Making Data-Driven Porting Decisions with Tuscan

Kareem Khazem
Earl T. Barr
Petr Hosek

Software typically outlives the platform that it was written for. This necessitates regular 'porting' of software to new platforms, a tedious and ad-hoc process. We wrote a framework, **Tuscan**, for automatically testing software portability to new platforms en-masse. We ran Tuscan on 2,699 programs on four different platforms and found that the majority of programs are locked in to a single platform, preventing them from building on other platforms. We also wrote a build wrapper, **Red**, which provides insight into the reasons that builds failed, and automatically fixes some of the most common build failures to help us gather more data.

Background

Compiler & linker	gcc clang lld gold
C Standard Library	glibc musl bionic uClibc
Architecture	x86_64 AArch64 SPARC

Software relies on many components to build and run—a selection of these components is called a **platform**. Ideally, it should be possible to build a program on many platforms without modifying it.

Several software projects have had difficulty when porting their programs to new platforms. These projects discovered portability problems as they went along, and had to create time-consuming fixes on an ad-hoc basis, duplicating efforts by other projects.

Alpine Linux	**Android**	**FreeBSD**	**Fuchsia**
(using **musl** as the C standard library)	(userspace & Linux kernel now compiled with **Clang**)	(compiled with **Clang**, linked with **LLD**)	(using **musl**, **Clang**, and **LLD**)

These issues are of concern to several groups:

- **Software developers** should like to ensure that their programs are future-proof, i.e. able to be built with innovative new build toolchains.
- **Authors of new platforms** must carefully evaluate how much existing software will fail to build because it is locked in to existing platforms.
- **Projects migrating to new platforms** need to quantify the time cost of porting existing software.

Our Contributions

We built an infrastructure to test the platform portability of programs in the wild en-masse. In brief:

- We wrote **Tuscan**, an automated environment for conducting deterministic tests on program builds under a variety of platforms.
- We wrote **Red**, a build wrapper that provides deep insight into why a build on a foreign platform failed.
- We used Tuscan and Red to build thousands of programs on several platforms.

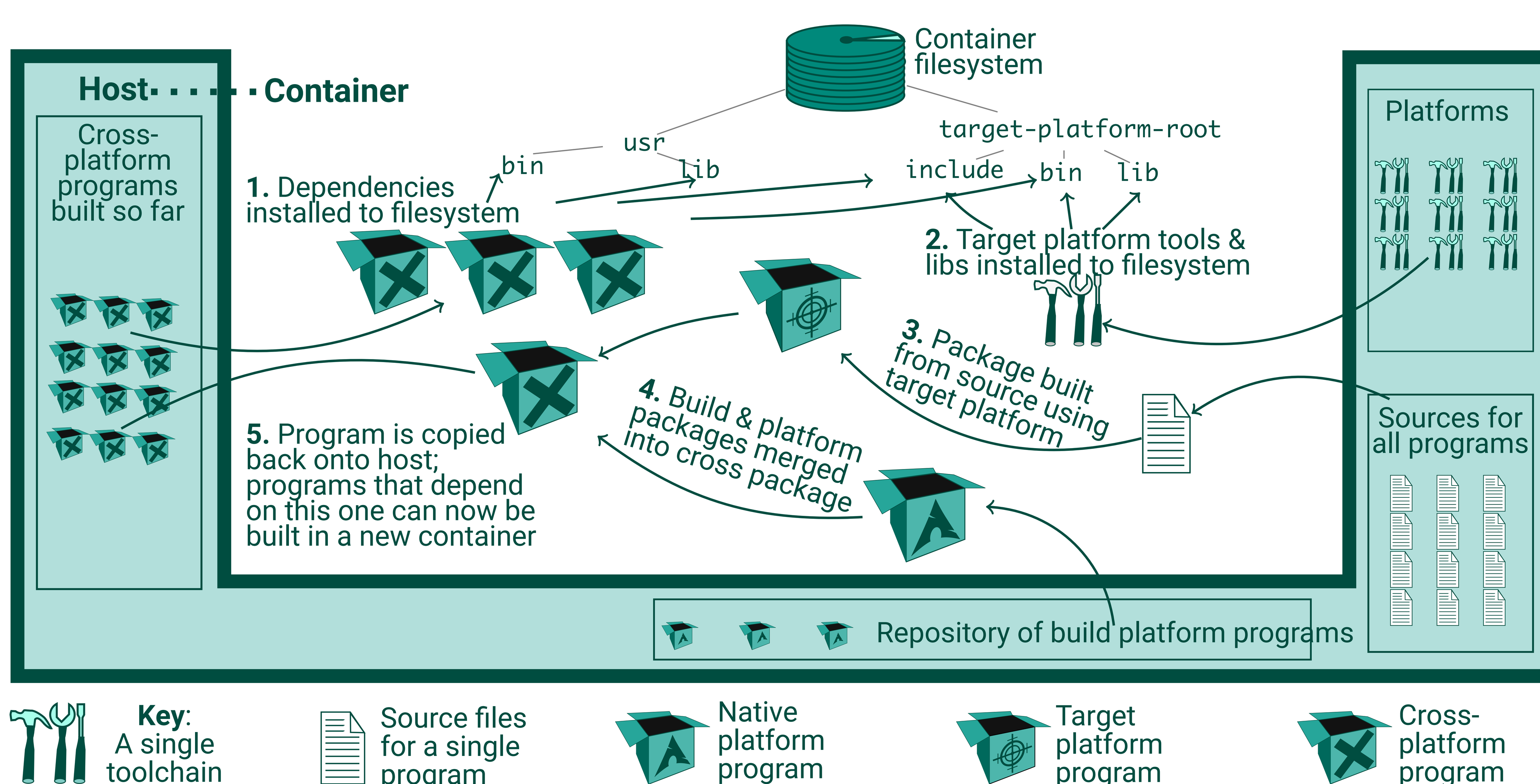
Tuscan

Tuscan solves several challenges that arise when rigorously testing cross-platform builds. It uses a Linux distribution to derive programs' dependency graphs and avoid the details of how to invoke the myriad of extant build systems. It uses containerisation to ensure that builds are isolated from each other and happen in a pristine environment. Tuscan exploits inter-program build parallelism to ensure that the tests scale while building individual programs serially to avoid non-determinism from scheduling and parallelism.

Red

Red provides diagnostic information on every sub-process spawned by a build, no matter what build tool is used. Red wraps around the build of a program whose portability we are testing, allowing us to understand the most common reasons for portability-related build failures. We used our catalogue of the most common build failures to teach Red how to *rescue* a failing build. Red can inject arbitrary code into a running process just before a subprocess spawn, allowing it to correct invocations that match a common pattern of build failure.

How It All Works



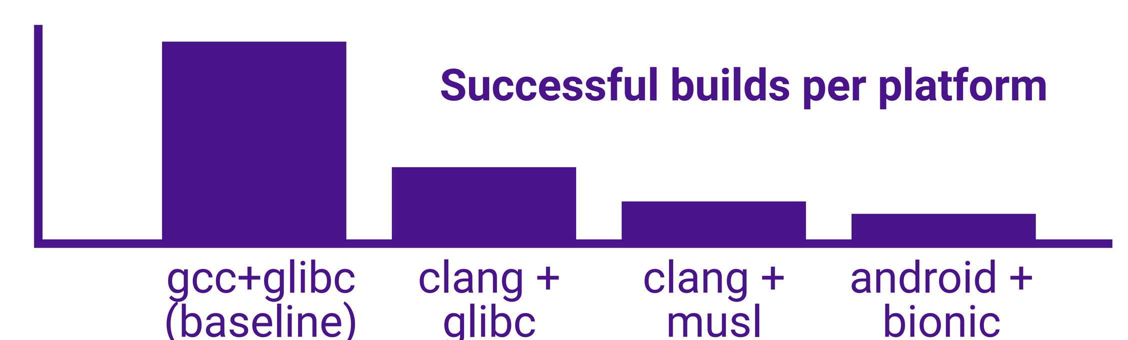
The diagram above shows how Tuscan builds a single program on a particular platform. On a single host computer (dark U-shaped region), multiple containers (white center region) will be running at once, building programs in reverse-dependency order. Once a program's dependencies have all been built and placed in the repository on the left, they are copied into the container (**1**). The compiler, C standard library, and other platform-specific tools are installed into the filesystem outside the usual file heirarchy (**2**). Tuscan then attempts to build the program using a platform that it was not originally intended to run on (the *target platform*, **3**). If the build succeeds, Tuscan merges the target platform-built program with the natively built program (**4**) so that any components of the program that need to run as build dependencies of other programs are able to run natively. This "cross-platform program" is then copied out of the container (**5**), at which point any programs that have a build-time dependency on that program can now be built, so the cycle continues. By wrapping each build with Red, we can understand why a build failed.

Results

We used Tuscan to build 2,699 programs on four platforms:

- the native platform (the **glibc** C library and **gcc** compiler);
- **glibc** C standard library and **clang** compiler;
- **musl** C standard library and **clang** compiler;
- **Bionic** C library and **Android gcc**, compiled to **Arm 32-bit**.

Many programs fail to build outside of their native platform:



There were several recurring causes of build failure:

Inclusion of non standards-conformat **header files** that don't exist on some platforms, but which could easily be replaced;

Undefined references to symbols inadvertantly exported by one implementation of the C standard library, but not others;

Hardcoded invocations to a particular implementations of compilers or other build tools;

and many others. Our results web page contains full details of all of the programs whose builds we tested, across all platforms, and gives detailed insight into the build process and reasons for build failure. We have identified concrete issues that platform authors and software developers can address, while providing a framework that can be used in the future to ensure that software remains portable to new platforms as they are developed.

Google UCL

https://karkhaz.github.io/tuscan