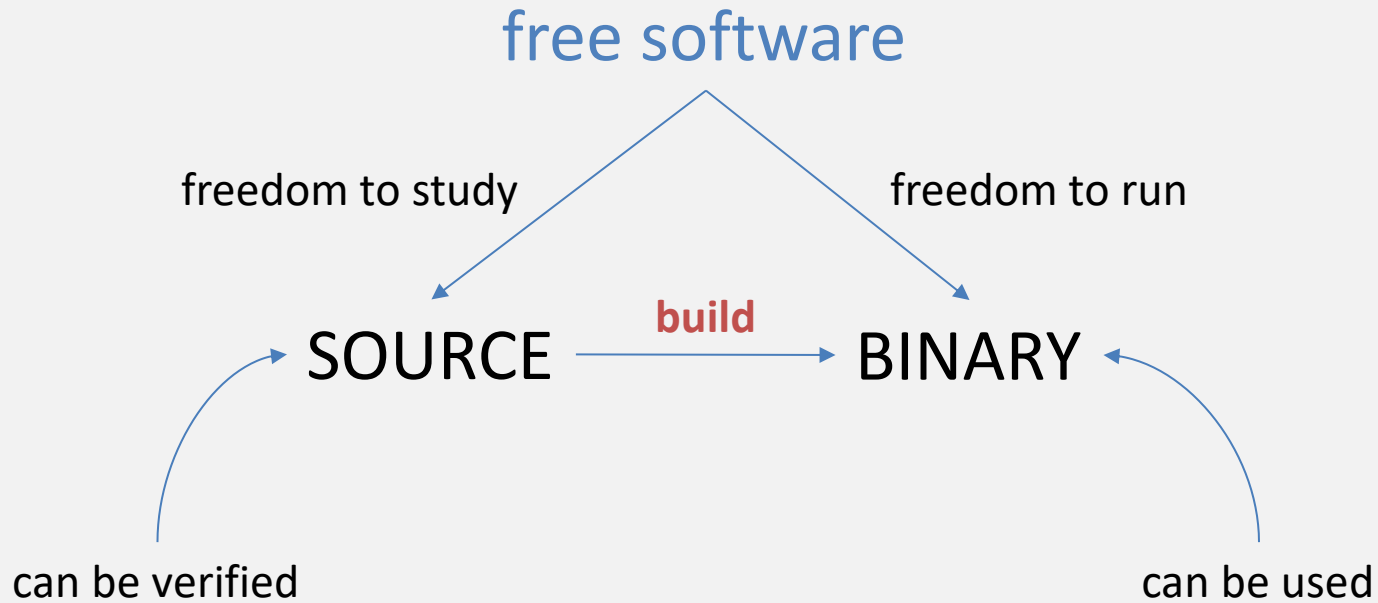


**Reproducibility:
from arithmetic operations
to building system packages**

Nikolay Vizovitin

Saint Petersburg
3-4 November 2017



REPRODUCIBLE

- “Anyone can repeat it”
- Given the same:
 1. Input data
 2. Relevant attributes of environment
- Produces bit-by-bit identical outputs

REPEATABLE

- “It produces same results”
- Given the same:
 1. Input data
 2. Runtime
- Produces bit-by-bit identical outputs

[\[edit\]](#) (S//NF) Strawhorse: Attacking the MacOS and iOS Software Development Kit

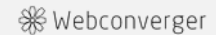
(S) Presenter: ██████████, Sandia National Laboratories

(S//NF) Ken Thompson's gcc attack (described in his 1984 Turing award acceptance speech) motivates the StrawMan work: what can be done of benefit to the US Intelligence Community (IC) if one can make an arbitrary modification to a system compiler or Software Development Kit (SDK)? A (whacked) SDK can provide a subtle injection vector onto standalone developer networks, or it can modify any binary compiled by that SDK. In the past, we have watermarked binaries for attribution, used binaries as an exfiltration mechanism, and inserted Trojans into compiled binaries.

(S//NF) In this talk, we discuss our explorations of the Xcode (4.1) SDK. Xcode is used to compile MacOS X applications and kernel extensions as well as iOS applications. We describe how we use (our whacked) Xcode to do the following things: -Entice all MacOS applications to create a remote backdoor on execution -Modify a dynamic dependency of securityd to load our own library - which rewrites securityd so that no prompt appears when exporting a developer's private key -Embed the developer's private key in all iOS applications -Force all iOS applications to send embedded data to a listening post -Convince all (new) kernel extensions to disable ASLR

(S//NF) We also describe how we modified both the MacOS X updater to install an extra kernel extension (a keylogger) and the Xcode installer to include our SDK whacks.

<https://theintercept.com/document/2015/03/10/strawhorse-attacking-macos-ios-software-development-kit/>



When “math” is non-reproducible

ARITHMETIC OPERATIONS

1. Produce consistent results
 1. From one run to the next
 2. From one set of build options to another
 3. From one compiler (or runtime) to another
 4. From one platform to another
2. While maintaining accuracy and performance
3. These objectives often conflict

It's convenient to think that:

1. for integer operations:

1. $-a/b == -(a/b)$

2. for floating-point operations:

1. $(a+b)+c == a+(b+c)$

2. $a+b == b+a$

3. $a*b == b*a$

4. and even $a+b == a+b$


```
$ cat div.c
#include <stdio.h>

int main()
{
    int result = -6 / 4;
    printf("%d\n", result);
    return 0;
}
$ gcc div.c -o div && ./div
-1
```

```
$ cat div.py
#!/usr/bin/env python3

result = -6 // 4
print(result)

$ ./div.py
-2
```

```
$ cat assoc.c
#include <stdio.h>

int main() {
    const double x = 0.1;
    const double y = 0.2;
    const double z = 0.3;
    if ((x + y) + z != x + (y + z))
        printf("non-associative\n");
    return 0;
}
```

```
$ gcc -O2 assoc.c && ./a.out
non-associative
```

1. “Catastrophic cancellation”:

$$(1.0 + 1e100) + -1e100 == 0.0$$

$$1.0 + (1e100 + -1e100) == 1.0$$

2. Sum and multiplication are not associative

3. May happen “behind the scenes”

E.g. due to automatic parallelization or SSE

1. Addition and multiplication, as defined by IEEE-754, is guaranteed to be commutative:
Each of the operations shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then coerced this intermediate result to fit in the destination's format.
2. So $a+b == b+a$ and $a*b == b*a$?

Floating-Point: $a + b == a + b$

```
$ cat double.c
#include <stdio.h>

double add1(double x) {
    return x + 1.0;
}

int main() {
    const double x = 0.012;
    const double y = x + 1.0;
    if (y != add1(x))
        printf("ERROR\n");
    return 0;
}
```

```
$ uname -m
i686
$ gcc double.c && ./a.out
ERROR
$ gcc -ffloat-store double.c && ./a.out
$
```

1. $a + b \neq a + b$
2. Reproduces with -00 and -01
3. Due to excess precision in FPU
4. GCC Bugzilla [“not a bug” #323](#)

1. Different implementations may not have the same accuracy
2. No official standard dictates accuracy or how results should be rounded (except for some functions)
3. For many functions there's no correct rounding algorithm with bounded run time

1. Client-server architectures
2. Game development (multiplayer, replay)
3. Tools

1. Recommendations:
 1. Extensively test everything you rely on
 2. Control tradeoffs between reproducibility, accuracy, and performance with compiler options
 3. Consider using fixed point arithmetic and/or specialized math libraries
2. On floating-point determinism:
 1. goo.gl/HaFKzX (what every computer scientist should know about floating-point arithmetic)
 2. goo.gl/tTyV2x (deterministic lockstep methods viability for game networking)

How to build tools with reproducible output

ALGORITHMS

Practical recommendations:

1. Avoid using global state
2. Avoid using true randomness
3. Prefer pure functions
4. Use deterministic algorithms
5. Test reproducibility!

Provide a verifiable path from source to binary

REPRODUCIBLE BUILDS

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

Reproducible builds are a set of software development practices that create a **verifiable path from** human readable **source code** to the **binary** code used by computers.

This supports **software freedom** to **study** how the program works, and **change** it to your needs.

1. Resisting attacks
2. Smaller binary differences
3. Quality assurance
4. Increased development speed
5. Build documentation
6. Easier license compliance (GPL)

1. Use a deterministic build system
2. Define a build environment
3. Distribute the build environment
4. Provide a comparison protocol

How to achieve repeatable builds

DETERMINISTIC BUILD SYSTEMS

1. Unstable ordering of inputs and outputs
2. Capturing volatile global state
3. True randomness

1. Build environment definition governs what should be deterministic
2. In general:
 1. Ensure inputs are stable
 2. Ensure outputs are stable
 3. Capture as little as possible from the environment
 4. Avoid true randomness

non-reproducible *

```
$ cat Makefile
SRCS = $(wildcard *.c)
tool: $(SRCS:.c=.o)
      $(CC) -o $@ $^
```

* with GNU Make >= 3.82

reproducible

```
$ cat Makefile
SRCS = util.c worker.c main.c
tool: $(SRCS:.c=.o)
      $(CC) -o $@ $^
```

```
$ cat Makefile
SRCS = $(sort $(wildcard *.c))
tool: $(SRCS:.c=.o)
      $(CC) -o $@ $^
```

non-reproducible

```
$ tar -cf archive.tar src
```

```
$ find src -print0 |  
sort -z |  
tar --no-recursion --null \  
-T - -cf archive.tar
```

reproducible

```
$ tar -cf archive.tar \  
src/util.c \  
src/worker.c \  
src/main.c
```

```
$ find src -print0 |  
LC_ALL=C sort -z |  
tar --no-recursion --null \  
-T - -cf archive.tar
```

non-reproducible

```
$ cat deps.py
...
for module in deps.keys():
    version = deps[module]
    print('%s (>= %s)'
          % (module, version))
```

reproducible

```
$ cat deps.py
...
for module in sorted(deps.keys()):
    version = deps[module]
    print('%s (>= %s)'
          % (module, version))
```

1. Tune hashing for dictionaries:

1. `PERL_HASH_SEED`

2. `PYTHONHASHSEED`

3. Why hashing is randomized?

1. goo.gl/62RYhF (oCERT-2011-003)

2. goo.gl/7VLG7h (Algorithmic Complexity Attacks)

2. Account for locale settings (sorting)

1. Limit access to global state
(e.g. sandbox build process)
2. Reset global state for build
(e.g. environment variables)
3. Include required global state as explicit inputs, version it as source code

1. Any volatile input (e.g. from network) can disappear or change
(e.g. NPM left-pad package: goo.gl/ayZwep)
2. Do not rely on remote data
3. Otherwise:
 1. Ensure integrity (checksums)
 2. Keep backups

1. Make version information deterministic
2. Current date and incremental build IDs aren't useful
3. Build ID could be a hash of the source code
4. Version information could be extracted from sources:
 1. Explicitly recorded
 2. VCS revision
 3. Changelog entry
 4. Hash of the source code

1. Best to avoid
2. Base on changelog / last VCS commit
3. Enforce:
 1. Run build under `faketime` (has drawbacks)
 2. Implement `SOURCE_DATE_EPOCH` environment variable
(spec: goo.gl/NNvvju, support: goo.gl/JwpHZz)
 3. Post-process (remove or normalize timestamps)

1. If possible, better to explicitly set `LC_ALL`
2. `LC_CTIME` affects time format
3. `LC_COLLATE` affects sorting order
4. `LC_CTYPE` affects input and output of many tools

```
$ echo a B b c | xargs -n1 |  
LC_ALL=C sort
```

```
B  
a  
b  
c
```

```
$ echo a B b c | xargs -n1 |  
LC_ALL=en_US.UTF-8 sort
```

```
a  
b  
B  
c
```

1. Most formats capture build environment details (mtime, file ordering, users, groups, uids, gids, permissions)
2. Use specific options to make output stable
3. Or post-process with tools like:
 1. `strip-nondeterminism`
 2. `re-dzip.sh`

1. Some tools record sources paths
2. Most compilers include it into debug info,
and therefore BuildID as well
(preserved by `strip`)
3. Use tool options (e.g. `gzip -n`)
4. Or post-process output

1. Use a predetermined value to seed a PRNG
2. Use tool options and environment variables to set the seed

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
             // guaranteed to be random.  
}
```

xkcd.com/221

```
$ gcc -flto -frandom-seed=worker.c -c worker.c  
$ nm -a worker.o | grep inline  
0000000000000000 n .gnu.lto_.inline.7bc2ea8a
```

Sources of randomness:

1. Temporary file names
2. Generated UUIDs
3. Filesystem
4. Protection against complexity attacks
5. LTO (symbol names)
6. Unique stamps in coverage data files
7. ...

1. Make sure all values are initialized
(don't capture random memory blocks)
2. Use tools like Valgrind

```
build/cbfs/fallback/bootblock.bin
Offset 1, 10 lines modified
1 00000000: bada baab 0200 0000 5845 0000
   0000 009a .....XE.....
2 00000010: 1200 0000 9c5e 0000 4b1d 0000
   4045 0000 .....^..K...@E..
3 00000020: 0000 009a 17b9 e22a 009b 1d3c
   0020 bd27 .....R...<..'
4 00000030: 009b 083c 0000 0825 fcff a923
   adde 0a3c ...<...%...#...<
5 00000040: efbe 4a35 0000 0aad feff 0915
   0400 0821 ...J5.....!
6 00000050: 2700 0010 0000 0000 feff 0010
   0000 0000 ..'.....

Offset 1, 10 lines modified
1 00000000: bada baab 0200 0000 5845 0000
   0000 009a .....XE.....
2 00000010: 1200 0000 9c5e 0000 4b1d 0000
   4045 0000 .....^..K...@E..
3 00000020: 0000 009a 1719 e252 009b 1d3c
   0020 bd27 .....R...<..'
4 00000030: 009b 083c 0000 0825 fcff a923
   adde 0a3c ...<...%...#...<
5 00000040: efbe 4a35 0000 0aad feff 0915
   0400 0821 ...J5.....!
6 00000050: 2700 0010 0000 0000 feff 0010
   0000 0000 ..'.....
```

1. All dependencies should be locked to specific version
 1. Via language package manager
 2. Or via “vendoring” (by checking in dependencies)
2. Test your package manager reproducibility!
(e.g. PHP composer randomizes autoloader)

Reproducibility support in build tools

BUILD TOOLS

1. Some build tools will help you achieve reproducibility. Use them!
2. Or apply the previous guidelines to your favorite build system. Yes, it's possible to have reproducible build with GNU Make.

Tup “the updater”

<http://gittup.org/tup/>

1. Limits and monitors file dependencies via
FUSE
2. Resets and controls use of environment
variables

Bazel “{Fast, Correct} – Chose two” (Google)

<https://bazel.build/>

1. Executes most build actions in a sandbox
2. Prevents use of `__TIME__`, `__DATE__`, etc.
3. Rewrites source paths in built objects
4. Normalizes outputs
5. Controls environment variables
6. Prevents use of global dependencies during compilation
7. Heavily focused on Google’s way of doing things

Buck “A high-performance build tool” (Facebook)

<https://buckbuild.com/>

1. Similar to Bazel
2. More flexible
3. Normalizes outputs
4. Allows control of environment variables

Pants “A fast, scalable build system” (Twitter)

<https://www.pantsbuild.org/>

1. Similar to Bazel
2. Written in Python (and a bit of Rust) rather than Java (Bazel, Buck)
3. Mostly useful for Java, Scala, Python, JS
4. Normalizes outputs
5. Controls environment variables

Please “cross-language build system with an emphasis on high performance, extensibility and reproducibility” (ThoughtMachine)

<https://please.build/>

1. Similar to Bazel and Buck
2. Written in Go rather than Java
3. Mostly useful for C++, Go, Java, Python
4. Normalizes outputs
5. Optional sandbox for build rules and tests

Meson “extremely fast and user friendly”

<http://mesonbuild.com/>

1. Frontend to Ninja, Visual Studio or Xcode
2. No wildcard support for sources
3. Reproducibility issues are routinely fixed
4. Used by systemd
5. A number of library definitions is provided by authors (<https://github.com/mesonbuild>)

1. Autotools background? → check out Meson
2. Use plain Makefiles? → try Tup
3. See what Bazel has to offer
4. Bazel is too restrictive? → try Buck
5. Not a fan of Java in a build tool? → try
Pants or Please

How to define and distribute a build environment

BUILD ENVIRONMENTS

1. Build tools and versions
2. Build system architecture
3. Operating system
4. Build path
5. User name
6. Locale & timezone
7. ...

1. Build from source
2. Use reference distribution
3. Define as a VM / container

1. Use external resources (Coreboot)
 - Use guidelines for remote files
2. Check in everything (*BSD, Bazel)
3. Ship the toolchain as a build product
(OpenWRT)

1. Use a stable distribution
(Debian, CentOS, etc.)
2. Record package versions
3. Hope the old versions stay available or mirror them
4. Used by Bitcoin, Tor Browser

1. Gitian
2. Docker
3. Vagrant

1. Used by Bitcoin, Tor Browser
2. Runs builds using LXC or KVM
3. “Descriptors” that describe the build using:
 1. Base distribution
 2. Packages
 3. GIT remotes
 4. Other input files
 5. Build script
4. <https://gitian.org>

1. Provides a way to describe Linux application container images
2. Build in a controlled environment
3. Bazel has support for building Docker images reproducibly
4. <https://www.docker.com>

1. Spins up and drives VirtualBox VM using Ruby and other scripts
2. Build in a controlled environment
3. Can be used under Windows and macOS
4. <https://www.vagrantup.com>

How to compare build results and test for reproducibility

VERIFYING REPRODUCIBILITY

1. Use direct comparison or cryptographic checksums
2. Ignore / strip embedded signatures

1. Build the same source two times with changes to the environment
2. Compare results using the comparison protocol
3. Example variations: goo.gl/jgvisg

Tools to make builds reproducible

TOOLS

1. Normalizes archive file formats
(gzip, zip, jar, ar, etc.)
2. Extensible

1. Overlay FUSE filesystem that introduces non-determinism into FS metadata (e.g. randomizes traversal order)
2. Reveals problems with unstable order of inputs

1. “Deep recursive diff”
2. Recursively unpacks file formats and compares their contents
3. <https://try.diffoscope.org>
4. Presents comprehensive report

```

51427INSERT INTO "targets" VALUES('www.talkoclock.
com',13609);
51428INSERT INTO "targets" VALUES('talky.io',13610);
51429INSERT INTO "targets" VALUES('www.talky.io',
13610);
51430INSERT INTO "targets" VALUES('ico.ee',13611);
51431INSERT INTO "targets" VALUES('www.ico.ee',13611);
51432INSERT INTO "targets" VALUES('ttu.ee',13611);
51433[.9300.lines.removed.]
60733CREATE TABLE git_commit
60734.....(git_commit TEXT);
INSERT INTO "git_commit"
60735VALUES('cd09fb8c2161a8d1280b848eaab3b14d35fe3044')
;
60736COMMIT;

51434INSERT INTO "targets" VALUES('www.talkoclock.
com',13540);
51435INSERT INTO "targets" VALUES('talky.io',13541);
51436INSERT INTO "targets" VALUES('www.talky.io',
13541);
51437INSERT INTO "targets" VALUES('ico.ee',13542);
51438INSERT INTO "targets" VALUES('www.ico.ee',13542);
51439INSERT INTO "targets" VALUES('ttu.ee',13542);
51440[.9314.lines.removed.]
60754CREATE TABLE git_commit
60755.....(git_commit TEXT);
INSERT INTO "git_commit"
60756VALUES('e78fe5d803208bf6c877dc675cdb4f1b719e7519')
;
60757COMMIT;

```

install.rdf

Offset 5, 15 lines modified

```

5     ....<Description about="urn:mozilla:install-
manifest">
6     .....<em:name>HTTPS-Everywhere</em:name>
7     .....<em:creator>Mike Perry, Peter Eckersley,
&amp; Yan Zhu</em:creator>
8     .....<em:aboutURL>chrome://https-everywhere/
content/about.xul</em:aboutURL>
9     .....<em:id>https-everywhere@eff.org</em:id>
10    .....<em:type>2</em:type> <!-- type: Extension -->
.....<em:description>Encrypt the Web!
Automatically use HTTPS security on many sites.</
em:description>
12    .....<em:version>5.0.6</em:version>
13    .....<em:multiprocessCompatible>true</em:
multiprocessCompatible>

```

Offset 5, 15 lines modified


```

5     ....<Description about="urn:mozilla:install-
manifest">
6     .....<em:name>HTTPS-Everywhere</em:name>
7     .....<em:creator>Mike Perry, Peter Eckersley,
&amp; Yan Zhu</em:creator>
8     .....<em:aboutURL>chrome://https-everywhere/
content/about.xul</em:aboutURL>
9     .....<em:id>https-everywhere@eff.org</em:id>
10    .....<em:type>2</em:type> <!-- type: Extension -->
.....<em:description>Encrypt the Web!
Automatically use HTTPS security on many sites.</
em:description>
12    .....<em:version>5.0.7</em:version>
13    .....<em:multiprocessCompatible>true</em:
multiprocessCompatible>

```

```
a1b06a4f19250080d177560a0f1ea260 677160 text optional wswiss_20131206-5_all.deb
aspell-de_20131206-5_all.deb
├── metadata
│   ├── @@ -1,3 +1,3 @@
│   ├── rw-r--r-- 0/0      4 Jun 11 16:19 2014 debian-binary
│   ├── -rw-r--r-- 0/0    2893 Jun 11 16:19 2014 control.tar.gz
│   ├── -rw-r--r-- 0/0  329600 Jun 11 16:19 2014 data.tar.xz
│   ├── +rw-r--r-- 0/0    2875 Jun 11 16:19 2014 control.tar.gz
│   └── +rw-r--r-- 0/0  329596 Jun 11 16:19 2014 data.tar.xz
├── control.tar.gz
│   ├── control.tar
│   │   ├── md5sums
│   │   └── Files in package differs
│   └──
├──
├── data.tar.xz
│   ├── data.tar
│   │   ├── ./usr/lib/aspell/de_affix.dat
│   │   ├── @@ -1,11 +1,11 @@
│   │   ├── # this is the affix file of the de_DE Myspell dictionary
│   │   ├── # derived from the igerman98 dictionary
│   │   ├── #
│   │   ├── -# Version: 20131206 (build 20150801)
│   │   ├── +# Version: 20131206 (build 20150802)
│   │   ├── #
│   │   ├── # Copyright (C) 1998-2011 Bjoern Jacke <bjoern@j3e.de>
│   │   ├── #
│   │   ├── # License: GPLv2, GPLv3 or OASIS distribution license agreement
│   │   ├── # There should be a copy of all of this licenses included
│   │   ├── # with every distribution of this dictionary. Modified
│   │   ├── # versions using the GPL may only include the GPL
│   │   └── ./usr/share/aspell/de-common.cwl.gz
│   │       ├── metadata
│   │       ├── @@ -1 +1 @@
│   │       ├── -gzip compressed data, last modified: Sat Aug  1 18:21:34 2015, max compression, from Unix
│   │       └── +gzip compressed data, last modified: Sat Aug  1 18:24:36 2015, max compression, from Unix
```

1. LD_PRELOAD of libfaketime
2. Makes a subject believe current time is the specified one or starts at the specified point
3. May cause issues with certain build tools that use timestamps for decision making (such as Make)

1. For Windows binaries 
2. Post-processes PEs and PDBs to make their builds reproducible

How to make system package builds reproducible

PACKAGING

1. Two parts:
 1. Build package contents reproducibly
 2. Package it reproducibly
2. deb: pbuilder, cowbuilder, sbuild
 - It's possible to post-process .deb as an archive
3. rpm: mock

Where to go from here?

FINAL THOUGHTS

REPRODUCIBLE

- “Anyone can repeat it”
- Given the same:
 1. Input data
 2. Relevant attributes of environment
- Produces bit-by-bit identical outputs

REPEATABLE

- “It produces same results”
- Given the same:
 1. Input data
 2. Runtime
- Produces bit-by-bit identical outputs

1. If working on closed source – probably not, repeatability & reliability should suffice
2. It takes considerable effort to implement
3. Requires continuous testing
4. Requires “rebuilders”

1. Smaller binary differences
2. Quality assurance
3. Increased development speed
4. Build documentation
5. Easier license compliance (GPL)

1. We build an advanced hosting platform
2. (Mostly) closed source
3. A lot of configurations
4. A lot of languages and technologies in use
5. Switch from make-like system to Buck
6. Focus on repeatability rather than reproducibility

1. Try building your project twice in different environment (or even the same one!) and compare results with diffoscope
2. <https://reproducible-builds.org>
Spearheaded by Debian community, includes numerous other projects
3. Provides continuous testing facilities at <https://tests.reproducible-builds.org>
4. If you have an Open Source project, write a helper script and submit it to be tested as well!

Questions?

THANK YOU