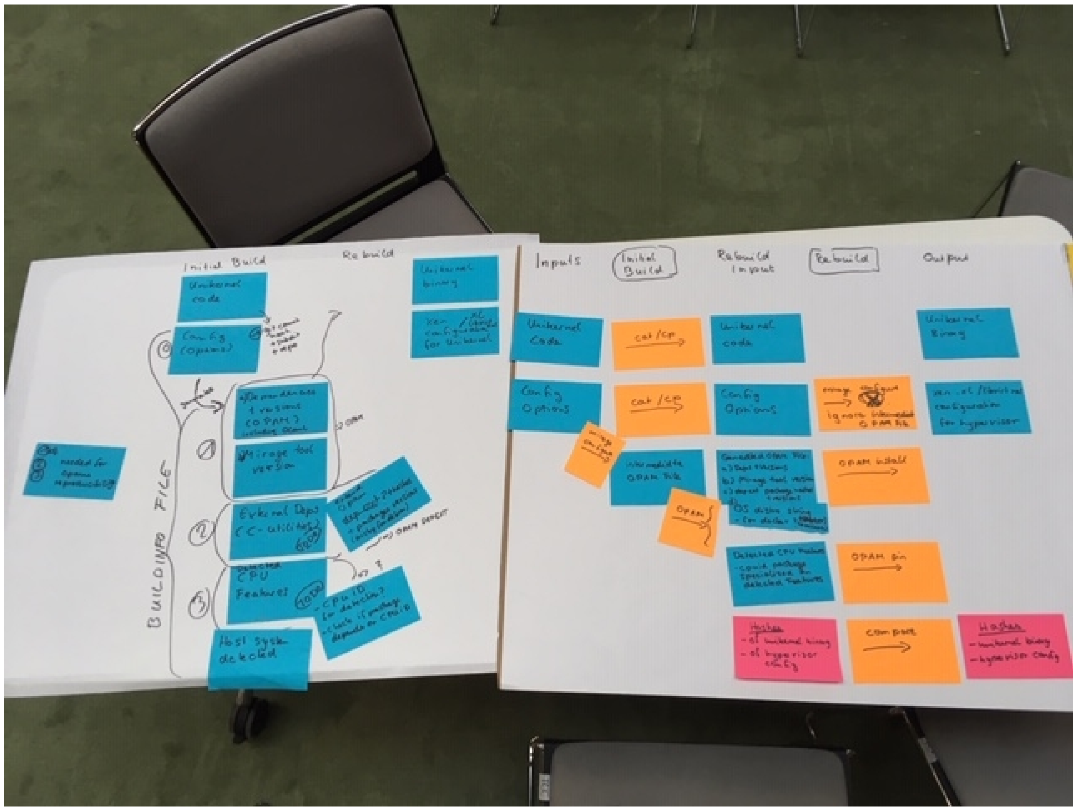

HTTP GATEWAY
localhost:8080 / ipfs / <HASH> / index.html
mydomain.org
DHT
LIBP2P
NAT
PUBSUB
RABIN
(FINGERPRINTING)
UNIX fs-v2
IPFS
RPM PACKAGE
ISO
TAR
HASH
FILESTORE
S3
LEVELDB
TRICKLE
DAG
TREE
localhost:8080 / gura.info
UNIX FILESYSTEM
CNAME
<HASH>

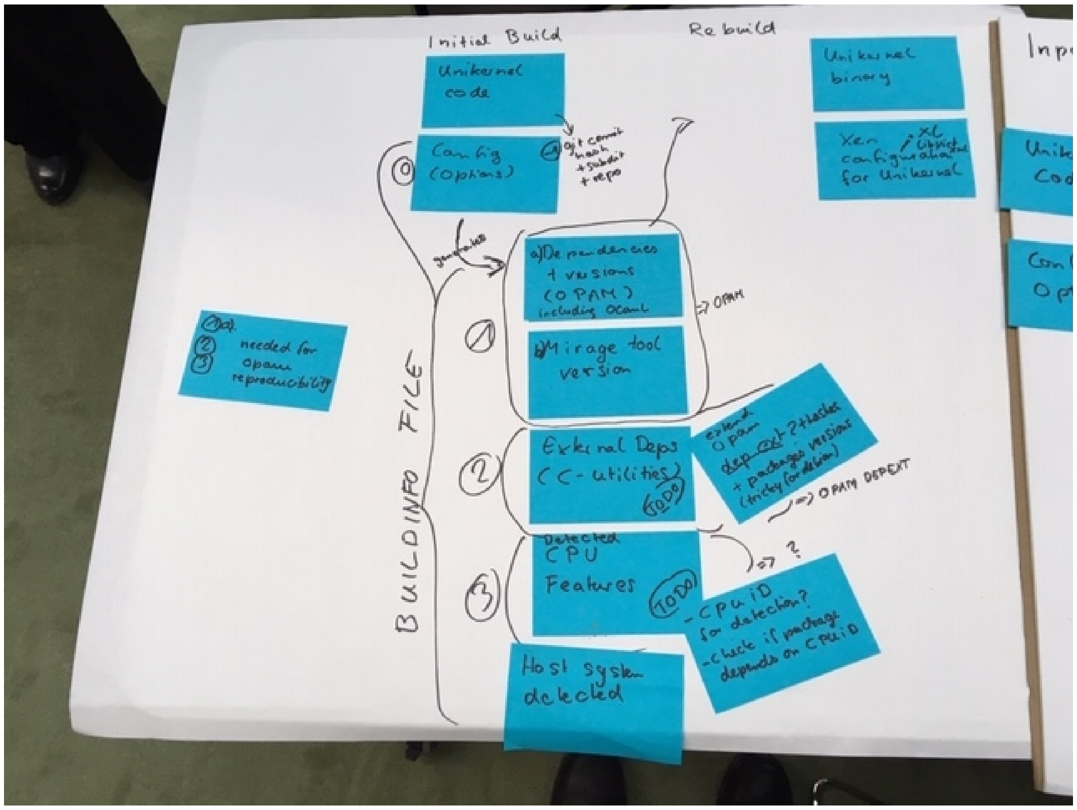


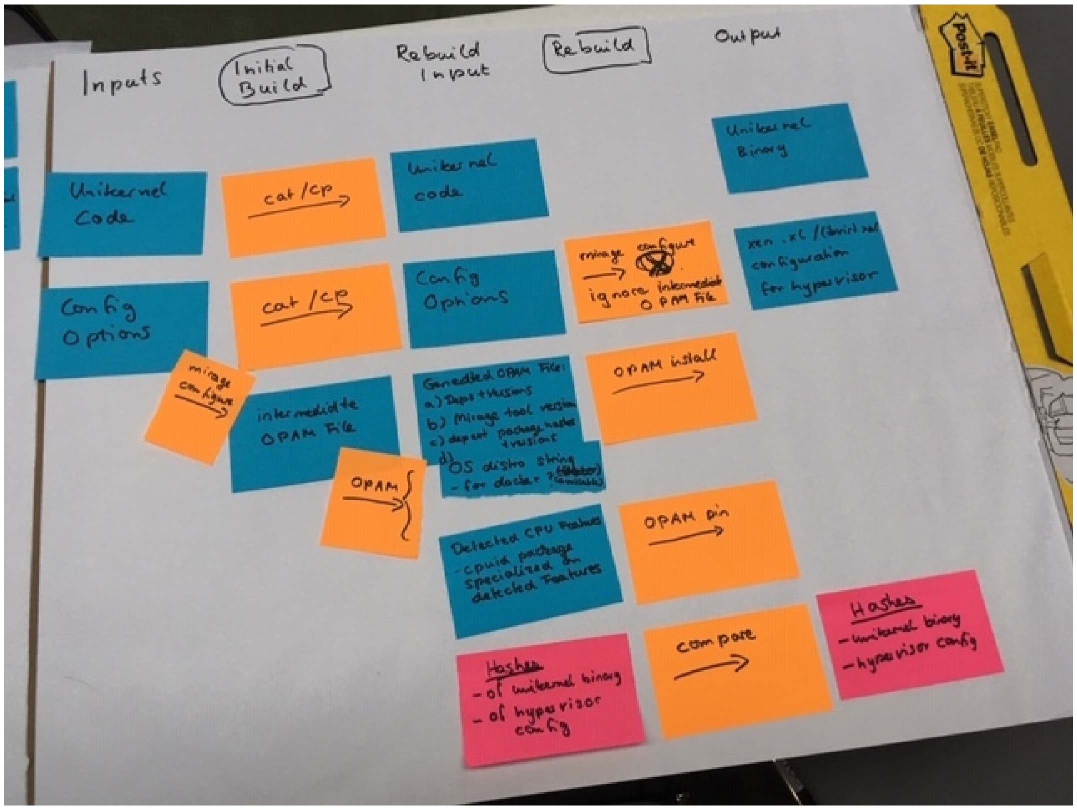
Day 3

Strategic working sessions IV

MirageOS




Initial Build
Rebuild
Unikernel code
Config (Options)
git commit hash + submit + repo
Unikernel binary
Xen configuration for Unikernel
xl libvirt
generates
a) Dependencies + versions (OPAM) including OCaml
b) Mirage tool version
=> OPAM
①a) ② ③ needed for opam reproducibility
BUILDINFO FILE
External Deps (C-utilities)
TODO
extend opam depext? + hashes + packages versions (tricky for debian)
=> OPAM DEPEXT
Detected CPU Features
TODO
=> ?
- CPU ID for detection? - check if package depends on CPUID
Host system detected



1st phase: specify the artefacts

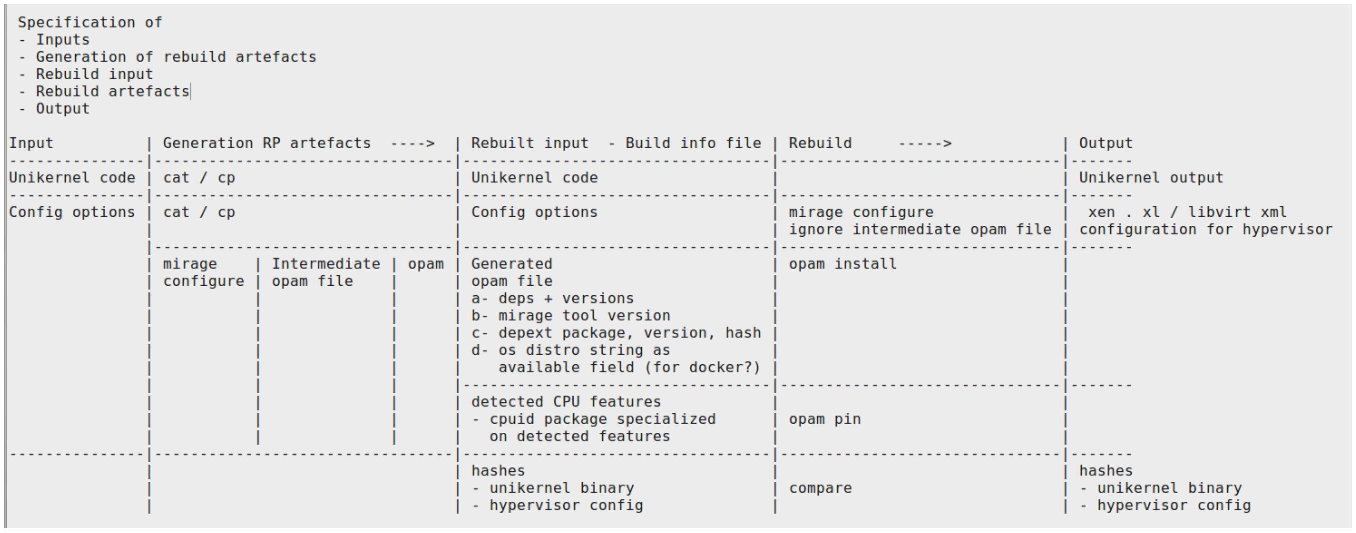
2nd phase: specify the transformations

3rd phase: specify the things to reproductibilytification

From the first input, using cp / mirage-configure & opam (todo), generates all informations for the rebuild input. Once we get these, using "simple" commands, it is possible to generate the rebuild and then compare unikernal binary & hypervisor config

Specification of

- Inputs
- Generation of rebuild artefacts
- Rebuild input
- Rebuild artefacts
- Output



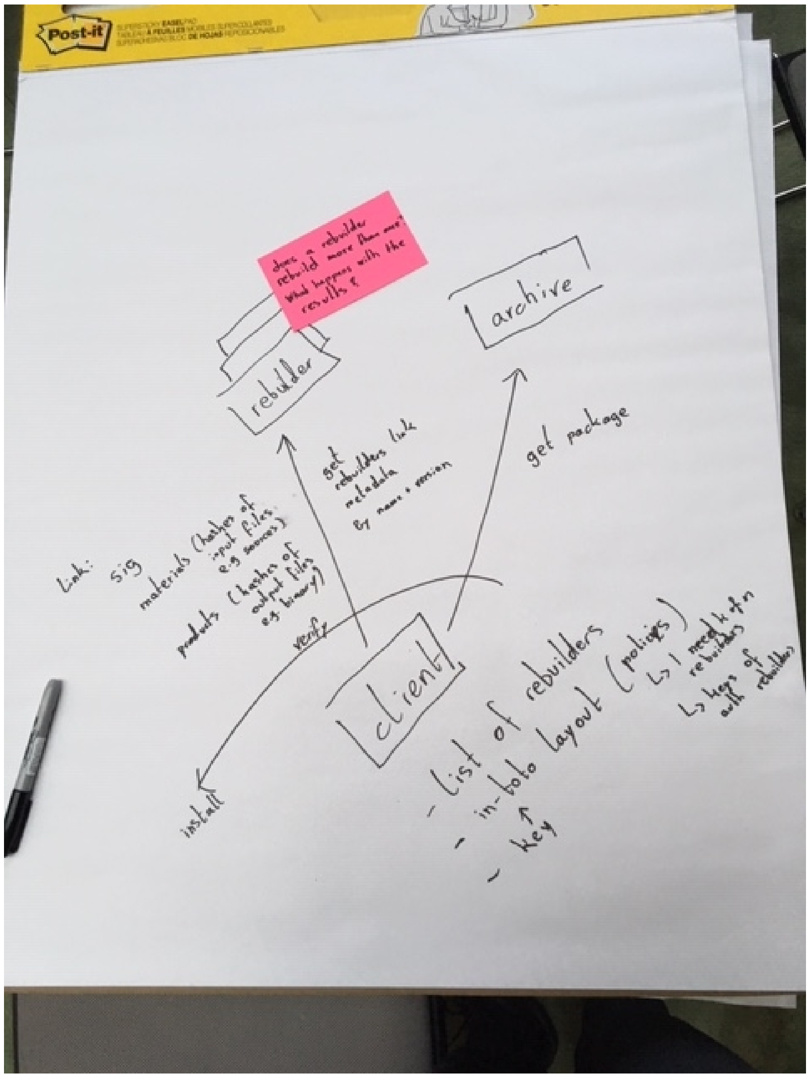
```
 Specification of
 - Inputs
 - Generation of rebuild artefacts
 - Rebuild input
 - Rebuild artefacts
 - Output

Input          | Generation RP artefacts   ----> | Rebuilt input  - Build info file | Rebuild     ----->            | Output
---------------|---------------------------------|----------------------------------|-------------------------------|-------
Unikernel code | cat / cp                        | Unikernel code                   |                               | Unikernel output
---------------|---------------------------------|----------------------------------|-------------------------------|-------
Config options | cat / cp                        | Config options                   | mirage configure              |  xen . xl / libvirt xml
               |                                 |                                  | ignore intermediate opam file | configuration for hypervisor
               |---------------------------------|----------------------------------|-------------------------------|-------
               | mirage    | Intermediate | opam | Generated                        | opam install                  |
               | configure | opam file    |      | opam file                        |                               |
               |           |              |      | a- deps + versions               |                               |
               |           |              |      | b- mirage tool version           |                               |
               |           |              |      | c- depext package, version, hash |                               |
               |           |              |      | d- os distro string as           |                               |
               |           |              |      |    available field (for docker?) |                               |
               |           |              |      |----------------------------------|-------------------------------|-------
               |           |              |      | detected CPU features            |                               |
               |           |              |      | - cpuid package specialized      | opam pin                      |
               |           |              |      |   on detected features           |                               |
---------------|---------------------------------|----------------------------------|-------------------------------|-------
               |                                 | hashes                           |                               | hashes
               |                                 | - unikernel binary               | compare                       | - unikernel binary
               |                                 | - hypervisor config              |                               | - hypervisor config
```

TODO:

- mainly depext: install pkg with version, retrieve installed package, version & hash, lock with depext
- on generated opam file, add os/ on available field
- hashes : generation & comparison
- mirage-configure: add mirage package - see https://github.com/mirage/mirage/pull/953
- include in the build info file the cpuid.ml

reprotest (see https://salsa.debian.org/reproducible-builds/reprotest also a debian package) -> run on opam repo ?

Rebuilders III


does a rebuilder rebuild more than once? what happens with the results?
archive
rebuilder
get rebuilders link metadata by name + version
get package
Link: sig
materials (hashes of input files e.g. sources)
products (hashes of output files e.g. binary)
verify
client
install
- list of rebuilders
- in-toto layout (policies)
↳ 1 need k of n rebuilders
↳ keys of all rebuilders
key

Rebuilders take 3

Recap:

- All agreed on we have an upstream repository
- The source code is packaged by the distribution maintainer
- The distribution maintainer uploads the source package in to the archive
- Binary packages are then compiled, the archive or main repository will rebuild twice for reproducibility testing

Two models for security checks for reproducible builds

 - When installing a package, you check it's building reproducibly
 - Audit server with a log, global consensus, what the correct binary for a source package looks like

The log server shows the entire history of the rebuilt packages.

The rebuilder has a list of packages

Upstream problems:
 - Mirroring upstream tars, in case they're deleted
 - Changing upstream artefacts

Discussion about verifying the correct source code is being used...

If bad source code gets in, then a bad binary comes out...

The source package is changed to look at a different, compromised upstream repository

F-Droid produces binaries in some cases that bit-for-bit match the upstream binary release.

For Debian, the client fetches metadata. It checks with the individual rebuilders to find out if they agree on the resulting binaries.

Discussion of binaries being misrepresented (a binary produced for nginx, being misrepresented to the user as apache)

List of things we want to talk about:

- Rebuilder API
- End to end security architecture
- How this applies to Guix
- tor browser, reproducible builds, rebuilder for Tor browser
- Want to get rebuilders out there in the real world
- How the rebuilder API would fit the F-Droid model
- How to populate the list of trusted rebuilders
- Non-distribution case for rebuilders
- Hosting rebuilders
- Debian specific ansible scripts, making that more general
- How to do cross-distro rebuilders

in-toto (is a thing?) It's a framework for creating and verifying things

https://buildinfo.debian.net/

It's a service, a bit hacky, different distros have different requirements

When a client queries the rebuilder, it's an online operation.

Fetching metadata from rebuilders shouldn't necessarily be an online operation, at least at install time.

Where does the source code come from for rebuilders? The buildinfo file contains the hash for the source.

The rebuilder creates a in-toto file.

What to standardise and share?

Should rebuilders work for multiple distributions?

 - Should be hashes of inputs and outputs
 - Hashing the buildinfo file...?

 - A value of standardisation, could be things that are not a distro?

 - For Torbrowser, the input would just be a commit

Maybe a standardised approach could help those who don't already have a concept

Existing rebuilders
 - A couple of universities run the rebuilder implementation for Debian.
 - Guix has a couple of build farms

We want to produce documentation on how rebuilders work.

Maybe a desired outcome is some advertising to encourage people to run rebuilders.

# Switch to the client side

 - The archive provides source packages, you have a number of rebuilers
 - The client has a list of rebuilders
 - in-toto layout, which has the policies
   - I need K rebuilders to agree...
 - also have a key for the layout

Rebuilders building twice... you should only have one version

Rebuilders building multiple times, is that a good idea?

Publishing history from rebuilders could be interesting, like a certificate transparency log

Policy or architecture...

Links:

http://158.39.77.214/

https://reproducible-builds.engineering.nyu.edu/

APT transport implementation detail: https://github.com/in-toto/apt-transport-in-toto/blob/develop/intoto.py#L69

Afternoon rebuilder discussion result

https://photos.app.goo.gl/tKqZoaMFhTYvXuvQ9

## PGO Profile-guided optimization

# PGO

Profile guided optimization are currently unreproducible and for Arch Linux enabled for some packages such as Firefox. Python.

Bernard has written an example of unreproducible PGO, and has actually fixed PGO issues for some packages.

https://build.opensuse.org/request/show/647618

https://github.com/bmwiedemann/theunreproduciblepackage/tree/master/pgo

* Investigate if it's possible to make PGO reproducible
* Is there a difference in PGO of Clang vs. GCC?
* PGO takes input to create a profile of the profile guided program, time is (not directly) a factor in getting a reproducible optimised binary. Since some
* Investigate if PGO is hardware dependend? Or is the issue how longer you run a Proifle how much different optimisations you get?

A simple PGO example with a fixed input is reproducible on at least two different sets of hardware. The question is, if a more complicated program with different input is reproducible.

Worked on identifing if grep can be PGO'd and reproducible such as openSUSE has achieved. The grep build isn't always reproducible but sometimes is. This might depend on how long it takes to run the test suite which is fed to the

http://pkgbuild.com/~jelle/grep-unrepro/

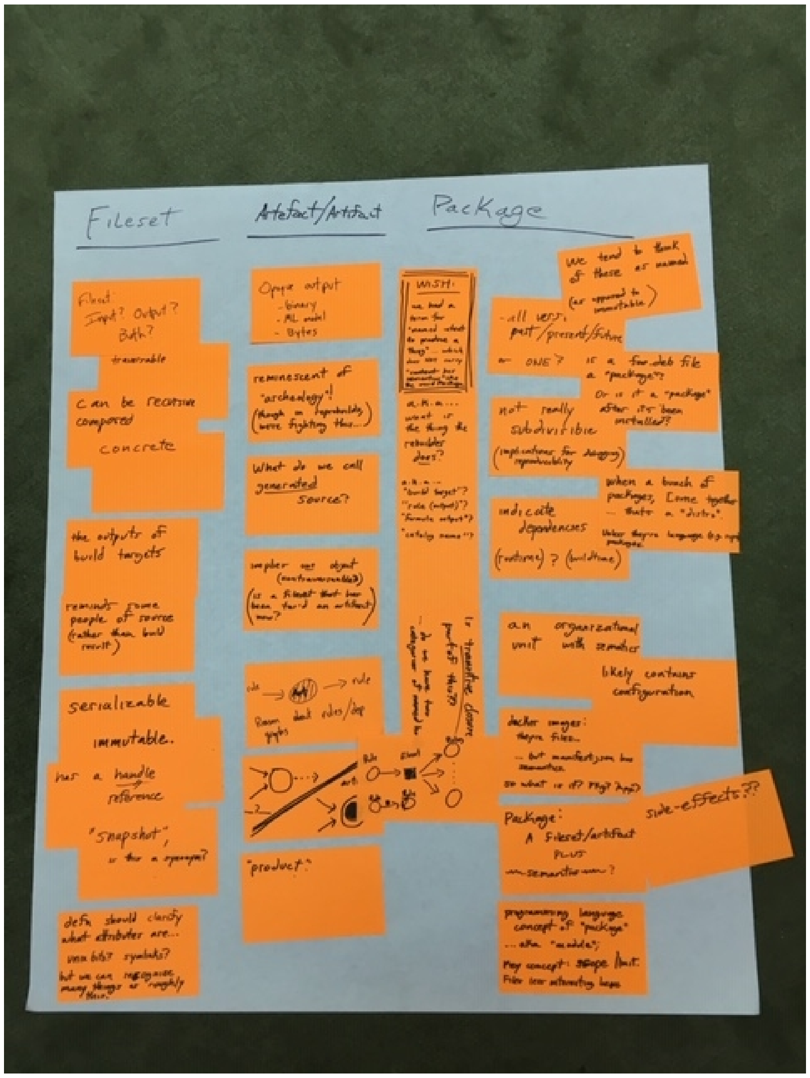
PGO works

- Compile with CFLAGS="-fprofile-generate -fno-profile-values"
- Run your test suite with CFLAGS="-profile-generate", this generates .gcda files
- make clean and make with CFLAGs="-fprofile-use" this compiles using the .gcda profile files

with our grep example:

- binaries where sometimes unreprodicible
- gcda profiles where unreprodicible

Terminology issues II



Filesets

Can be input or output

Can be recursively composed

Output of build targets

Serializable

Reminds some people of source, rather than build results.

Immutable

has a _handle_/reference

Synonymous with snapshot?

Definition should clarify attributes: e.g. unix bits, symlinks. But we can recognize many things as roughly this.

Concrete

Artefact/Artifact

Opaque object (binary, ml model, bytes)

Noun is reminescent of "archeology"!

Generated Source?

Implies one object (non-traversable); Is a fileset that has been turned into a "tar" archive an artifact?

Product of some process

Packages

A package could be the "least granular unit".

Are packages all past/present/future versions or just a particular version?

Cannot be recursive

Not subdivisble which has implications for debugging reproducibility

Indicates dependencies (runtime/buildtime/install)?

Organization unit with semntics

Docker images: they're files, but also have a mainfest.json which contains semantics. So is this a package? An application?

In a programming language the concept of a "package" or "module" to define scope. Files are less interesting.

Is a deb a package by itself or is it only a package when it is installed?

A bunch of packages coming together... is a "distro"... Unless they're language packages (e.g. npm).

Likely contains configuration data.

We tend to think of packages as something that is named rather than immutable. The name must be resolved to actual "things".

Side effects?

Wish list

We need a term for "named intent to produce a thing"... which does NOT carry "content has semantics" like the word "package". What is the thing the rebuilder _does_?

Potential terms: "build target", "rule (output)", "catalog name"

Is **transitive closure** part of this? Do we have two missing terms?

"Rule/Transform" -> Fileset -> Other Rules/Transforms

Need to reference the output of an unevauluated rule/transform.

Build

Builds are a function of a fileset to a fileset

tl;dr;

- Filesets/Artifacts are values
- Builds are functions
- Builds are a function of a fileset to a fileset
- "Packages" are a fileset with semantics

Debugging for reproducibility issues

We discussed the 10 known sources of indeterminism documented in https://github.com/bmwiedemann/theunreproduciblepackage/

and covered Bernhard's how-to-debug list:

- autoclassify
- provenance (build log, make V=1 VERB=1 , strace -f) ; autoprovenance
- grep source for strings next to diff ("This file was autogenerated at")
- grep source for known troublemakers (`date, $(date, %Y, year, strftime, ctime, asctime, stat (mtime,inode,dev), readdir, os.listdir, os.walk, glob.glob, find, random, fprofile-use)
- review diffs in .rb.buildroot.diff - e.g. .c .h .a configure Makefile
- use difflog to see if something interesting happens related to indeterministic files
- review relevant files (configure{,.ac}, Makefile{,.in,.am}, CMakeLists.txt, setup.py)

to be added into https://github.com/bmwiedemann/reproducibleopensuse where the useful (OBS/openSUSE-related) debugging tools are

Getting openSUSE on testing infrastructure

openSUSE added to t.r-b.o db and html is being generated.

For this, openSUSE's published http://rb.zq1.de/compare.factory/reproducible.json is imported into reproducibledb

https://salsa.debian.org/qa/jenkins.debian.net/commit/5e429a958763d1ba2fd90aafffeb64ab25e7e396

https://salsa.debian.org/qa/jenkins.debian.net/commit/673fd148299a943de9af0daf1d13678a9617e1b1

https://salsa.debian.org/qa/jenkins.debian.net/commit/502138c149494047eb19900b7ac408d64079cbe3

https://salsa.debian.org/qa/jenkins.debian.net/commit/d216ef4f0e1e0015e3a0180680e0d8faee7809f4

https://salsa.debian.org/qa/jenkins.debian.net/commit/9704f22211fc1a9a04d6fc0335b76191fec89438

https://salsa.debian.org/qa/jenkins.debian.net/commit/19b01d7ea611ebd84bdf1a76a64a396ff7e9a5fa

https://salsa.debian.org/qa/jenkins.debian.net/commit/68e0d31da013b7b7b6d63aeebbb6c062cdbd2800

https://salsa.debian.org/qa/jenkins.debian.net/commit/dd60da7d54f551c5d08e8f75607fff137b130fa2

https://salsa.debian.org/qa/jenkins.debian.net/commit/c891a86566efbd6c4aed32628dd0d94f0717a363

https://salsa.debian.org/qa/jenkins.debian.net/commit/a30c3c18f9fc9e6adcbf53fdcf2dfed9a0452a96

Strategic working sessions V

Qubes OS reproducible hacking

Qubes OS hacking:

packagage building - it works!

https://github.com/QubesOS/qubes-builder/pull/66

https://github.com/QubesOS/qubes-builder-rpm/pull/41 Fix handling components with multiple packages

https://github.com/QubesOS/qubes-vmm-xen/pull/49

https://github.com/QubesOS/qubes-builder-debian/pull/26

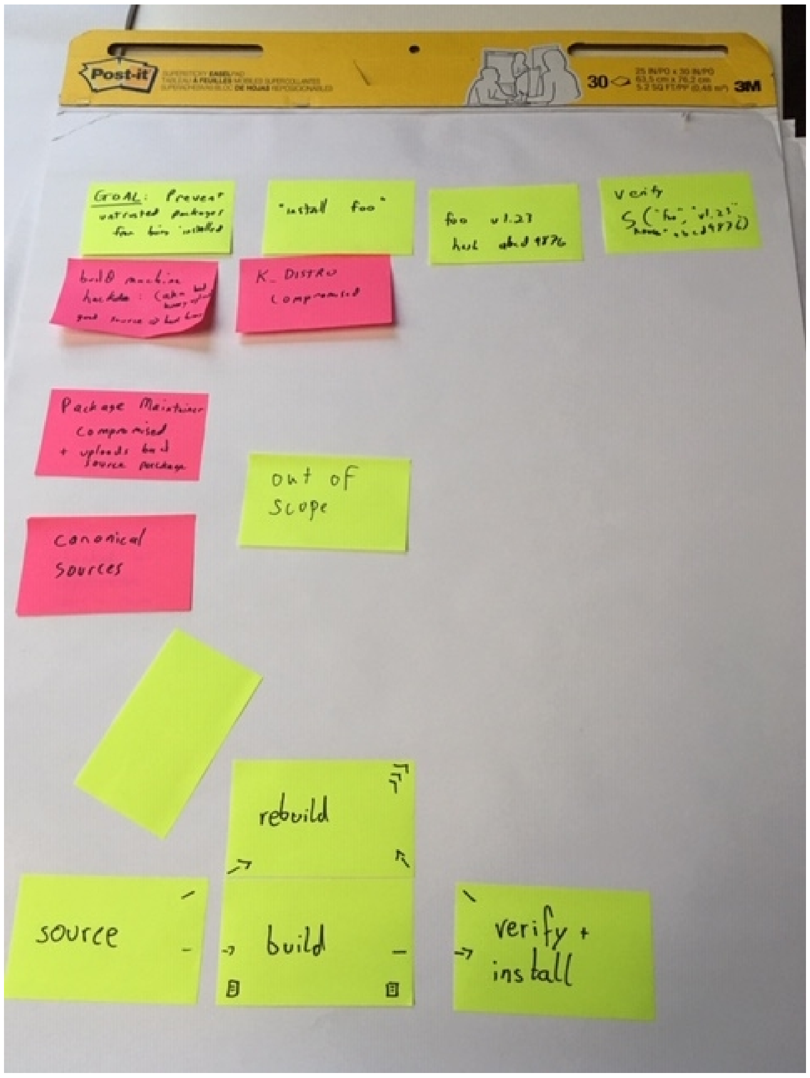
Transparency log for Debian and Tor Browser

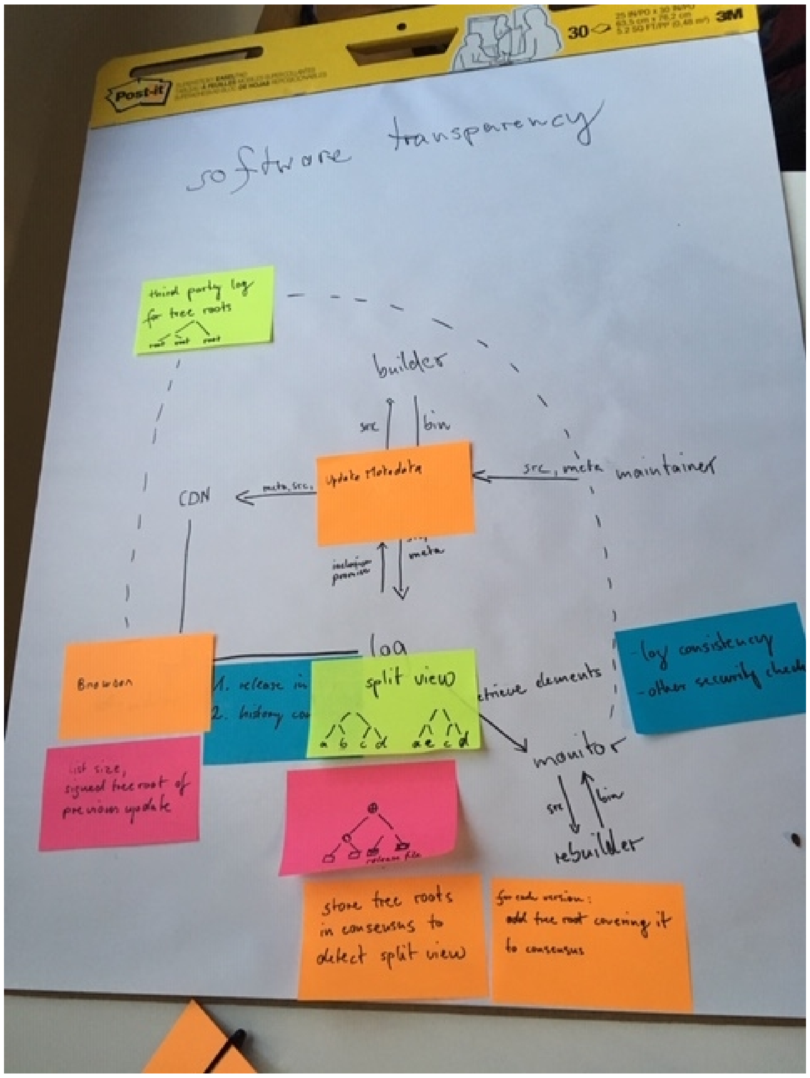
Torbrowser and Debian can follow a similar model

- some more investigation is needed which threats are addressed by tunneling some of the communication edges over tor
- addressing split view attacks, the directory authorities include into their consensus proposal:

  for each supported Torbrowser version, a signed tree root which covers that release
- clients can then check their view is consistent with the DA consensus

Hack time


GOAL: Prevent untrusted packages from being installed
"install foo"
foo v1.23
verify
build machine hacked
K_DISTRO compromised
Package Maintainer compromised + uploads bad source package
canonical sources
out of scope
rebuild
source
build
verify + install


software transparency
third party log for tree roots
builder
src
bin
Update Metadata
src, meta
maintainer
CDN
meta, src.
inclusion proof
meta
log
Browser
1. release in
2. history co
split view
a b c d
a e c d
retrieve elements
log consistency
- other security checks
monitor
src
bin
rebuilder
list size, signed tree root of previous update
store tree roots in consensus to detect split view
for each version: add tree root covering it to consensus