Making Data-Driven Porting Decisions with Tuscan

Kareem Khazem
University College London
London, UK
karkhaz@karkhaz.com

Earl T. Barr
University College London
London, UK
e.barr@ucl.ac.uk

Petr Hosek
Google
Mountain View, CA, USA
phosek@google.com

ABSTRACT

Software typically outlives the platform that it was originally written for. To smooth the transition to new tools and platforms, programs should depend on the underlying platform as little as possible. In practice, however, software build processes are highly sensitive to their build platform, notably the implementation of the compiler and standard library. This makes it difficult to port existing, mature software to emerging platforms—web based runtimes like WebAssembly, resource-constrained environments for Internet-of-Things devices, or innovative new operating systems like Fuchsia.

We present Tuscan, a framework for conducting automatic, deterministic, reproducible tests on build systems. Tuscan is the first framework to solve the problem of reproducibly testing builds cross-platform at massive scale. We also wrote a build wrapper, Red, which hijacks builds to tolerate common failures that arise from platform dependence, allowing the test harness to discover errors later in the build. Authors of innovative platforms can use Tuscan and Red to test the extent of unportability in the software ecosystem, and to quantify the effort necessary to port legacy software.

We evaluated Tuscan by building an operating system distribution, consisting of 2,699 Red-wrapped programs, on four platforms, yielding a 'catalog' of the most common portability errors. This catalog informs data-driven porting decisions and motivates changes to programs, build systems, and language standards; systematically quantifies problems that platform writers have hitherto discovered only on an ad-hoc basis; and forms the basis for a common substrate of portability fixes that developers can apply to their software.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Compilers*; *Reusability*; *Maintaining software*; Software evolution;

KEYWORDS

build systems, portability, toolchains

ACM Reference Format:
Kareem Khazem, Earl T. Barr, and Petr Hosek. 2018. Making Data-Driven Porting Decisions with Tuscan. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213855

1 INTRODUCTION

Under relentless social and technological pressure, our computing platforms have evolved rapidly. Nevertheless, the adoption of tomorrow's platforms is hindered by the assumptions of today: programmers bake dependencies into their code, making porting programs needlessly painful. The 32-bit timestamp-porting efforts in the Linux kernel [Bergmann 2015; Corbet 2017] and glibc [Aribaud 2015] attest to these assumptions, as do the examples below. In this paper, we concentrate on measuring a program's dependence on its build platform—consisting of the compiler, standard libraries, and other components that we define in Section 2. We focus on the portability of programs, not the correctness of the program after it has been built for an alternative platform. This is because *(1)* the portability of a program is prerequisite to correctly running the program on multiple platforms, and *(2)* we wish to gain insight into the state of portability of programs in the wild, at massive scale, without restricting ourselves to per-program correctness checks. We are motivated by the problem of porting programs to platforms where these components differ from the 'usual' implementations out of necessity, as in the following examples.

WebAssembly is a bytecode that executes inside a browser sandbox. It heralds the combination of two thus-far distinct worlds: desktops and embedded systems. WebAssembly is intended as a platform to execute fully-fledged desktop applications, yet, for enhanced security and portability, its execution environment and build toolchain resemble those used for embedded programs. This combination violates assumptions that desktop developers have about the platform. For example, WebAssembly programs are linked to a minimal implementation of the C Standard Library (libc). One such minimal libc is musl [Felker 2011], which differs from the more extensive GNU C library despite being POSIX-compliant [Musl 2017]. These differences make porting codebases difficult, as with the LibreSSL project's porting experience [Devs 2014].

Fuchsia, a new operating system being developed at Google, breaks platform-specific assumptions in a different way. Fuchsia uses a microkernel that departs from several POSIX assumptions, like "everything is a file", so many existing applications will fail to build or exhibit changes at runtime. The toolchain Fuchsia uses also differs from the usual GNU/Linux toolchain, replacing GCC with Clang. The issues caused by switching away from the 'traditional' toolchain are well-known anecdotally, as with the FreeBSD community's effort [Davis 2012; FreeBSD 2017; Maste 2016] to switch to a Clang and LLD-based build, or the effort to compile Linux with Clang [Edge 2017; Hines et al. 2017; Webster et al. 2014].

Even 'traditional' platforms, like Linux distributions, grapple with the redundance and maintenance cost of discovering and fixing portability problems in an ad-hoc manner. Although standards by interoperability bodies such as freedesktop.org have made it easier to write cross-platform software, most Linux distributions separately maintain additional patches to fix portability issues in the software that they distribute. To evaluate the state of program portability in the wild, we had to seek out distributions that explicitly refuse to maintain such patches, as we detail in Section 3.1.

These examples demonstrate that discovering porting problems in a haphazard fashion, and fixing them as they arise, is not scalable and duplicates effort across projects. To solve this problem, we created Tuscan, a framework for testing portability issues that scales up to thousands of programs and works fully automatically and deterministically. Platform writers, application developers, and researchers can use Tuscan to catalog the most common portability issues across a wide range of software.

Tuscan builds programs and their dependencies, compiling them to be run on a platform of the user's choice. Software in the wild is often written in an unportable manner, meaning it is desirable to *(a)* understand why a program failed to build on a particular platform, and *(b)* automatically intervene to tolerate a broken build, proceeding with compilation while noting that it would have failed. While we do not aim to have a general solution for automatically tolerating arbitrarily broken builds, we found that partially addressing problem *(b)* allowed us to proceed further through a build before failing, often even succeeding at a previously-failing build. This increased the diversity of build failures that we were able to catalog.

To catalog build errors (problem *a*), we wrote Red, a build wrapper. Red monitors process spawns of a build to report on exactly why that build failed. Through our tests of a large corpus of software using Tuscan and Red, we gathered a detailed catalog[1] of the most common reasons for builds to fail—described in Section 4.3.

To automatically tolerate builds (problem *b*), Red can hijack the build process and fix errors that match a pattern in the catalog. Red helps to gather more data about build failures on different platforms, because when Red 'rescues' a program's build, Tuscan is then able to build all of that program's dependencies. When evaluating Tuscan and Red, we used a corpus of software that, by design, was configured to have a sparse dependency graph—again helping us to learn about a large variety of build failures, since a failure to build a single program did not prevent many other programs from building. We elaborate on these design considerations in Section 3.

Contributions.

(1) We define and solve the *cross-platform namespace collision problem*, a prerequisite to building programs for arbitrary platforms *en-masse*, in Section 2.
(2) We implement Red, a tool for *(a)* monitoring and logging the build of a program, and *(b)* automatically repairing program builds that are about to fail in one of several common failure modes. On the platform on which we observed the most build failures, Red's intervention caused 48% of failing builds to succeed.
(3) We present Tuscan, a test harness for automatically building programs on arbitrary platforms. Tuscan realizes our solution to the cross-platform namespace collision problem, allowing developers and researchers to deterministically test build systems, and enabling them to make data-driven porting decisions.
(4) We evaluate our work on a large corpus of programs. We obtain a detailed catalog of portability problems in the wild. The catalog is a living document that can be updated when the software ecosystem is updated to address the pervasive portability issues that we found; it also informs and motivates the effort to fix these issues. The source code for Tuscan and Red is available,[2] as is the catalog and the program corpus that we use for our evaluation.[1]

[1]https://karkhaz.github.io/tuscan/

2 THE CROSS-PLATFORM NAMESPACE COLLISION PROBLEM

We solve the problem of automatically building large numbers of programs on arbitrary platforms to expose incompatibilities between programs and the build infrastructure used on each platform. This has only been solved through significant ad-hoc, manual intervention. We present our solution, *cross-platform packages*, after formalizing several prerequisite concepts.

A *toolchain* is a sequence of programs that transforms a body of source code into a binary object and then into a running process. For C and C++ projects, the toolchain consists of (at least) a *preprocessor*, *compiler*, *linker*, *compiler runtime library*, and the *C and C++ standard libraries*, which build the source, and a *dynamic linker*, which loads shared libraries at runtime. Some projects also require other *binary utilities* (for manipulating object files). In this paper, we focus on the build process; runtime behavior is out-of-scope.

In most projects, a *build tool* runs the toolchain to build the codebase. A *meta-build tool* chooses which toolchain to use, and generates a manifest for the build tool to run; together, these tools form a *build system*. Although build systems are not part of the toolchain, they are a large source of incompatibilities between programs and toolchains. The *toolchain lock-in problem* is the dependency of a program on a single toolchain. Porting locked-in programs to newer toolchains requires costly manual intervention.

A *build platform* is the environment that a build executes in; it comprises an *architecture* and a toolchain. A *target platform* is the platform where a program will be installed. Programs are built on a particular platform for execution on a target platform. Usually, the build and target platforms are one and the same; our focus is cross-platform builds where they differ. A change in one of the components of the platform can cause a program to fail to build or to run incorrectly, even when the program itself does not change. Ideally, source code should build independent of the underlying platform; Section 4 investigates the extent to which this holds.

Programs do not exist in isolation; they typically depend on other programs and libraries, both to run and to be built from source. At the same time, a single body of source code, when built, can yield multiple related programs and libraries. The interactions between these two notions—dependencies and multiple build outputs—are captured by the *software packages* that operating system distributions provide to their users.

A *package* contains all installable components that a build system generates from a body of source code for a single platform. Installable components include programs, libraries, header files, and configuration data. Inside the package, these components are arranged in a file hierarchy that mirrors the filesystem where the package will be installed. A package P also contains metadata that specifies what other packages P depends on—either to build P from source or to run the components that P installs. A package's target platform determines its contents. For example, the programs and

[2]https://github.com/karkhaz/tuscan

libraries comprise object code that runs on the target platform's architecture. A package's header files may also differ, due to the evaluation of preprocessor macros that are conditioned on the target platform. Concretely, packages for platforms conforming to the Filesystem Hierarchy Standard [FHS Group 2004] install programs to `/usr/bin`, libraries to `/usr/lib`, and headers to `/usr/include`. The layout of a package's file hierarchy must match the platform that it is to be installed on, not the platform where it was built.

Typically, packages are built for the same platform they target; this is called a *native build*. For example, the Debian operating system distribution is built for several platforms; the Debian project provides hardware to developers so they can build their packages natively [Debian 2017]. In some cases, however, the build platform cannot be the same as the target platform. This may be because the target platform is unavailable to the developers doing the build or because the target platform is so resource-constrained that it is impractical to build software on it. Building a package on one platform for a different target platform is called *cross-compiling*. Build tools such as Autoconf [Vaughan et al. 2000] support cross-compilation through specifying the `--build`, `--host` and `--target` flags. Cross-compilation subtly differs from a native build. For example, Autoconf still attempts to compile test programs for the target platform, but it does not attempt to run them, since the machine code for the target platform will typically not run on the build platform. Unfortunately, build tools, such as Autoconf, rarely explicitly identify every external program that a successful build needs to run: builds may need to invoke a compiler, file and text manipulation programs (*e.g.* `sed`, `awk`), and other tools. This matters because builds rely on the ability to run other programs from dependent packages and those programs must run on the *build* platform; at the same time, a cross-compile build must link against dependent libraries that have been built for the *target* platform. We formalize these constraints in the following definition.

Definition 1 (Cross-Platform Namespace Collision Problem). *Let P, Q be (not necessarily distinct) packages whose builds depend on another package, D. Without loss of generality, let the build of P need to execute a program provided by D, and let the build of Q need to link against a library provided by D. To build P, programs provided by D and built for the* build *platform must be installed on the filesystem. To build Q, libraries provided by D—but this time built for the* target *platform—must be installed on the filesystem. The* cross-platform namespace collision problem *is the namespace collision that arises when the same artifacts, built for two different platforms, must be installed on the same filesystem to proceed with a build.*

Developers can work around the cross-platform namespace collision problem for a single package by manually placing the libraries and programs in the locations that the package expects. Building a program natively on the target hardware or in an emulator for the target platform so that dependencies for the target platform can be run, is another common workaround. This latter strategy is used by Debian developers when building the entire distribution from scratch for platforms where the hardware is hard to come by. However, this relies on the availability of requisite hardware or emulation environment for the target platform which is not always the case (*e.g.* when building for WebAssembly). We solve the cross-platform namespace collision problem through a generalization of software packages. Our *cross-platform packages* allow the simultaneous, non-conflicting installation of both build and target software components, making cross-platform builds possible without any manual intervention.

A *cross-platform package* is one that segregates the binaries and libraries of multiple build platforms from each other and from the headers and libraries of target platforms into disjoint file hierarchies. A cross-platform package superimposes the builds of multiple platforms. Concretely, the package contains binaries and libraries for one or more *build* platforms $B_1, \ldots, B_n$, as well as libraries and headers for one or more *target* platforms $T_1, \ldots, T_n$. Rather than being installed under the default `/usr` directory, the components for each platform are each installed in a custom directory beside `/usr`. Namely, the binaries and libraries for each build platform are installed in /B_1/`bin` and /B_1/`lib`, …, /B_n/`bin` and /B_n/`lib`; and the libraries and headers for each target platform are installed in /T_1/`include` and /T_1/`lib`, …, /T_n/`include` and /T_n/`lib`. The target-platform libraries and headers are thus available for linking and inclusion when building a dependent package for that platform.

Toolchain components can typically be built with a flag, usually `--with-sysroot`, which specifies an alternatively-rooted file hierarchy (as opposed to the default `/usr`). This means that toolchains can be trivially configured to be aware of cross-platform packages, without modifying the toolchain components' source code.

3 DESIGN AND IMPLEMENTATION

We implemented Tuscan to test the portability of thousands of programs for different target platforms, detailed in Section 4. To fix the build problems Tuscan uncovers, we implemented Red, a build environment that automatically fixes build problems that match common patterns that Tuscan discovers. Tuscan creates a highly isolated build environment; calculates the program dependency graph; and invokes a Red-wrapped build of each program in its own copy of the build environment, in topological order of the dependency graph. The design of Tuscan and Red solve several concrete challenges, which we summarize next.

3.1 Challenges and Solutions

Scale and Automation. A useful technique for performing automated builds on massive scale is to use the infrastructure provided by operating system distributions, *c.f.* Kroening and Tautschnig's work (Section 5). Distributions provide a unified interface for invoking the plethora of build systems. One can run this interface for each of the programs in the distribution, rather than manually issuing the build incantations—ensuring scalability and determinism of the overall process. We therefore used a distribution as a source of programs and build metadata for our evaluation.

When running build experiments, one has a choice between running at massive scale—investigating platform-dependence of the software ecosystem on the whole—or building only a few programs and then ensuring that the build was correct, by invoking `make check` if the program supplies a test suite. Tuscan can operate both ways—it can check that built packages run correctly by invoking a test script after the build finishes. On the large program corpus we used for this work, only 16% of programs supplied even rudimentary tests for build correctness. Nevertheless, this work focuses on program portability rather than build correctness; we chose to scale rather than manually verifying the sanity of a small number of builds.

Isolation and Reproducibility. It is vital that TUSCAN emits the same results on multiple runs over an identical program corpus. Our results in Section 4.3 motivate improvements to programs and the toolchains used to build them; TUSCAN's determinism gives us confidence that a different result on *updated* programs is really due to a change in those programs' portability. However, subtle changes in the environment between one build and another can affect whether a build succeeds, as highlighted by Mytkowicz et al. [2009]. Furthermore, machine configurations can cause results to vary between users: different libraries being installed, scheduling differences due to number of CPU cores, and different toolchain versions can all cause a program build to differ. Finally, the artifacts left behind from *previous* builds can change what gets built.

We overcome these challenges by using Docker containers as an isolated build environment—TUSCAN creates a new container for each build, ensuring a clean initial state. Containers do not 'save' their state when they are torn down, so a subsequent run of TUSCAN proceeds in exactly the same way as any previous run. Each container's initial state is minimal, containing only the environment variables and programs that a build needs. Using a distribution means that we leverage that distrubution's build system: distributions configure the packages they ship to build without any user intervention, and contribute to the effort to ensure that packages build reproducibly [Cascadian et al. 2014]. The fact that TUSCAN builds packages without user intervention eliminates another source of nondeterminism. Finally, TUSCAN allocates a single CPU core to each container and adds `-j1` to builds' `$MAKEFLAGS` to avoid nondeterminism from scheduling and ensure that builds proceed serially.

Dependency Domino Effect. Building a large number of programs is a partially serial process, because some programs require that other programs have been built before their build can begin. These dependencies form the edges of a forest whose nodes are program builds. The builds at each node of the forest depend on the transitive closure of their children being successful; a 'domino effect' occurs when a single build fails, since every program that transitively depends it cannot then be built. An operating system distribution's *dependency factor* is the average size of the transitive closure of a program's children; higher dependency factors lead to a more severe domino effect when a single program fails to build. Distributions that supply a fully-fledged desktop environment with a large number of pre-installed programs (like Debian, Fedora, or Ubuntu) tend to have a high dependency factor. Distributions that are intended to be configured from the ground up (like Arch Linux, Alpine Linux, or Gentoo) have low dependency factors. Using such a distribution leads to a larger diversity of build errors, as a larger variety of programs can be built. This is because user-facing applications tend to be at the roots of the forest, while libraries are nearer the leaves; build failures higher up the tree therefore prevent a disproportionate amount of dependent user-facing programs from building.

Vanilla Builds. Our aim is to measure the platform dependence of programs in the wild. Distributions can frustrate this goal: to support multiple platforms, they manually patch the programs that they distribute. A manual search[3] through Alpine Linux finds 150 patches specifically to fix dependency on the GNU C Library (rather than the MUSL C library used by Alpine), while distributions that support a large number of architectures (like Debian or Gentoo) redundantly and separately maintain similar patches. In contrast, Arch Linux "ships software as released by the original developers" [wiki 2016] and is only built for the `x86_64` architecture with a typical GNU toolchain (described in Section 4.2). We therefore chose Arch Linux as our source of packages and build metadata. The Docker containers that TUSCAN uses are also based on Arch, which does not limit the platforms that TUSCAN can run on.

3.2 Individual Package Build Process

TUSCAN's main task is to build packages using a non-default target platform. We describe the process of building a single package in this section; the process is illustrated in Figure 1.

TUSCAN creates a fresh Docker container for the build of each package. Packages built by TUSCAN are stored on the user's filesystem (outside of the container that built them), so that any packages that depend on that package are able to use it later. In Figure 1, the user's filesystem is the dark gray region. It contains a repository of all the packages that have been built so far; these will be cross-platform packages, as described in Section 2. The container used for building a single package is the white region in the center; data is copied into and out of the container from the user's filesystem.

If the cross-platform package repository contains all the dependencies of a package P, then TUSCAN starts building P. Referring to Figure 1, TUSCAN performs the following steps:

(1) copies and installs the dependencies of P into the container;
(2) installs the target toolchain binaries, headers and libraries;
(3) builds a target-platform package with the target toolchain;
(4) generates a cross-platform package by combining the target-platform package with the build-platform package;
(5) copies the cross-platform package out of the container.

At **step (1)**, as well as the build-platform components that are installed into `/usr/bin` and `/usr/lib`, the cross-package dependencies will install target-platform libraries and headers into the `lib` and `include` directories—as mentioned in the remark on cross-platform file layout (Section 2)—so that P can link against those libraries. The hybrid package contains target-platform specific versions of tools and libraries; the build-platform versions of these tools and libraries may already be installed in the container. At **step (2)**, TUSCAN therefore installs the target-platform toolchain in a location on the filesystem that is disjoint from the build-platform components. It is a build error if P's build system attempts to invoke the build toolchain, as opposed to the target toolchain that we wish to build with. Therefore, TUSCAN leaves the build toolchain installed in its usual location and uses RED to redirect invocations to the target toolchain (see Section 3.4).

In **step (3)**, TUSCAN copies P's sources into the container and builds them with the target toolchain. The user will have configured the appropriate environment variables (`$CC`, `$LD_FLAGS` etc.) so that—if the package uses a well-behaved build system—it will build using the target toolchain installed in **step (2)** and link against

[3] `git clone git://git.alpinelinux.org/aports && find aports -name "musl*.patch"`

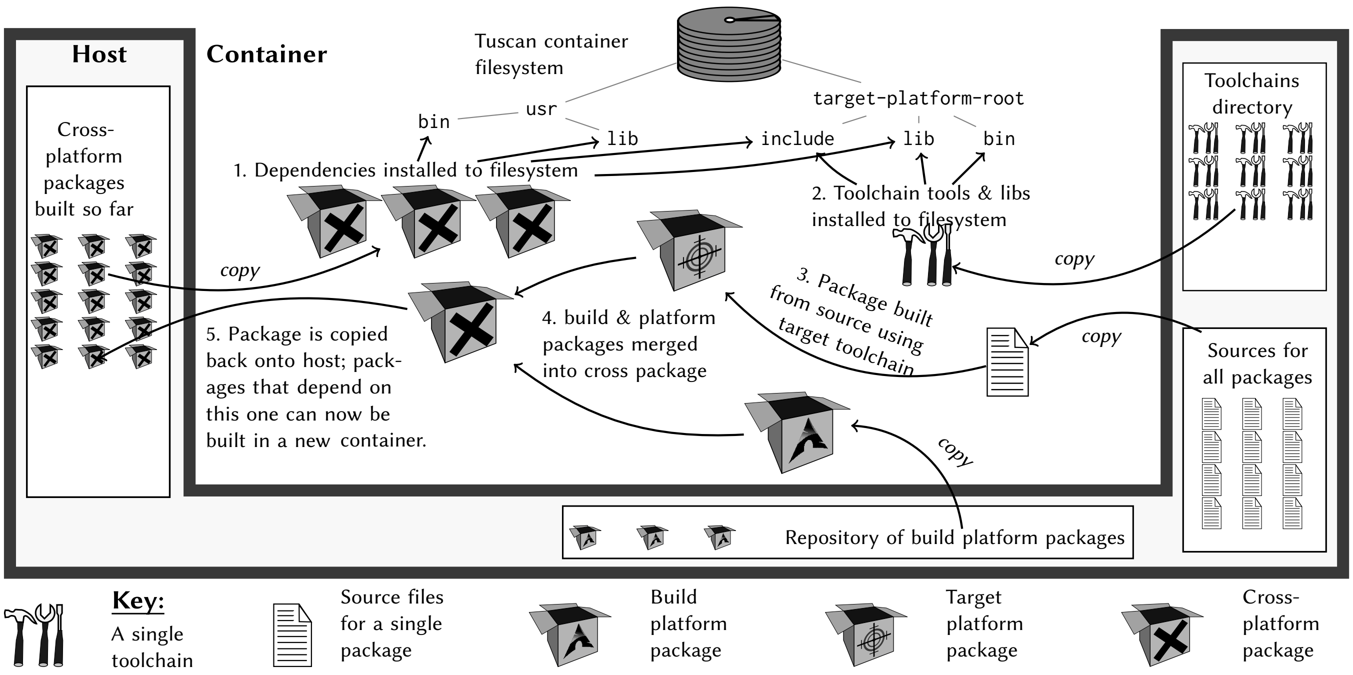
Host
Container
Tuscan container filesystem
Cross-platform packages built so far
bin
usr
lib
target-platform-root
include
lib
bin
Toolchains directory
1. Dependencies installed to filesystem
2. Toolchain tools & libs installed to filesystem
copy
copy
3. Package built from source using target toolchain
4. build & platform packages merged into cross package
5. Package is copied back onto host; packages that depend on this one can now be built in a new container.
copy
Sources for all packages
copy
Repository of build platform packages
Key:
A single toolchain
Source files for a single package
Build platform package
Target platform package
Cross-platform package

Figure 1: How a single package is built, as described in Section 3.2. Tuscan runs on a single *host computer* (the gray-bordered *U*-shape) and uses a *container* (the central region) to build the package. The host computer permanently stores the source code and packages for all the packages that we want to build, as well as the toolchains that we build with; these files are copied into a container whenever they are needed. The result of running a container is a cross-platform package, which is copied out of the container onto the host—Tuscan then builds any packages that depend on that package, and the cycle starts again. Several containers may be running in parallel all on the same host; this diagram shows what happens inside a single container.

the target-platform libraries installed in **step (1)**. Many packages do not in fact have well-behaved build systems or are unportable. Wrapping the build with Red can mitigate some of these issues, but in many cases, the build fails. If *P* fails to build, Tuscan halts the container; builds that transitively depend on *P* will not be run.

In **step (4)**, if a target platform package of *P* was successfully built in **step (3)**, Tuscan combines it with the build-platform version of *P* to create a cross-platform package. The repository of build-platform packages will have been downloaded at the same time as the sources, as noted in Section 3.3. The cross-platform package thus consists of two sets of files built from the same source: some built with the build toolchain, and others built with the target toolchain.

In **step (5)**, Tuscan adds *P* to the repository of cross-platform packages built so far. Tuscan also copies the build logs and data from Red out of the container, ready for data analysis.

3.3 Inputs, Reproducibility, and Extensibility

To aid reproducibility of our results, Tuscan takes the following four components as input. The *source repository* is a set of source codebases, along with metadata describing the dependencies between those codebases. The *build platform repository* is a set of packages that were built from the source repository to be run on the build platform (i.e. the build and target platforms are the same). The *container* is a Docker image of an environment that allows for building one of the source packages from the source repository. The *toolchain script* is a script that builds a toolchain into a container.

The first three items must be generated at the same time, because not all software is forward-compatible. Tuscan allows both reproduction and repeatability of our results by supporting two options for obtaining these four components: downloading them afresh, or using previously-generated components. By default, Tuscan downloads the latest version of the first two items (currently about 290 GB of data) and generate a matching Docker image. It also downloads and builds the toolchain that the user specified. This means that the study can be repeated in the future with updated toolchains and programs. Tuscan caches data that were previously downloaded so a run of Tuscan on a particular set of components can be repeated. Our results can be reproduced by dropping the code, images and toolchain that we used for this paper (or any other run of Tuscan) into Tuscan's top-level directory.

Tuscan is extensible and can be used for many different build-related experiments: varying the compiler, C Standard Libary implementation, and architectures as we do for our evaluation (Section 4) is just one possibility. Since Tuscan takes a *toolchain script* as input, one can use Tuscan to measure the effects of varying different aspects of the build toolchain: using static *vs.* dynamic linking, comparing POSIX to non-POSIX toolchains, or experimenting with building for exotic architectures.

3.4 Red—the Redirected Execution Dæmon

We implemented a stand-alone tool called Red to monitor and automatically fix problems with the build of a single package. Red wraps the invocation of a build process (*e.g.* `make` or Arch Linux's makepkg) and produces detailed diagnostics: the tree of processes spawned by the build, the environment variables set at each of those process spawns, and the invocation of key system calls.

Using Red to monitor builds revealed that there are several common categories of build failures that arise from unportability; we detail these categories in Figure 6. We thus built Red to modify, as well as merely monitor, the build process. Red modifies the build by shadowing key system calls and C standard library functions with custom implementations, *e.g.* the functions that are used to spawn new processes on POSIX (the `exec` (3) and `posix_spawn` (3) function families). Red injects these implementations into the build process using the `$LD_PRELOAD` environment variable. This functionality means that, by modifying incorrect build tool invocations, we can sidestep known failures and continue running the build—while logging that the build would have failed without Red's intervention. This allows us to collect more data on the extent of program unportability in the wild. Since Red intercepts C standard library calls that are invoked by all programs, no matter what language they are written in, Red's mitigations are agnostic of the build system Red is wrapping. Examples of mitigations that Red can apply include (1) correcting build-influencing environment variables like `$CC` and `$PATH` just before a process spawn; (2) checking that files that are opened with `fopen()` actually exist, and synthesizing stubs for them if they do not; and (3) rewriting process spawns to ensure that the build system invokes the *target* platform build tools rather than the *build* platform build tools. By applying these mitigations, Red completely rescued some builds and allowed others to run for longer before they failed (Figure 6).

Red is easily extensible with new mitigations. This means that platform writers and application developers can continue to test mitigations to new portability issues as they arise. A Red mitigation is simply a function, written in C, that detects when a standard library function is being invoked in a way that will break the build—and then corrects that invocation so that the build can proceed successfully. Together, Red and Tuscan engender a cycle of discovery and mitigation of issues that block porting of programs to new platforms. By running Red-wrapped builds of thousands of programs in Tuscan, developers can discover the most pressing build issues that prevent the most programs from being easily ported; implement mitigations for these issues in Red; and then test the builds again, discovering new porting issues that manifest further down the line. This data-driven approach allows the developers of innovative new platforms—like the examples we noted in Section 1—to discover, in advance, the obstacles to the adoption of their platform. By systematically testing the portability of the software ecosystem, platform authors can plan on writing compatibility 'shims' to allow legacy software to build on new platforms, or quantify how much work would be needed to resolve pervasive portability issues.

Red is derived from BEAR [Nagy 2016], which is used to generate compilation databases. BEAR uses `$LD_PRELOAD` to intercept spawns of new processes so that it can print out their names. Red extends this functionality by detecting patterns of incorrect invocations and repairing them as well as logging their names.

4 EVALUATION

In the previous section, we described the design of Tuscan—a framework for testing toolchains by building programs *en-masse.* We tested several toolchains, described in Section 4.2, with Tuscan, on several thousand packages (described in Section 4.1).

Scope. We evaluate portability issues by varying the toolchain across several axes: C standard library, compiler, and architecture. By changing only a single factor each time we run the experiment, we are able to pinpoint which build issues are caused by particular parts of the toolchain. We do not vary other parts of the toolchain, and our evaluation focuses on POSIX toolchains only—though Tuscan works for any toolchain whose source code is available.

Tuscan can check whether a build yields a valid binary by calling `make check` or similar (Section 3.1). However, only 16.1% of the programs in our corpus have such a test suite. One can therefore either evaluate build errors at scale or in detail. This evaluation focuses on scale; we are the first to catalog build errors on a cross-platform build of an operating system distribution at this scale.

4.1 Corpus

At time of writing, Arch Linux distributes 9,851 packages for the x86_64 architecture. We do not attempt to build all of these packages, since many of them were involved in convoluted circular dependency chains. Circular dependencies are an intrinsic aspect of building software—see the comment about "toolchain build order" on the Arch Linux build manifest for glibc [McRae 2017] noting that glibc needs to be built in order to build itself. Tuscan implements automatic dependency resolution, and does not attempt to build packages that are involved in a circular dependency. In practice, we found that the set of 2,699 packages that Tuscan does build seems fairly representative. Of these, 1,534 are of interest due to their containing C or C++ code; we built the remaining non-C/C++ packages to satisfy their dependencies. We were interested in C and C++ codebases as these languages are the most platform-dependent, and the build systems and configuration tools that those codebases use are correspondingly prone to containing errors. Statistics [Debian 2018] indicate that these languages are the most commonly used in a large operating system, so identifying and addressing portability issues in such codebases is especially important.

Figure 2 shows the sizes of the programs that we built in lines of code (LOC), in lieu of attempting to quantify the complexity of the programs' build systems. while metrics about build systems that are more refined than LOC counts do exist (*e.g.* the one presented by Martin and Cordy [2016]), the majority of the build scripts in the

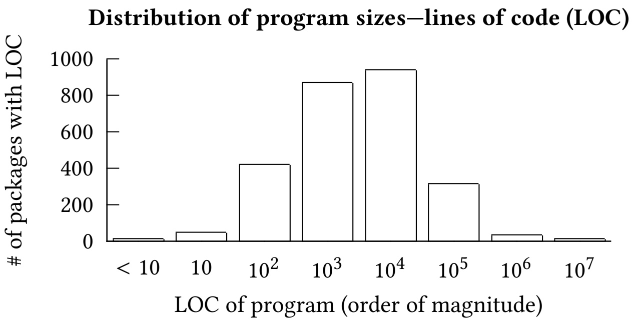

Figure 2: Most of the packages that we used for this study are thousands or tens of thousands of lines long.

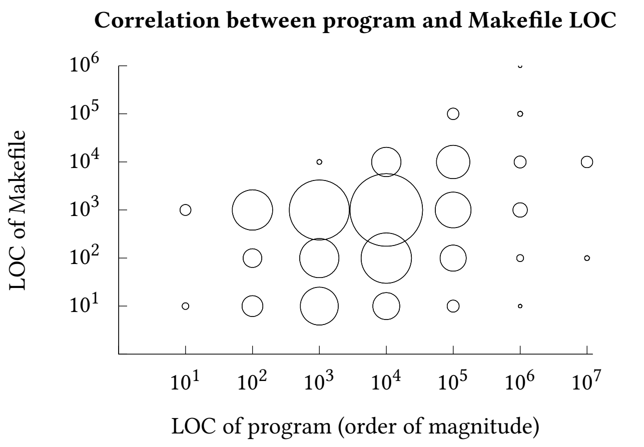

Figure 3: The area of a circle at coordinates x, y shows the *number* of programs having on the order of x lines of code, and which use Makefiles on the order of y lines long.

programs we studied were generated rather than hand-written—meaning that the complexity is correlated with the LOC count anyway, as noted in that study. The packages that we tried to build were written in 236 million lines[4] of C and C++ code, with 36 million lines written in other languages. For programs that used (or generated) Makefiles, we found that the number of lines of Makefile were indeed loosely correlated with the executable LOC (Figure 3). The data point at $(10^1, 10^3)$ is an artifact of verbose, machine-generated Makefiles; it represents 160 Perl libraries that each export a small function, but use a large Perl-specific Makefile.

About 60% of the programs used a `configure` script, or the `cmake` tool, as part of the build process. The majority of `configure` scripts were generated by the GNU Autoconf tool. These scripts are responsible for detecting the capabilities of the build environment, searching for required libraries and headers, and then correctly setting up the build process in light of these findings—so their functionality is relevant to this study. We remark on how well these tools handled the use of alternative toolchains in Section 4.3.

As well as building packages using the alternative platforms described in Section 4.2, we attempted to build all the packages on the *default* Arch Linux platform. This allows us to focus on build issues that occur with packages that *do* build with the default Arch platform, but *fail* to build with an alternative platform. In our presentation of results, we only discuss such alternative-platform broken packages, but we note here that we were surprised at the number of packages—273 of the 1,534 C/C++ packages—that do not even build successfully using the default platform. Since we can only build a package if all of its dependencies have built, these 273 failures meant that we are unable to attempt a build of a further 170 dependent packages—a total of 443 failures with the default platform. 140 of these failures are because the source code for the packages[5] was missing—including 87 that we wished to use for our experiment. Other failures were genuine build errors, which have been manually worked around by the package maintainers.

[4]All lines-of-code counts obtained using SLOCCount [Wheeler 2004].
[5]We reported these packages to the Arch Linux developers; they are listed at https://www.archlinux.org/todo/packages-with-missing-sources/.

4.2 Platforms Used for Building

We used four different platforms in our experiments. As a 'control' for our experiment, we use the same platform that Arch Linux—and most other GNU/Linux—developers use for building packages: this is the `x86_64` architecture, with a toolchain comprising GCC, glibc and libstdc++. Our second platform changes only the compiler (to Clang); this is a platform similar to the one used by projects such as LLVMLinux. The third platform we use builds on the second by also changing the C standard library to musl, and the C++ standard library to libc++. This is a representative of 'non-GNU' toolchains as used by WebAssembly or Fuchsia. Finally, we demonstrate Tuscan's ability to build programs for a different target architecture by using the Android toolchain—comprising of GCC, Bionic and libstdc++—and building for the ARM32 architecture rather than `x86_64`. Given the popularity of Android, this is one of the most widely used 'alternative' toolchains used on Linux-based systems.

4.3 Results

Build systems are fragile: they exacerbate porting difficulties, slowing the emergence of new platforms. This section quantifies the extent of the problem, showing Tuscan and Red's practical utility. Tuscan collects and classifies build-related problems cheaply; here, we present that data and how it can be used to resolve porting problems. Red automatically fixes some of the problems Tuscan identifies. We quantify how many problems Red fixes and leverage the Tuscan results to prioritize future extensions to Red. In this section, we present all our results relative to the GNU toolchain.

Few packages build successfully on a platform different from the baseline; Figure 4 shows the number of packages that successfully built on each platform. The platforms are arranged in increasing order of divergence from the baseline: the second platform changes the compiler, the third changes the compiler and standard C library, and the last platform additionally changes the architecture. Each deviation from the baseline causes an additional set of build failures. In this section, we compare the build of each package on the build platform and on an alternative target platform in order to discover the root causes of any portability issues. We therefore do not consider packages that fail to build on the build platform.

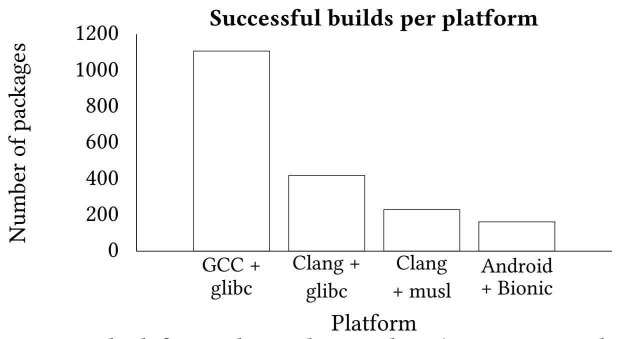

Figure 4: The leftmost bar indicates the C/C++ programs that built successfully on the baseline platform. The other bars show how many of those baseline-successful C/C++ packages built successfully on other platforms. When wrapping builds with Red (Section 3.4), Tuscan builds a much larger number of packages successfully; see Figure 6.

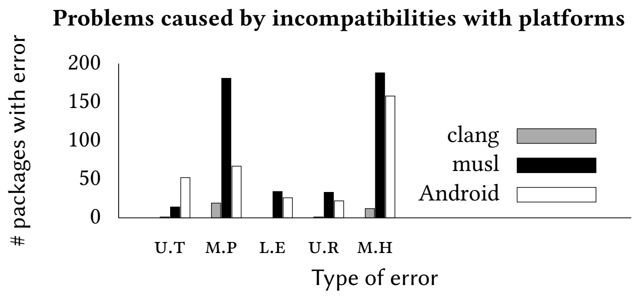

Figure 5: The bars indicate the number of packages that failed due to an error caused by incompatibility with a particular platform. The error types are described in the paragraphs below: "Unknown Type," "Malformed Path," "Linker Error," "Undefined Reference," and "Missing Header".

Given the low rate of successful builds, we decided to investigate what the most common causes of build breakage were. Figure 5 shows the five most common failures that broke the build, again across the 1107 baseline-successful C and C++ programs.

Missing header. Platform-specific headers are included by 351 packages. The build of such packages throws a 'missing header' error when they are built on alternative target platforms; this accounted for the majority of build errors. The largest culprits were the `gnu/stubs-32.h` and `libintl.h` files, which together were included 143 times; and various Linux-specific headers, such as `linux/types.h`, which was included 104 times. Tuscan thus provides valuable data on what platform-specific features developers rely on the most—in this case, architecture-specific typedefs for various widths of integers. Developers of new platforms can use these data to make empirically-driven decisions on what APIs to support, for example by providing a compatibility 'shim' that allows the many packages that include this header to build flawlessly on the new platform. The exact platform-specific headers that each of those 351 packages includes are detailed under the `missing_header` entries in our catalog[1].

Undefined references. This occurs when the code references symbols inadvertantly exposed by glibc. Code that relies on such references breaks when built on platforms that use a different libc; this problem did not occur on the Clang+glibc platform. The most common undefined symbol was `rpl_malloc`, which is a known bug in GNU Autoconf when building for non-glibc platforms [Dickey 2003]. The discussion in that reference indicates that effort to fix the bug was not forthcoming, but Tuscan provides a compelling case for developers of new platforms to step up to the task—doing so would encourage more seamless adoption of their platform.

Unknown type. This occurs when the code uses types inadvertently exposed by glibc; this failure caused 67 errors. For example, musl intentionally does not define integer types that are BSD extensions to the standard [Kreitman 2004] (*e.g.* `u_int32_t`, `u_int64_t` as opposed to the standard `uint32_t`, `uint64_t`). Standards committees have, in the past, legitimized *de facto* implementations by ratifying their interfaces in the official standard—examples include the C committee adding flexible array members and zero-length arrays to the C99 standard [ISO 1999], which had long been extensions to GCC. We do not take a stance on whether the C language standard should be extended to specify that conformant implementations must supply commonly-implemented typedefs. We note, however, that Tuscan supplies concrete data about the usage of these types in real-world codebases, informing decisions about what common extensions ought to be added to the next version of the standard.

Compiler error. There were only 5 compiler errors, all of which were caused by command line flags supported by GCC but unsupported by Clang (*e.g.* `-Wlogical-op`, `-Woverride-init`). This result highlights the mostly-successful effort to make Clang a drop-in replacement for GCC. Tuscan demonstrates its utility by providing Clang developers with the flags that Clang doesn't support and that are used in real codebases. Clang developers have been receptive to the results on our catalog[1], and we are working to patch Clang to maintain command-line parity with GCC.

Linker error. Linker errors account for 59 failures. The root cause of these errors is incorrectly-written Autoconf manifests; Autoconf generates incorrect configure scripts from these manifests, which then set the build tool up with the wrong include path. The difficulty of writing Autoconf manifests is anecdotally known, but our catalog concretely demonstrates that the most common consequence of hastily put-together manifests is configure scripts that only work accidentally, when the build and target platforms are the same.

Malformed path. There were 267 cases where the `$PATH` environment variable was incorrectly set when the build system forked a subprocess. Many build tools rewrite `$PATH` in order to include directories where language-specific tools live. For example, we observed 599 instances where build tools needing access to Perl programs rewrote the `$PATH` to include `/usr/bin/site_perl`, `/usr/bin/core_perl`, and similar directories. Such build tools do not respect values of the `$PATH` set by the user; therefore, it is not possible to port programs using these tools to platforms where custom `$PATH` needs to be specified.

The figures that we describe above relate to packages whose builds were broken. However, Red was able to intercept and repair several of these problems. Figure 6 shows how often Red recovered a build that would otherwise have failed without Red's intervention.

Frequently, builds under the Clang+glibc platform exhibited problems that Red was able to correct *and no other problems.* Thus, the black bars over 'native invocation,' 'malformed path,' and 'empty path' in Figure 6 are quite tall for Clang+glibc . These bars represent builds that *would* have broken under that platform if Red had not intervened. The black bars on Figure 4 on page 7 for Clang+glibc would have been twice as tall if we had included Red-rescued builds in that graph: there were 1,000 successful builds for Clang+glibc, and Figure 6 shows a total of 481 Red-recovered builds.

Red does not rescue as many complete builds for the other two target platforms. This is because they have a much higher rate of irrecoverable errors (such as the missing headers discussed previously). Red remains a useful tool for these platforms, as it allows the build to proceed further than it would have before breaking. This allows us to test more build problems than if the build had broken

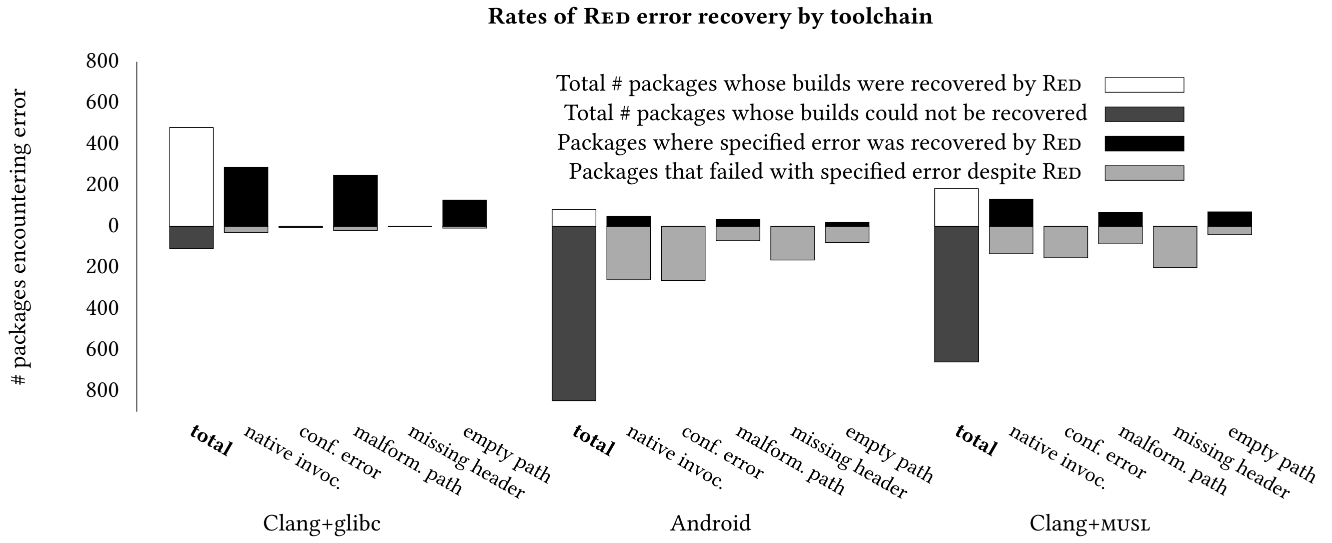

Figure 6: Several build issues were common over multiple toolchains. Red is able to salvage the build in many cases where the build makes an invocation to a native tool, or where the meta-build system mangles or completely empties the $PATH variable. The large number of *failing* builds that made native invocations on the Android and clang+musl toolchains failed *after* Red had corrected the native invocation—meaning that the build got further before failing, allowing us to collect more data on why builds break on those toolchains. The negative y-axis does not represent negative numbers of packages; rather, we use it to juxtapose successful (black) with unsuccessful (gray) builds.

due to a Red-recoverable error. In fact, 34% of build-breaking errors on Clang+musl happened *after* a Red-recovered error.

Red Limitations. There are many reasons why target platform incompatibilities cause builds to break. Red therefore needs to be taught about each type of build failure for it to rescue builds that encounter that failure mode. While we implemented support for rescuing the most common types of build failure that we observed, Figure 6 shows that there are a large number of failures for other reasons, especially when the Standard C Library implementation varies from the baseline.

5 RELATED WORK

Our work is related to efforts to model and empirically study build systems, and work that builds large corpuses of software at scale—either to analyze the software itself or to gain insight into the build.

Modeling and Studying Build Systems. There has been a growing realization in academia that configuration and build systems are worthy of study in their own right, either by actually running builds or by parsing and modeling build scripts symbolically. Nadi and Holt [2012] investigate the Linux kernel build system through the latter technique, parsing the kernel's build and configuration files and running that representation through a SAT solver to detect inconsistencies. They found that inconsistencies between the various parts of the build system do exist, prompting a further study by Nadi et al. [2013] on how these inconsistencies arise. Another work that parses and analyses build files is [Zhou et al. 2015], which extracts 'configuration knowledge' from Makefiles to determine under what conditions certain builds are possible. These approaches intrinsically provide precise information on problems or inconsistencies with the build system under scrutiny, since the approach is tailored to one particular build system. Our tool, Red, takes the opposite approach by actually requiring that the build be run. This means that Red can introspect on any build (controlled by Make, CMake or some other future build tool) and provide insight into why the build failed. Red's error-reporting precision rests on its knowledge of the toolchain rather than the build system, so its error catalog carries over to new build systems. We designed Red to be extensible with knowledge about new toolchains, so that it can be taught to detect and intercept new 'patterns' of build failures.

An example of a tool that *runs* the build to glean information about the build system is MAKAO [Adams et al. 2007b]. MAKAO produces visualizations of a build and includes a query language to search for events in the build log, e.g. errors, tool invocations and output file generation. This work is valuable to understand the build progress of a single codebase; Tuscan provides a framework for automatically running such analyses on thousands of codebases in a scalable way. The MAKAO authors use the `xtrace` tool to monitor spawned processes in the build. This is a more versatile approach than the static parsing work in the previous paragraph, as `xtrace` can wrap arbitrary build tools (not just Make). We also take this approach: Red wraps around any build invocation, and operates at a finer granularity than MAKAO, intercepting system calls as well as process spawns. This granularity allows Red to repair the build environment and redirect system calls during the build, which process monitoring tools like `xtrace` or `strace` cannot do.

Adams et al. [2007a] use MAKAO to measure the evolution of a single program (the Linux kernel), finding increases in complexity and also giving an account of the maintenance pressures that drive the evolution of the build system. McIntosh et al. [2011] also study maintenance effort on several large codebases. TUSCAN can help researchers with this kind of study by allowing build wrappers like MAKAO to reproducibly execute in an isolated environment. Kerzazi et al. [2014] address the question of why builds break both qualitatively and quantitatively, through interviews with developers and analyses on build failures. Their quantitative data is gleaned from 6 months' worth of build data of a web application, and they advance several hypotheses about what circumstances cause builds to break. An automated framework like TUSCAN can help to confirm these results in a reproducible way; TUSCAN can be run repeatedly on the same corpus of programs, or take new sets of software as input to measure how build breakages change over time.

Building Code at Scale. Kroening and Tautschnig [2014] inspired TUSCAN: they were the first to build an entire Linux distribution on an alternative toolchain. They ran a static analysis tool on every program in the Debian operating system, using Debian's package metadata to provide a uniform build interface. Their 'alternative toolchain' is the CBMC tool [Clarke et al. 2004], which emulates GCC's command-line interface. Their experiment also tackled platform portability, because they compile to `goto`-binaries rather than ELF object files. Following in their footsteps, TUSCAN uses a distribution to provide a uniform way of automatically building software.

TUSCAN solves two problems that Kroening and Tautschnig did not face: the cross-platform namespace collision problem and the toolchain lock-in problem. `goto-cc` can output binaries that contain *both* executable ELF data *and* `goto`-binary code, so they were able to install `goto`-binaries in the usual location on the filesystem. We have had to work with alternative architectures (such as Arm) with a different machine code format, making it impossible to combine build- and target-platform object code into the same binary. TUSCAN helps researchers to run similar large-scale experiments even when their analysis tools do not emit such a hybrid-format object. Kroening and Tautschnig did not face toolchain lock-in because `goto-cc` itself is a drop-in replacement for GCC, accepting the *same* command-line flags and input language. Faithfully emulating a compiler's flags and input language (a problem that TUSCAN also solves) consumed development time the CBMC authors could have profitably spent working on the static analysis tool itself.

We hope that by solving the cross-platform build and toolchain lock-in problems, TUSCAN will pave the way for the research community to perform large-scale static analyses, similar to Kroening's and Tautschnig's but without the engineering effort needed to make static analysis tools drop-in replacements for GCC. One such experiment would extend RED to permit the application of the Clang Static Analyzer [Zaks 2017] to a large corpus of existing software.

Atlidakis et al. [2016] perform both static and dynamic analysis to determine what *abstractions* and *frameworks* are being used by POSIX applications on the Android, Ubuntu, and macOS platforms. They find that POSIX is lacking in several frameworks that developers need, and that developers are thus writing to higher-level frameworks than POSIX. Our empirical results in Section 4.3 highlight the *consequences* of this divergence: packages that have to rely on platform-specific APIs become tied to that API, making it difficult to port those packages to new platforms and thus hindering those platforms' adoption. In contrast to TUSCAN, Atlidakis et al. only surveyed applications running on their *native* platform. Our work goes further by quantifying how programs' builds break when ported to *new* platforms.

Mytkowicz et al. [2009] investigate the effects of seemingly-innocuous changes in the environment on running programs; their experiments bear some similarities to ours, and we designed TUSCAN to elide many of the sources of non-determinism that they describe. They conducted their experiments on several different physical machines as well as a virtual machine, taking care to manually set up a minimal environment for their experiments. Such a minimal environment is inherent to the Docker containers that TUSCAN uses, and we feel that TUSCAN would be an ideal test bench for investigations like theirs. Mytkowicz et al. use the `$LD_PRELOAD` variable to instrument programs, as we do with RED.

Al-Kofahi et al. [2014] implement a tool, MkFault, for discovering build issues specifically arising from errors in Makefiles. MkFault's domain-specific knowledge about GNU Make means that it can track down the root cause of a Makefile-induced build failure to a high degree of accuracy. By comparison, RED is a more general purpose build wrapper. It can intercept the invocations of any build system; by necessity, this means that it does not discern the root cause of a build failure with the precision that MkFault can. MkFault also seems targeted at discovering errors in the build logic of a Makefile. By contrast, we focus on discovering errors that stem from assumptions about the target platform, and which would break the build even if the Makefile were written flawlessly.

6 CONCLUSION

With the growing importance of new toolchains and targets, software portability is more important than ever. To determine the current state of affairs, we built TUSCAN, a framework for conducting controlled, deterministic tests on build systems at scale and RED, a build wrapper to automatically identify and overcome problems that prevent software from building for alternative target platforms. Using these tools, we tested 1,551 C and C++ programs over three toolchains and found out that the current situation is far from perfect: the least-compatible toolchain successfully built only 15% of programs, while the most compatible toolchain managed 38%.

RED identified several root causes for build failures. By implementing automatically-applied mitigations for some of these root causes in RED, we were able to run our tests for longer, sometimes even completing the build successfully while logging what would have broke the build in RED's absence. While we wrote the mitigations manually, we believe there is scope for research into automatically-learned mitigations, *e.g.* using machine learning. Furthermore, RED is easily extensible by practitioners who wish to continue testing the portability of the software ecosystem. We hope that platform authors and software developers will take up the mantle of systematically testing and resolving portability issues, undertaking to ensure that legacy software is portable to innovative new platforms. In aid of this, TUSCAN, RED, the program corpus that we used for this study, our raw test results, and our catalog, can all be found online[1].

ACKNOWLEDGMENTS

We would like to thank the anonymous referees for their constructive comments, as well as Rich Felker, Sean Klein, Roland McGrath, and Mark Seaborn for their suggestions and feedback.

REFERENCES

Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. 2007a. The Evolution of the Linux Build System. *ECEASST* 8 (2007).

Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. 2007b. Design recovery and maintenance of build systems. In *23rd IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 114–123.

Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault Localization for Make-Based Build Crashes. In *ICSME '14*. 526–530.

Albert Aribaud. 2015. Y2038 Proofness Design. (2015). https://sourceware.org/glibc/wiki/Y2038ProofnessDesign Accessed: 2017-04-10.

Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 2016. POSIX abstractions in modern operating systems: the old, the new, and the missing. In *Proc. 11th European Conference on Computer Systems (EuroSys)*, Cristian Cadar, Peter Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues (Eds.). ACM, 19:1–19:17.

Arnd Bergmann. 2015. Linux kernel patchset: "converting system calls to 64-bit time_t"'. (2015). https://lwn.net/Articles/643407/

Vagrant Cascadian, Chris Lamb, Holger Levesen, and Lunar. 2014. Reproducible Builds. (2014). https://reproducible-builds.org

Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*. 168–176.

Jonathan Corbet. 2017. 2038: only 21 years away. (2017). https://lwn.net/Articles/717076/

Brooks Davis. 2012. Clang as default compiler November 4th. (2012). https://lists.freebsd.org/pipermail/freebsd-current/2012-September/036480.html

Debian 2017. Debian ARM ports project. (2017). https://www.debian.org/ports/arm

Debian 2018. Debian Statistics: Historical Lines of code. (2018). https://sources.debian.org/stats/#hist_sloc

Devs 2014. How compatible is LibreSSL. (2014). https://devsonacid.wordpress.com/2014/07/12/how-compatible-is-libressl/

Thomas E. Dickey. 2003. Re: Why is malloc being defined as `rpl_malloc` ?? (2003). http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20161212/412152.htm

Jake Edge. 2017. Building the kernel with Clang. (2017). https://lwn.net/Articles/734071/

Rich Felker. 2011. Musl libc. (2011). https://www.musl-libc.org

FHS Group 2004. Filesystem Heirarchy Standard. (2004). http://www.pathname.com/fhs/pub/fhs-2.3.pdf

FreeBSD 2017. Ports and Clang—Build Failures with Fixes. (2017). https://wiki.freebsd.org/PortsAndClang#Build_failures_with_fixes Accessed: 2017-04-10.

Stephen Hines, Nick Desaulniers, and Greg Hackmann. 2017. Compiling Android userspace and Linux kernel with LLVM. In *LLVM Developers' Meeting, San Jose, CA, USA*. https://www.youtube.com/watch?v=6l4DtR5exwo

ISO 1999. International Standard ISO/IEC 9899:1999—Programming Languages—C. (1999). https://www.iso.org/standard/29237.html

Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why Do Automated Builds Break? An Empirical Study. In *30th IEEE International Conference on Software Maintenance and Evolution*. 41–50.

Struart Kreitman. 2004. `u_int32_t` vs `uint32_t`. (2004). https://lists.freedesktop.org/archives/release-wranglers/2004-August/000923.html

Daniel Kroening and Michael Tautschnig. 2014. Automating Software Analysis at Large Scale. In *MEMICS '14*. 30–39.

Douglas H. Martin and James R. Cordy. 2016. On the maintenance complexity of makefiles. In *Proc. 7th International Workshop on Emerging Trends in Software Metrics (WETSoM@ICSE)*. ACM, 50–56.

Ed Maste. 2016. Linking the FreeBSD base system with LLD – status update. (2016). http://lists.llvm.org/pipermail/llvm-dev/2016-March/096449.html

Shane McIntosh, Bram Adams, Thanh H. D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. 2011. An empirical study of build maintenance effort. In *Proc. 33rd International Conference on Software Engineering (ICSE)*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 141–150.

Allan McRae. 2017. Glibc PKGBUILD manifest for Arch Linux. (2017). https://git.archlinux.org/svntogit/packages.git/tree/trunk/PKGBUILD?h=packages/glibc

Musl 2017. Functional differences from glibc. (2017). wiki.musl-libc.org/wiki/Functional_differences_from_glibc Accessed: 2017-04-10.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing wrong data without doing anything obviously wrong!. In *ASPLOS '09*.

Sarah Nadi, Christian Dietrich, Reinhard Tartler, Richard C. Holt, and Daniel Lohmann. 2013. Linux variability anomalies: what causes them and how do they get fixed?. In *Proc. 10th Working Conference on Mining Software Repositories (MSR)*, Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim (Eds.). IEEE Computer Society, 111–120.

Sarah Nadi and Richard C. Holt. 2012. Mining Kbuild to Detect Variability Anomalies in Linux. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, Tom Mens, Anthony Cleve, and Rudolf Ferenc (Eds.). IEEE Computer Society, 107–116.

Làszlò Nagy. 2016. Build EAR. (2016). https://github.com/rizsotto/Bear

Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. 2000. *GNU Autoconf, Automake, and Libtool*. Sams Publishing. https://www.sourceware.org/autobook/autobook/autobook.html

Behan Webster, Jan-Simon Möller, Vinícius Tinti, and other contributors. 2014. LlvmLinux. (2014). https://llvm.linuxfoundation.org/

David A. Wheeler. 2004. SLOCCount. (2004). http://www.dwheeler.com/sloccount/

Arch Linux wiki. 2016. Arch Linux (article on the official wiki). (2016). https://wiki.archlinux.org/index.php/Arch_Linux

Anna Zaks. 2017. Clang Static Analyzer. (2017). https://clang-analyzer.llvm.org

Shurui Zhou, Jafar M. Al-Kofahi, Tien N. Nguyen, Christian Kästner, and Sarah Nadi. 2015. Extracting Configuration Knowledge from Build Files with Symbolic Analysis. In *3rd IEEE/ACM International Workshop on Release Engineering (RELENG)*, Bram Adams, Stephany Bellomo, Christian Bird, Foutse Khomh, and Kim Moir (Eds.). IEEE Computer Society, 20–23.