Wholly!: A Build System For The Modern Software Stack

Loic Gelle[1]*, Hassen Saidi[2], and Ashish Gehani[2]

[1] Ecole Polytechnique, France
[2] SRI International, USA

Abstract. *Wholly!* is an automated build system for the modern software stack. It is designed for reproducible and verifiable builds of optimized and debloated software that runs uniformly on traditional desktops, the cloud, and IoT devices. *Wholly!* uses Linux containers to ensure the integrity and reproducibility of the build environment. It uses the `clang` compiler to generate LLVM bitcode for all produced libraries and binaries to allow for whole program analysis, specialization, and optimization. The `clang` compiler and install tools are all built with *Wholly!* as well. *Wholly!* has been applied to build Alpine Linux, Docker containers, microservices, and IoT software. We show that software packages built in *Wholly!* are faster, smaller, and more amenable to whole program analysis.

1 Introduction

The modern software stack has evolved from desktops to monolithic servers, and finally to the cloud and IoT devices. Driven by the need for rapid scalability, developers can either deploy an application on a thin operating system layer such as OSv [14], an operating system designed for the cloud, or in a container such as Docker [3] running on a traditional operating system. Applications can also be broken up into a number of microservices. Microservices can run in individual distinct containers, in virtual machines (VMs), as unikernels [10, 16, 8] running on a hypervisor or on bare metal, or even as a single function as a service. The diversity of these computing platforms creates new challenges for formal verification. We are particularly interested in applying source code analysis such as *abstract interpretation* [2] and *predicate abstraction* [6]. When analyzing source code, it is necessary to understand the build process for different platforms, and to account for all used libraries so that whole program analysis is possible. This requires dealing with the following challenges:

Bloated software: Running a simple application in a container often involves building a container image that includes a number of libraries that are not necessary to run the application. For microservices, the bloatware is worse. Running a single Javascript function on Amazon Lambda or a service like Standard Library [18] requires running the entire Node.js interpreter on top of an Ubuntu image that contains libraries that are not needed for running Node.js. On a typical desktop running a Linux distribution, hundreds of packages and thousands of possibly extraneous libraries are installed.

* While visiting SRI.

Complex build processes and dependencies: `gcc` is the most popular compiler on Linux platforms. Even when using `clang`, `gcc` libraries are used during compilation. One has to carefully trace the build process to know precisely which supporting libraries, which build scripts, which linker, and which install tools have been used during any build process.

Lack of reproducibility: Given the complexity of build processes, and the reliance on dynamic linking and bloated deployment environments, it is often impossible to guarantee that software built in two different builds is going to behave exactly the same. This is particularly important in domains such as scientific computing [15], where reproducing the same results may be paramount to the integrity of the scientific method.

We present *Wholly!* [20], a tool for building and packaging software that explicitly defines dependencies, produces build processes that are repeatable and verifiable, and allows for whole program analysis. The result is leaner, faster, and debloated software packages that can be deployed on a variety of platforms. *Wholly!* is a tool for building and releasing software packages with C/C++ code. `gcc` is the de-facto standard compiler on Linux systems today. However, *Wholly!* uses the `clang` compiler to generate LLVM bitcode. *Wholly!* uses `musl-libc`, an efficient implementation of `libc` to produce leaner packages. `clang` has been used to build entire operating systems, such as FreeBSD and macOS. `musl-libc` is used to build Alpine Linux [1] with `gcc`. *Wholly!* is the first project to combine both `clang` and `musl-libc` to build Alpine and Docker containers.

We use a `clang` compiler that was built in *Wholly!* producing a faster and leaner compiler. The produced LLVM bitcode is used to further optimize the code using partial evaluation [17] and software winnowing [11], and for applying formal verification techniques [7]. In this paper, we describe the design, philosophy, evaluation, and applications of *Wholly!*, explaining how it enables whole program analysis of large and complex software.

2 System Overview

Each package is described in *Wholly!* by a simple recipe that contains all the information needed to build the package. Builds are performed in Docker containers to control the environment and provide isolation. For each package, build products are organized in fine-grained packages such as libraries, binaries, headers, and runtime support. *Wholly!* eventually releases the fine-grained sub-packages in the form of Docker images [3] that are easily reusable as dependencies to build more complex packages, or as production software. *Wholly!* uses static linking to produce small packages. Sub-packages can either be used as containers, or can be used on any Linux platform, making them highly portable across the different flavors of the Linux operating system.

2.1 Package Description

Two files are needed to describe a *Wholly!* package: a *recipe* file that is used for building the package, and a *contents* file that is used for releasing it. The *recipe* file describes what happens at build time. It is a small YAML-formatted file that contains the:
- link to download the source code for the package,
- name of the other *Wholly!* sub-packages that are build dependencies,

– invocations that need to be run in order to build the package, and
– for some packages, the path to additional resource files – patches for example – that will be used during the build stage.

Figure 1 shows a *Wholly!* package recipe for the `sqlite` database, version `3.18`.

```
release_date: 2017-07-18
variables:
  - pkg_name: sqlite
  - pkg_ver: '3.18.0'

# Dependencies
dependencies:
  musl-libc:
    - headers
    - libs
  readline-7.0:
    - headers

# Source
source:
 http://www.sqlite.org/2017/{pkg_name}-autoconf-3180000.tar.gz

# Build stage
build:
  - CC=gclang
    CFLAGS="-static"
    ./configure --prefix={__INSTALL_DIR__} --enable-shared=no
  - make
  - make install
```

Fig. 1. Recipe file for the package `sqlite-3.18`.

Figure 2 shows the *contents* file for the package. The contents file also uses the YAML format. It splits the package into different sub-packages, each of which is described by the:
– list of files that compose the sub-package, and
– checksum that is used to check the integrity of the build and to refine dependency management.

2.2 Building

Each package is built separately in a Docker container. This guarantees isolation and control over the:
– system files that are present during the build,
– environment variables being set,
– compiler and tools being used, and
– dependencies that are brought in.

Docker is an open platform to build, ship, and run distributed applications on desktops, data center VMs, or the cloud. Docker uses the resource isolation features of the Linux kernel, such as `cgroups` and kernel `namespaces`, and a union-capable file system, such as `OverlayFS`, to allow independent *containers* to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines. Docker can build images automatically by reading the instructions from a *Dockerfile*. A Dockerfile is a text document that contains all the commands a user would issue at

```
bc:
    checksum: sha256:4f9170f7c2cb4f701dac9826509ed14de8a0aeb1597d36bbeb499dd8bdbee00c
    files:
    - /usr/bc
bin:
    checksum: sha256:557f6deeeafc8dc3bff27f8ed97ef65ed15b5060f6e6f1f646e190db96573eb0
    files:
    - /usr/bin/sqlite3
headers:
    checksum: sha256:0663e892fbb7fbcfb1f9402de9ff8efdaddb09c6730cfba88218273967880c38
    files:
    - /usr/include/sqlite3.h
    - /usr/include/sqlite3ext.h
libs:
    checksum: sha256:e99ccebd1c087ee4137fffe6cf5efb04cbb78b14a9460da60fdd2c7ce8038328
    files:
    - /usr/lib/libsqlite3.a
```

Fig. 2. Contents file for the package `sqlite-3.18`.

a shell command line to assemble an image. The execution of each command defines a layer in the file system that is cached by Docker. Each layer is referred to by a random identifier that can be used to "pull" only that specific layer of the file system.

In order to build the desired package, *Wholly!* turns the recipe file into a Dockerfile that will be read and executed by the Docker Engine. Each build follows this pattern:

1. A Docker container is launched and populated with a base image that contains the elements common to all the builds – compiler, linker, and environment.
2. Files from the dependency sub-packages are copied into the container.
3. Source code for the package to build is downloaded.
4. Build commands in the recipe file are executed.

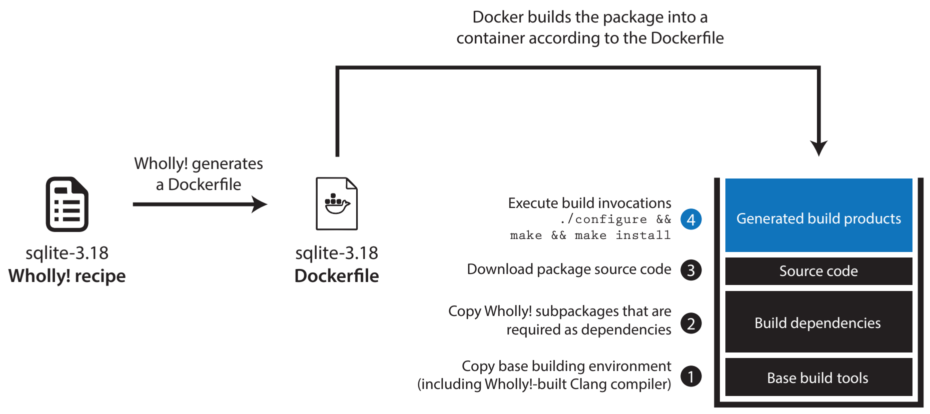
Docker builds the package into a container according to the Dockerfile
Wholly! generates a Dockerfile
sqlite-3.18
Wholly! recipe
sqlite-3.18
Dockerfile
Execute build invocations
./configure && make && make install
4
Download package source code
3
Copy Wholly! subpackages that are required as dependencies
2
Copy base building environment (including Wholly!-built Clang compiler)
1
Generated build products
Source code
Build dependencies
Base build tools

Fig. 3. Schematic view of the build stage for the *Wholly!* `sqlite-3.18` package.

At the end of this process, the Docker image contains, among other artifacts, the files from the desired package that have been built and installed. Using Docker's layered file system, we can determine the exact list of files that have been created during the build process. From these files, *Wholly!* creates a contents file that lists the files contained in each produced sub-package.

2.3 Sub-packaging

The image that is obtained from the previous step contains a lot of extraneous files, since it reflects the final state of the entire build environment. Starting from that, interesting files from the package are copied into different sub-packages, based on the specification in the contents file. The sub-package `sqlite-3.18-bin`, for example, only contains the file `/usr/bin/sqlite3` that has been built.

After all the sub-packages have been released in the form of new Docker images, *Wholly!* verifies that the checksums of these images are consistent with the ones provided in the contents file. This ensures the integrity of the build, while at the same time serving as a proof of reproducibility. Initially, the checksum is written to the contents file when the subpackage is first created. Fine-grained sub-packaging also enables fine-grained dependency management. The package `sqlite-3.18`, for example, only requires the headers from `musl-libc` and `readline-7.0` and the libraries from `musl-libc` instead of the whole contents of these packages.

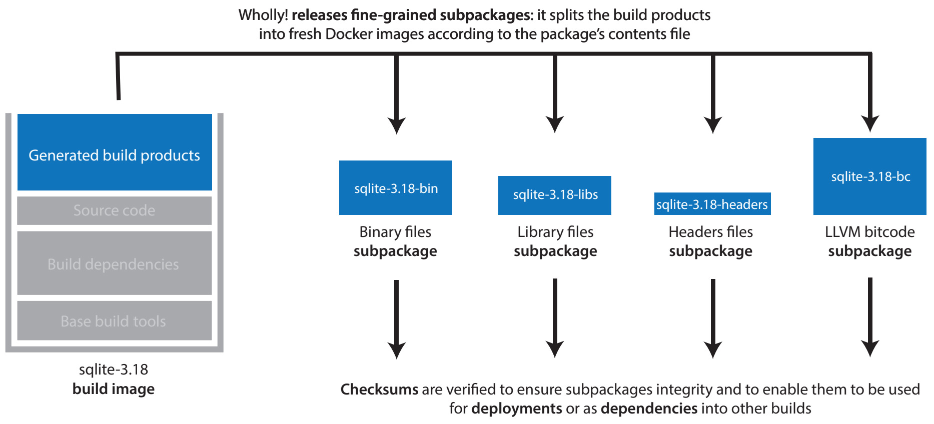
Wholly! **releases fine-grained subpackages:** it splits the build products into fresh Docker images according to the package's contents file
Generated build products
Source code
Build dependencies
Base build tools
sqlite-3.18 **build image**
sqlite-3.18-bin
Binary files **subpackage**
sqlite-3.18-libs
Library files **subpackage**
sqlite-3.18-headers
Headers files **subpackage**
sqlite-3.18-bc
LLVM bitcode **subpackage**
Checksums are verified to ensure subpackages integrity and to enable them to be used for **deployments** or as **dependencies** into other builds

Fig. 4. Schematic view of the release stage for the *Wholly!* `sqlite-3.18` package.

3 Design

In this section, we provide an in-depth description of the components and the design motivations of our build mechanism. In particular, we describe how *Wholly!* ensures traceability and reproducibility of builds.

3.1 Traceability of Builds

Wholly! uses recipes that are as simple and small as possible. They are designed to be understood easily, yet they contain every element that is necessary to the build. In particular, the dependencies are clearly stated, and since we want to keep the build environment as minimal as possible, only the required sub-packages are imported at compile time. *Wholly!* then transforms this recipe into a Dockerfile that will be used to launch the build container. Since Dockerfiles are becoming increasingly complex as new features are implemented in Docker, we chose a simple YAML format, completely independent from Docker, for our recipes.

```
FROM wholly-readline-7.0-headers as wholly-readline-7.0-headers-files
FROM wholly-musl-libc-headers as wholly-musl-libc-headers-files
FROM wholly-musl-libc-libs as wholly-musl-libc-libs-files
FROM wholly-base-image

# Bringing dependencies in
COPY --from=wholly-readline-7.0-headers-files / /
COPY --from=wholly-musl-libc-headers-files / /
COPY --from=wholly-musl-libc-libs-files / /

# Getting source
WORKDIR /build
RUN curl \
    "http://www.sqlite.org/2017/sqlite-autoconf-3180000.tar.gz" \
    -o src.tar.gz
RUN mkdir sqlite-3.18 && tar xf src.tar.gz -C sqlite-3.18 \
    --strip-components 1

# Building
WORKDIR /build/sqlite-3.18
RUN WLLVM_CONFIGURE_ONLY=1 CC=gclang CFLAGS="-static" \
    ./configure --prefix=/usr --enable-shared=no
RUN make
RUN make install
```

Fig. 5. Dockerfile generated by *Wholly!* to build `sqlite-3.18`.

Figure 5 describes the Dockerfile generated automatically by *Wholly!* from the recipe file described in Figure 1. The `FROM` command in the Dockerfile is used to import a previously built Docker image to build a more complex one. It allows users to control what goes into their Docker image. This is extremely important when controlling how packages are built. On a desktop, a build script would often look for dependencies in multiple locations, and will use whatever it can find in the file system. Tracing a build therefore often involves monitoring every build step for file accesses, for instance. Similarly, at runtime the execution of software will depend on the version of the dynamically linked libraries that are installed on the particular system. This makes it impossible to ensure that the software will execute uniformly across platforms. We control exactly what goes into a Docker image when we build a package. This allows us to trace the provenance of every single build product.

The contents file also greatly improves the clarity and the traceability of the builds. It can be used to account for every single file that is present in a released sub-package, and then in a container that runs in production.

3.2 Systematic Production of LLVM Bitcode

Wholly! uses `clang` as its C/C++ compiler. `clang` is capable of generating LLVM bitcode, an intermediate representation that is platform-independent and can be used for program transformation and optimization.

In order to benefit from this feature, *Wholly!* uses *gllvm* [5], a fast and concurrent wrapper for `clang` that generates both native objects and LLVM bitcode files. During a build, the bitcode for the whole package can easily be produced by calling the wrapper `gclang` instead of `clang`. Using this, *Wholly!* produces LLVM bitcode systematically for each of its packages, making it easier to analyze, transform, or optimize the packages at the LLVM bitcode level.

Wholly! uses `musl-llvm` [13], a fork of `musl-libc` [12]. `musl`'s efficiency is unparalleled in Linux `libc` implementations. Designed from the ground up for static linking, `musl` carefully avoids importing large amounts of code or data that the application will not use. The advantage of `musl-llvm` over `musl-libc` is that there is LLVM bitcode generated for all of `musl-llvm` except for a handful of functions that require assembly for part of their implementation.

3.3 Clarity of Build Environment

The build environment is kept as minimal as possible. It runs atop an Alpine Linux Docker image that contains only the tools necessary for the builds – in particular, a compiler and a linker – and can be reproduced using the Dockerfile below.

The minimalism of Alpine Linux as a build environment is consistent with the idea of lean builds: only the necessary runtime is present to ensure working builds, and there are no extra files that are not needed. Additionally, the build container is completely transparent and can be replicated by anyone using Docker.

An important aspect of the workflow is that the build environment is used as a disposable container: it is launched, populated with the base build environment and the build dependencies; when building and release of the package have completed successfully, the container is just killed and never reused. This way, every build is performed in a fresh and isolated environment.

3.4 Reproducibility of Builds

The build environment is a mere container, populated by following simple Dockerfiles. It is thus easily replicable, ensuring that builds are always performed in exactly the same way.

To imbue the builds with determinism and reproducibility, *Wholly!* also relies on sub-package checksum verification. After every build, sub-package checksums are consistently checked against the reference checksums in the contents file. A match attests to the correctness of the package's contents. To ensure that identical recipes produce the same sub-packages, *Wholly!* sets the last modification and access time of every file that is copied into a sub-package image since this metadata feeds into the checksum calculation.

3.5 Static Building

Although *Wholly!* can be used to build any suitable recipe, we chose to use only static linking of binaries and libraries instead of dynamic linking.

```
FROM alpine

# Building dependencies
RUN apk update
RUN apk add make binutils file git curl

# Copy build tools
RUN mkdir -p /tools/bin
COPY musl-clang/bin/* /tools/bin/
COPY clang-4.0/bin/* /tools/bin/
RUN ln -s /tools/bin/clang-4.0 /tools/bin/clang
RUN ln -s /tools/bin/clang-4.0 /tools/bin/clang++
RUN ln -s /tools/bin/clang-4.0 /tools/bin/clang-cpp

# Install gllvm
RUN apk add go musl-dev
ENV GOPATH="/usr/local/bin"
RUN go get github.com/SRI-CSL/gllvm/cmd/gclang
RUN go get github.com/SRI-CSL/gllvm/cmd/gclang++
RUN go get github.com/SRI-CSL/gllvm/cmd/get-bc
RUN mv ./usr/local/bin/bin/gclang /tools/bin
RUN mv ./usr/local/bin/bin/gclang++ /tools/bin
RUN mv ./usr/local/bin/bin/get-bc /tools/bin

RUN apk del go musl-dev

# Install tools
RUN chmod +x /tools/bin/*

# Folders
RUN mkdir -p /usr/bc
RUN mkdir -p /build
RUN mkdir -p /install

# Static environment variables
ENV PATH="/tools/bin:/root/go/bin:${PATH}"
ENV LLVM_CC_NAME musl-clang
ENV LLVM_CXX_NAME musl-clang++
ENV WLLVM_BC_STORE /usr/bc
```

Fig. 6. Dockerfile for *Wholly!* base build environment.

The reason is that we consider dynamic linking as being inconsistent with deployments on the modern software stack. With the popularity of cloud computing and emerging microservices, we need deployments that are smaller, faster and more specialized. While dynamic linking is suitable for general-purpose desktops, it leads to bloated and heavy deployments when applied to the container ecosystem, the cloud in general, and IoT devices. Indeed, using shared libraries instead of static binaries delays to runtime things that could have been determined at compile time:

- External symbols are resolved at runtime by a dynamic linker, adding a non negligible overhead to execution time.
- Unused functions from shared libraries are included in the deployment, whereas they could have been eliminated by link-time optimization.
- Dynamically-linked applications are dependent on a specific runtime that needs to be replicated in the target platform – including correct libraries and dynamic linker path and versions.

The *Wholly!* recipes that we created enforce the use of static linking so as to get specialized and smaller deployments easily. It allows us to release packages that make the most of link-time optimization to eliminate unused code while ensuring that these packages can run on a wide range of Linux-based systems, without assumptions being made about the runtime environment. Indeed, as a positive side effect of this, our static deployments are smaller than the ones found in the Docker ecosystem, since our resulting binaries ship with less runtime and library code. They are also faster, since they require no runtime symbol resolution and thus less time is spent in kernel code and context switches.

4 Evaluation

Wholly! recipes and Docker containers can be used to build arbitrary software packages for multiple target platforms. We applied *Wholly!* to Linux packages, targeting the x86-64 platform, and focused on efficient and debloated packaging. In this section, we evaluate the packages built by *Wholly!* against those provided by the lightweight and minimal Alpine Linux distribution. We directly compare selected packages from the two systems, using size and performance as metrics. We also evaluate how *Wholly!* contributes to performing GNU-independent builds of packages.

4.1 Size of Packages

Wholly! particularly targets minimal Docker deployments of applications. Given this, we compare the size of ready-to-deploy Docker images built on top of *Wholly!* packages on the one hand, and on Alpine Linux packages on the other hand. In the interest of ensuring a fair comparison, the packages that we compare have the same version, ship the same files, and are confirmed to run.

For the example of `nginx-1.12`, a small and popular HTTP server, Figures 7 and 8 show the two Dockerfiles that we use to construct images that will be compared. The Alpine Linux version is the smallest possible deployment that we can make using Alpine and its `apk` package manager. Some *Wholly!* packages need additional configuration and runtime details; in the case of `nginx-1.12`, we need to import a `busybox` shell, set up directories, and create users. Since our packages are independent of the platform and runtime, this configuration step is not performed at build time.

```
FROM alpine:latest
RUN apk update
RUN apk add nginx
```

Fig. 7. Dockerfile for Alpine's `nginx-1.12` deployment. Only packages from distribution repositories are imported.

Table 1 compares the size of the Docker images built for a representative set of packages.

For most of the packages, the *Wholly!* version is smaller, which is not surprising since the binaries are built statically and benefit from link-time optimization. The only exception among the tested packages is Node.js; this is because the vanilla Alpine package is built differently than the *Wholly!* version, and contains less functionality. The size of *Wholly!* packages makes them more consistent with small and specialized deployments in the cloud or in constrained environments, such as embedded systems.

	Alpine-based image (in MB)	*Wholly!*-based image (in MB)
nginx-1.12	6.44	4.32
bzip2-1.0	5.31	0.425
sqlite-3.18	8.73	1.24
python-2.7	43.6	35.4
nodejs-6.11	36.8	54.4
clang-4.0	225	165

Table 1. Size comparison for Docker images between *Wholly!* and Alpine. *Wholly!* packages are usually significantly smaller than their Alpine equivalent.

4.2 Performance of Deployments

In what follows, we try to see how *Wholly!* packages compete with commonly used applications and deployments in terms of performance. For this purpose, we analyzed the performance of different `nginx` servers and `clang` compilers.

We use the Dockerfile shown in Figure 8 to build our *Wholly!* `nginx` server. Note that we include configuration commands so as to be able to connect to the server on port 80. The other servers that we use for this comparison are the official Docker images – available at Docker Hub – `nginx:mainline-alpine` and `nginx:official`, the latter being the default image available for `nginx`. We chose the other servers because they are amongst the most pulled and deployed Docker images, according to Docker Hub [4]. We then apply the following process to all three images:

1. Run the `nginx` image in a Docker container on the host machine.
2. Check that `http://localhost:80/index.html` is reachable and returns the default `nginx` web page.
3. Run and benchmark 10,000 successive requests to this webpage using Apache's `ab` tool.

The results of the benchmark are available in Figure 9. The *Wholly!* deployment is slightly faster, and most importantly much smaller than the other ones.

We also compare *Wholly!*'s `clang` compiler to the one provided by Alpine Linux. To achieve this, we generate a number of random C files using `Csmith` [21], and measure the time needed to compile all these files sequentially with each compiler. We ensure that both compilers produce the exact same object files using hash comparisons. The results of the benchmark are provided in Figure 10 and show that *Wholly!*'s version of `clang` is significantly faster than the Alpine Linux version. Since we use our *Wholly!*-generated compiler in *Wholly!*, we get very good build performance for our packages. More detailed benchmarks using `perf` show that *Wholly!*'s statically built version of `clang` triggers fewer time-consuming operations, such as context switches.

4.3 Contribution to Environment-independent Builds

One of the objectives of *Wholly!* is to make builds easily reproducible. We achieve this by providing a Docker-based build environment that is minimal and easy to replicate. We also note that most of Linux's user space software is implicitly dependent on the GNU build environment and provide a workaround to avoid this.

```
FROM wholly-nginx-1.12-bin as bin
FROM wholly-nginx-1.12-rt as rt
FROM wholly-nginx-1.12-conf as conf
FROM busybox

COPY --from=bin / /
COPY --from=rt / /
COPY --from=conf / /

RUN mkdir /var/cache && mkdir /var/cache/nginx && mkdir /var/run \
  && touch /var/run/nginx.pid && addgroup -S nginx \
  && adduser -D -S -h /var/cache/nginx -s /sbin/nologin -G nginx nginx \
  && ln -sf /dev/stdout /var/log/nginx/access.log && ln -sf /dev/stderr /var/log/nginx/error.log

# Patch default configuration file to use port 80
COPY nginx.conf.patched /usr/conf/nginx.conf

EXPOSE 80

STOPSIGNAL SIGTERM

CMD ["nginx", "-g", "daemon off;"]
```

Fig. 8. Dockerfile that we use to build a runnable *Wholly!* `nginx-1.12` server.

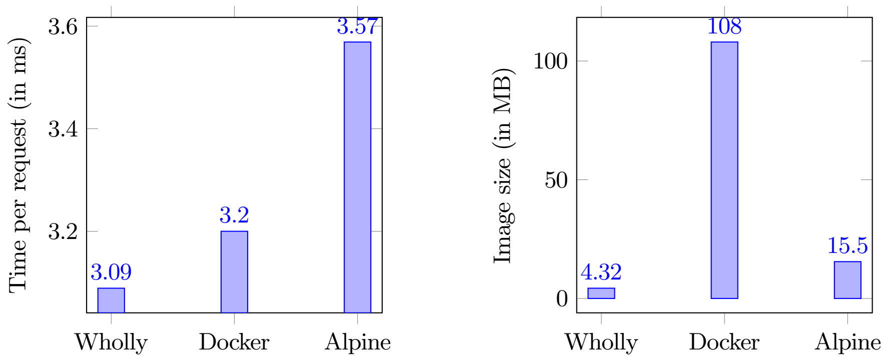

Fig. 9. Benchmarks for the `nginx` server show that the *Wholly!*-based version is slightly faster and much smaller than equivalents pulled from the Docker Hub.

Like Alpine Linux, we chose to build all of our C packages against `musl-libc`, a lightweight C standard library that is an alternative to GNU's `glibc`. Unlike Alpine, *Wholly!* uses `clang` instead of `gcc` as its C compiler. Indeed, this non-GNU toolchain makes it harder to build packages that would compile without modification using `gcc` and `glibc`. In particular, some functional differences between `musl-libc` and `glibc` require patching, and we make use of wrapper scripts for the compiler and linker to automatically build with our non-standard libraries. Still, some packages like `busybox` just don't support `clang` and require `gcc` to build. The Tuscan Catalog [19] illustrates how this issue plagues many Linux packages.

However, most of the packages can be built using *Wholly!*, provided we create appropriate patches for them. In what follows, we give examples of packages that require such processing:

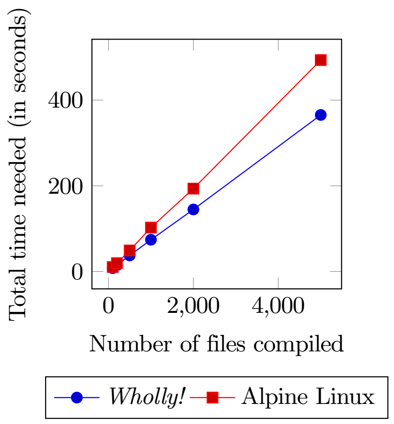

Fig. 10. Performance benchmark for `clang`: the *Wholly!*-built binary outperforms the one from the Alpine Linux repository by close to 40%.

- `nodejs-6.11` provides the `fully-static` flag to statically build the binary, but still passes the `--rdynamic` flag to the compiler, which prevents the creation of a statically-linked executable. This shows that `nodejs-6.11`'s build scripts were not even tested to build a real statically-linked executable. We created a patch to fix the build in *Wholly!*.
- Statically building a full-featured `python-2.7` executable is anticipated in the build scripts, but is not officially documented. Designing a recipe to build it is quite complicated and requires manually editing setup files.
- Surprisingly, `clang-4.0` package required the most patching. Despite its aim at being a replacement for `gcc` and related tools, it is incompatible out-of-the-box with `musl-libc`'s macros, for which we needed 4 patch files. One extra patch was required to remove the `-Wl,-rpath-link` flag that was consistently passed at compile time. Figure 11 shows a sample patch.

Our effort shows that it is possible to build the Linux user space in an environment that is not tied to the GNU toolchain. Indeed, *Wholly!* uses `clang` as its C compiler and `musl-libc` / `libc++` as its C / C++ standard libraries to build Alpine Linux's user space.

5 Applications

Wholly! has been used to construct efficient and portable builds of software packages for a number of platforms. We have built an entire Linux distribution, an entire Linux container ecosystem, and specialized virtual machines with microservices using *Wholly!*.

Alpine Linux: This is an independent, non-commercial, general-purpose Linux distribution. It is built around `musl-libc` and `busybox`. It is therefore smaller and more resource efficient than traditional GNU/Linux distributions. Alpine Linux uses its own package manager, called `apk`, and was designed with security in mind. Alpine Linux has a dedicated build infrastructure that consists of a number of scripts used to

```
diff -uNr cmake.old/modules/AddLLVM.cmake cmake/modules/AddLLVM.cmake
--- cmake.old/modules/AddLLVM.cmake 2017-01-17 13:47:58.000000000 -0800
+++ cmake/modules/AddLLVM.cmake 2017-06-05 08:40:55.000000000 -0700
@@ -671,7 +671,7 @@
     list(APPEND ALL_FILES "${LLVM_MAIN_SRC_DIR}/cmake/dummy.cpp")
   endif()
   if( EXCLUDE_FROM_ALL )
     add_executable(${name} EXCLUDE_FROM_ALL ${ALL_FILES})
   else()
@@ -1314,7 +1314,7 @@
   if(NOT ARG_OUTPUT_DIR)
@@ -1426,10 +1426,6 @@
     if(${CMAKE_SYSTEM_NAME} MATCHES "(FreeBSD|DragonFly)")
       set_property(TARGET ${name} APPEND_STRING PROPERTY
                    LINK_FLAGS " -Wl,-z,origin ")
-    elseif(${CMAKE_SYSTEM_NAME} STREQUAL "Linux"
-           AND NOT LLVM_LINKER_IS_GOLD)
-      set_property(TARGET ${name} APPEND_STRING PROPERTY
-                   LINK_FLAGS " -Wl,-rpath-link,
-                       ${LLVM_LIBRARY_OUTPUT_INTDIR} ")
     endif()
   else()
     return()
```

Fig. 11. The patch `cmake_fix_no_dynlinker_build.patch` is required for `clang-4.0` to handle static building correctly.

build packages. We chose it as a target platform because it is a Linux distribution that is built exclusively using `musl-libc`. However, it uses `gcc`, and not `clang`. We aim to build an entire Linux distribution with *Wholly!*, producing fine-grained packages, precise definitions of package dependencies, and for each library and executable, produce the corresponding LLVM bitcode to enable whole program analysis, transformations, and optimizations.

We have automatically translated Alpine build scripts to *Wholly!* recipe files. Our recipes are often simpler, more intuitive, and readable than Alpine build scripts. Another reason we chose Alpine Linux is the growing popularity of Docker containers. Docker has started using Alpine Linux for its base container images.

Docker Containers: Docker is the world's leading software container management platform. Containers are rapidly gaining traction as the preferred platform for deploying cloud applications and microservices. Docker containers are built using Dockerfiles and *Wholly!* uses Dockerfiles to build arbitrary software packages. As a result, it is possible to directly export packages and sub-packges as Docker images. Because we use static linking and link-time optimiation, Docker images produced by *Wholly!* are both smaller in size and faster. When utilizing microservices, it is desirable to support a fast deployment cycle. Being able to build images efficiently also enhances developer productivity by speeding up the debug and test cycle.

Minimal VMs: In 2017, Docker unveiled LinuxKit [9], a toolkit for building secure, lean, and portable Linux subsystems. It allows to bundle a number of Docker images, along with kernel support, to be built into a stand-alone VM. Because our Docker images are leaner and faster than those provided by Docker, we are able to produce more efficient VMs that can be booted on IoT devices, as well as on typical cloud infrastructure.

6 Whole Program Analysis

It is well known that when applying formal verification to source code, what is verified is not what is executed. Because an application can be compiled with different compilers and may be deployed with different versions of libraries, it is not possible to guarantee that source code verification translates into correct execution.

The systematic production of LLVM bitcode for any application built with *Wholly!*, allows us to apply formal verification to all of the application code and the libraries it is dependent on. We use SeaHorn [7], a fully automated analysis framework for LLVM-based languages, to perform a number of program analyses, such as abstract interpretation, invariant generation, memory safety, and bounded model-checking. Because of the reproducibility of builds, guaranteed by *Wholly!*, a formally analyzed applications is proved to exhibit the same behavior on different platforms.

The verification at the LLVM bitcode level can be done after code specialization using partial evaluation and code winnowing techniques, such as the ones implemented in OCCAM [11]. These techniques can be taken one step further so that I/O operations are specialized and all read and writes to files are replaced with in-memory loads and stores. This often improves performances by avoiding the execution of expensive system calls. Of note is that it also ensures that what is being verified is close to what is being executed.

7 Conclusion

We have presented *Wholly!*, a tool for building efficient, fine-grained, and LLVM-based packages for Alpine Linux, Docker containers, and LinuxKit VMs. *Wholly!* can be used in the future for cross-compiling these and arbitrary packages to support multiple architectures and operating systems. *Wholly!*'s uniform and reproducible build process will avoid many of the portability issues reported in recent studies [19], and can provide reproducible builds for a range of areas, including scientific computing. Combined with LLVM-based software specialization and optimization frameworks, such as OCCAM [11], and formal verification tools, such as SeaHorn [7], *Wholly!*supports the production of debloated, efficient, and verified code that can be deployed in practice.

8 Acknowledgement

This material is based upon work supported by the US National Science Foundation (NSF) under Grant ACI-1440800, Department of Homeland Security (DHS) Science and Technology Directorate, and the Office of Naval Research (ONR) under Contract No. N68335-17-C-0558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DHS, or ONR.

References

1. Alpine Linux, https://alpinelinux.org/
2. Patrick Cousot and Radhia Cousot, **Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints**, *4th ACM Symposium on Principles of Programming Languages (POPL)*, 1977.
3. Docker, https://www.docker.com/
4. Docker Hub, https://hub.docker.com/
5. gllvm, https://github.com/SRI-CSL/gllvm
6. Susanne Graf and Hassen Saidi, **Construction of abstract state graphs with PVS**, *9th International Conference on Computer Aided Verification (CAV)*, 1997.
7. Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge Navas, **The SeaHorn verification framework**, *27th International Conference on Computer Aided Verification (CAV)*, 2015.
8. Haskell Lightweight Virtual Machine, https://galois.com/project/halvm/
9. LinuxKit, https://github.com/linuxkit/linuxkit
10. Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft, **Unikernels: Library operating systems for the cloud**, *18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
11. Gregory Malecha, Ashish Gehani, and Natarajan Shankar, **Automated software winnowing**, *30th ACM Symposium on Applied Computing (SAC)*, 2015.
12. musl libc, https://www.musl-libc.org/
13. LLVM musl libc, https://github.com/SRI-CSL/musllvm
14. OSv, http://osv.io/
15. Stephen Piccolo and Michael Frampton, **Tools and techniques for computational reproducibility**, *GigaScience*, Vol. 5(1), 2016.
16. Rumprun unikernel, https://github.com/rumpkernel/
17. Christopher Smowton, **I/O Optimisation and elimination via partial evaluation**, *Ph.D. Thesis, Cambridge University*, 2014.
18. Standard Library, https://stdlib.com/
19. Tuscan Catalog, https://karkhaz.github.io/tuscan/
20. Wholly!, https://github.com/SRI-CSL/Wholly/
21. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr, **Finding and understanding bugs in C compilers**, *32nd ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011.