1

CDE: Automatically Package and Reproduce Computational Experiments

Philip J. Guo

Google, Inc.

CONTENTS

Although many social, cultural, and political barriers hinder reproducible research, one large technical barrier to reproducibility is that it is hard to distribute scientific code in a form that other researchers can easily execute on their own machines. Before your colleagues can run your computational experiments, they must first obtain, install, and configure compatible versions of the appropriate software and a variety of dependent libraries, which can be a frustrating and error-prone process. If even one portion of one dependency cannot be fulfilled, then your experiment will not be reproducible.

To eliminate this technical barrier to reproducibility, we have created a tool called CDE, which stands for **C**ode, **D**ata, and **E**nvironment packaging. CDE is easy to use: You simply execute the commands for your experiment under its supervision, and CDE automatically packages up all of the code, data, and environmental state that your commands accessed. When you send that self-contained package to your colleagues, they can rerun those exact commands on their machines without first installing or configuring anything. Moreover, they can even adjust the parameters in your original code and rerun to explore related hypotheses, or run your code on their own data sets to see how well it generalizes.

By using CDE to package your experimental code, data, and environment when you publish a paper, you can ensure that both you and your colleagues can reproduce the paper's results in the future. CDE currently works on 32- and 64-bit x86-Linux operating systems. In short, if you can run the original experiment on your own Linux computer, then your colleagues can run and modify it on their Linux computers without any setup effort.

CDE is free and open-source, available at `http://www.pgbovine.net/cde.html`

1.1 Motivation

The simple-sounding task of taking software that runs on one person's machine and getting it to run on another machine can be painfully difficult in practice, even if both machines have the same operating system. Since no two machines are identically configured, it is hard for developers to predict the exact versions of software and libraries already installed on potential users' machines and whether those conflict with the requirements of their own software. Thus, software companies devote considerable resources to creating and testing one-click installers for products such as Microsoft Office, Adobe Photoshop, and Google Chrome. Similarly, open-source developers must carefully specify the proper dependencies in order to integrate their software into package management systems [2] (e.g., RPM on Linux, MacPorts on Mac OS X). Despite these efforts, online forums and mailing lists are filled with discussions of users' troubles in compiling, installing, and configuring software and dependencies.

Researchers are unlikely to invest the effort to create one-click installers or wrestle with package managers, since their job is not to release production-quality software. Instead, they usually "release" their software by uploading their source code and data files to a server and writing some informal installation instructions. There is a slim chance that their colleagues will be able to run their research code "out-of-the-box" without some technical support.

1.2 CDE System Overview

In this chapter, we present a tool called CDE [1] that makes it easy for people to get their software running on other machines without the hassle of manually creating a robust installer or dealing with user complaints about dependencies. CDE automatically packages up the **C**ode, **D**ata, and **E**nvironment required to run a set of x86-Linux programs on other x86-Linux machines without any installation (see Figure 1.1). To use CDE, the user simply:

1. Prepends any set of Linux commands with the `cde` executable. `cde` executes the commands and uses `ptrace` system call interposition to collect all code, data, and environment variables used during execution into a self-contained package.
2. Copies the resulting CDE package to an x86-Linux machine running any distribution (*distro*) from the past ∼5 years.
3. Prepends the original packaged commands with the `cde-exec` executable to run them on the target machine. `cde-exec` uses `ptrace` to redirect file-related system

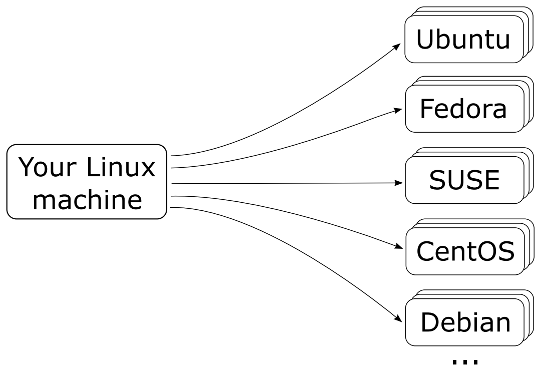

FIGURE 1.1
CDE enables users to package up any Linux program and deploy it to all modern Linux distros.

calls so that executables can load the required dependencies from within the package. Execution can range from ∼0% to ∼30% slower.

The main benefits of CDE are that creating a package is as easy as executing the target program under its supervision, and that running a program within a package requires no installation, configuration, or root permissions.

In addition, CDE offers an *application streaming mode*, described in Section 1.5.3. Figure 1.2 shows its high-level architecture: The system administrator first installs multiple versions of many popular Linux distros in a "distro farm" in the cloud (or an internal compute cluster). The user connects to that distro farm via an ssh-based protocol from any x86-Linux machine. The user can now run *any* application available within the package managers of any of the distros in the farm. CDE's streaming mode fetches the required files on-demand, caches them locally on the user's machine, and creates a portable distro-independent execution environment. Thus, Linux users can instantly run the hundreds of thousands of applications already available in the package managers of all distros without being forced to use one specific release of one specific distro[1].

We will use an example to introduce the core features of CDE. Suppose that Alice is a climate scientist whose experiment involves running a Python weather simulation script on a Tokyo data set using this Linux command:

```
python weather_sim.py tokyo.dat
```

Alice's script (`weather_sim.py`) imports some 3rd-party Python extension modules, which consist of optimized C++ numerical analysis code compiled into shared libraries. If Alice wants her colleague Bob to run and build upon her experiment, then it is not sufficient to just send her script and `tokyo.dat` data file to him. Even if Bob has a compatible version of Python on his machine, he will not be able to run her script until he compiles, installs, and configures the extension modules that she used (and all of their transitive dependencies).

[1]The package managers included in different releases of the same Linux distro often contain incompatible versions of many applications!

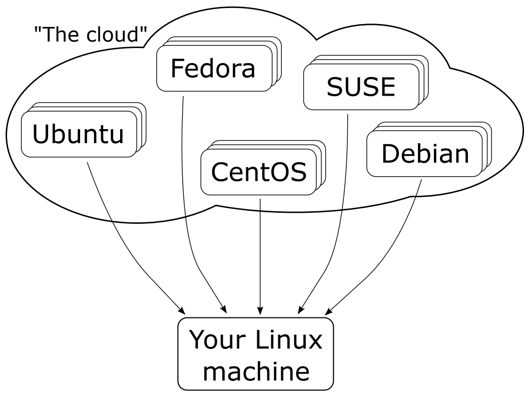

FIGURE 1.2
CDE's streaming mode enables users to run any Linux application on-demand by fetching the required files from a farm of pre-installed distros in the cloud.

1.2.1 Creating A New Package With `cde`

To create a self-contained package with all dependencies required to run her experiment on another machine, Alice prepends her command with the `cde` executable:

```
cde python weather_sim.py tokyo.dat
```

`cde` runs her command normally and uses the Linux `ptrace` mechanism to monitor all files it accesses throughout execution. `cde` creates a new sub-directory called `cde-package/cde-root/` and copies all of those accessed files into there, mirroring the original directory structure. For example, if her script dynamically loads an extension module (shared library) named `/usr/lib/weather.so`, then `cde` will copy it to `cde-package/cde-root/usr/lib/weather.so` (see Figure 1.3). `cde` also saves the values of environment variables in a file within `cde-package/`.

When execution terminates, the `cde-package/` sub-directory (which we call a "CDE package") contains all of the files required to run Alice's original command.

1.2.2 Executing A Package With `cde-exec`

Alice zips up the `cde-package/` directory and transfers it to Bob's Linux machine. Now Bob can run Alice's experiment without installing anything on his machine. He unzips the package, changes into the sub-directory containing the script, and prepends the original command with the `cde-exec` executable[2]:

```
cde-exec python weather_sim.py tokyo.dat
```

`cde-exec` sets up the environment variables saved from Alice's machine and executes the version of `python` and its extension modules from within the package. `cde-exec` uses `ptrace` to monitor all system calls that access files and rewrites their path arguments to the corresponding paths within the `cde-package/cde-root/` sub-directory. For example, when her script requests to load the `/usr/lib/weather.so` extension library using an `open` system call, `cde-exec` rewrites the path argument of the `open` call to `cde-package/cde-root/usr/lib/`

[2]The package contains a copy of `cde-exec`.

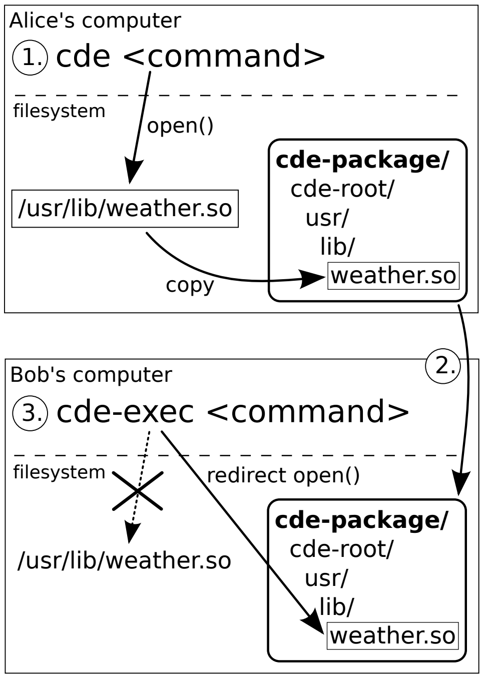

FIGURE 1.3
Example use of CDE: 1.) Alice runs her command with `cde` to create a package, 2.) Alice sends her package to Bob's computer, 3.) Bob runs that same command with `cde-exec`, which redirects file accesses into the package.

`weather.so` (see Figure 1.3). This path redirection is essential, because `/usr/lib/weather.so` probably does not exist on Bob's machine.

Not only can Bob reproduce Alice's exact experiment, but he can also edit her script and data set and then re-run to explore variations and alternative hypotheses, as long as his edits do not cause the script to import new Python extension modules that are not in the package. Also, since a CDE package is a directory tree, Bob can add additional data set files into the package to run related experiments.

1.2.3 CDE Package Portability

Alice's CDE package can execute on any Linux machine with an architecture and kernel version that are compatible with its constituent binaries. CDE currently works on 32-bit and 64-bit variants of the x86 architecture (i386 and x86-64, respectively). In general, a 32-bit `cde-exec` can execute 32-bit packaged applications on 32- and 64-bit machines. A 64-bit `cde-exec` can execute both 32-bit and 64-bit packaged applications on a 64-bit machine. Extending CDE to other architectures (e.g., ARM) is straightforward because the `strace` tool that CDE is built upon already works on many architectures. However, CDE packages cannot be transported *across* architectures without using a CPU emulator.

In practice, CDE packages are portable across Linux distros released within approximately five years of the distro where the package originated [13]. Besides sharing with colleagues such as Bob, Alice can also deploy her package to run on a cluster for more computational power or to a public server for real-time weather simulation reporting. Since she does not need to install any software or libraries as the root user, she does not risk perturbing existing software on those machines. Finally, having her script and all of its dependencies (including the Python interpreter and extension modules) encapsulated within a CDE package makes it somewhat "future-proof" and likely to still run on her machine even when its version of Python and associated extensions are upgraded in the future.

Users can combine CDE with a virtual machine to achieve greater portability. For example, if Alice wants her colleagues who run Windows, Mac OS, or an antiquated Linux to reproduce her experiments, she can put her CDE package within a Linux virtual machine (VM) and distribute the entire VM image. However, the price to pay for such portability is increased file size: A VM image file can be 10 to 100 times larger than a CDE package because it contains the entire operating system.

Finally, unlike language-based portability technologies (such as Java or Python `virtualenv`), CDE works on Linux programs written in any language or mix of languages.

1.2.4 Ignoring Files and Environment Variables

By convention, Linux directories such as `/dev`, `/proc`, and `/sys` contain pseudo-files (e.g., device files) that do not make sense to include in a CDE package. Also, environment variables such as `$XAUTHORITY` and the corresponding `.Xauthority` file (for X Window authorization) are machine-specific. Informed by our debugging experiences and user feedback, we have manually created a blacklist of directories, files, and environment variables for CDE to ignore, so that packages can be portable across machines. By "ignore" we mean that `cde` will not copy those files (or variables) into a package, and `cde-exec` will not redirect their paths and instead access the real versions on the machine. This user-customizable blacklist is implemented as a plain-text options file. Figure 1.4 shows this file's default contents.

CDE also allows users to customize which paths it should ignore (leave alone) and which it should redirect into the package, thereby making its sandbox "semi-permeable." For example, one computational scientist chose to have CDE ignore a directory that mounts an NFS share containing huge data files, because he knew that the machine on which he was going to execute the package also mounts that NFS share at the same path. Therefore, there was no point in bloating up the package with those data files.

1.2.5 Non-Goals

Our philosophy in designing CDE was to create the simplest possible tool that would allow a large class of real-world Linux programs to be portable across a range of contemporary distros. One way we have kept CDE's design simple was to limit its scope. Here are some tasks that CDE is *not* designed to perform:

- **Deterministic replay**: CDE does not try to replay exact execution paths like record-replay tools [8, 17, 20] do. Thus, CDE does not need to capture sources of randomness, thread scheduling, and other non-determinism. It also does not need to create snapshots of filesystem state for rollback/recovery.

- **OS/hardware emulation**: CDE does not spoof the OS or hardware. Thus, programs that require specialized hardware or device drivers will not be portable across machines. Also, CDE cannot capture remote network dependencies.

- **Security**: Although CDE isolates target programs in a chroot-like sandbox, it does not guard against attacks to circumvent such sandboxes [11]. Users should only run CDE packages from trusted sources. (Of course, the same warning applies to *all* downloaded software.)

```
# These directories often contain pseudo-files that shouldn't be tracked
ignore_prefix=/dev/
ignore_exact=/dev
ignore_prefix=/proc/
ignore_exact=/proc
ignore_prefix=/sys/
ignore_exact=/sys
ignore_prefix=/var/cache/
ignore_prefix=/var/lock/
ignore_prefix=/var/log/
ignore_prefix=/var/run/
ignore_prefix=/var/tmp/
ignore_prefix=/tmp/
ignore_exact=/tmp

ignore_substr=.Xauthority     # Ignore to allow X Window programs to work

ignore_exact=/etc/resolv.conf # Ignore so networking can work properly

# Access the target machine's password files:
# (some programs like texmacs need these lines to be commented-out,
#  since they try to use home directory paths within the passwd file,
#  and those paths might not exist within the package.)
ignore_prefix=/etc/passwd
ignore_prefix=/etc/shadow

# These environment vars might lead to 'overfitting' and hinder portability
ignore_environment_var=DBUS_SESSION_BUS_ADDRESS
ignore_environment_var=ORBIT_SOCKETDIR
ignore_environment_var=SESSION_MANAGER
ignore_environment_var=XAUTHORITY
ignore_environment_var=DISPLAY
```

FIGURE 1.4
The default CDE options file, which specifies the file paths and environment variables that CDE should ignore. `ignore_exact` matches an exact file path, `ignore_prefix` matches a path's prefix string (e.g., directory name), and `ignore_substr` matches a substring within a path. Users can customize this file to tune CDE's sandboxing policies (see Section 1.2.4).

- **Licensing**: CDE does not attempt to "crack" software licenses, nor does it enforce licensing or distribution restrictions. It is ultimately the package creator's responsibility to make sure that he/she is both willing and able to distribute the files within a package, abiding by the proper software and data set licenses.

1.3 Use Case Categories

Since we released the first version of CDE on November 9, 2010, it has been downloaded at least 10,000 times as of November 2012 [1]. We cannot track how many people have directly checked out its source code from GitHub [1], though.

We have exchanged hundreds of emails with CDE users and discovered five salient real-world use cases as a result of these discussions:

Creating reproducible computational experiments: The results of many computational science experiments can be reproduced within CDE packages because their code is output-deterministic [8], always producing the same outputs (e.g., statistics, tables, graphs) for a given input. We have received several emails describing how researchers have used CDE to make their experiments reproducible, including

- robotics motion planning experiments using C++ and OpenGL code [22],
- genetic algorithms for social networking using C++ and R code [18],
- and biological fingerprint identification using the LibSVM machine learning library and the Open Babel computational chemistry toolbox [15].

Distributing research software: The creators of several research tools found CDE online and used it to create portable packages that they uploaded to their websites:

The website for Graph-Tool, a Python/C++ module for analyzing graphs, lists these (direct) dependencies: "GCC 4.2 or above, Boost libraries, Python 2.5 or above, expat library, NumPy and SciPy Python modules, GCAL C++ geometry library, and Graphviz with Python bindings enabled" [6]. Unsurprisingly, lots of people had trouble compiling it: 47% of all messages on its mailing list (137 out of 289) were questions related to compilation problems. The author of Graph-Tool used CDE to automatically create a portable package (containing 149 shared libraries and 1909 total files) and uploaded it to his website so that users no longer needed to suffer through the pain of manually compiling it.

Arachni, a Ruby-based tool that audits web application security [5], requires six hard-to-compile Ruby extension modules, some of which depend on versions of Ruby and libraries that are not available in the package managers of most modern Linux distributions. Its creator, a security researcher, created and uploaded CDE packages and then sent us a grateful email describing how much effort CDE saved him: *"My guess is that it would take me half the time of the development process to create a self-contained package by hand; which would be an unacceptable and truly scary scenario."*

A British research programming team used CDE to make portable packages for their protein crystallography software[3]. Similarly, a team at Johns Hopkins University made CDE packages for CAWorks[4], medical visualization software for computational anatomy.

[3]http://www.ccp4.ac.uk/
[4]http://cis.jhu.edu/software/caworks/

The following email snippet from a conversation with its lead developer provides a sense of the complex dependencies that CDE automatically encapsulated: *"My program which is called CAWorks is huge with a massive dependency list. ParaView the base program uses vtk, python, zlib and of course Qt and all of their dependent libraries. While my program CAWorks adds to this libcurl, openssl, ITK and vxl, blitz, dicom, ivcon, clapack, getopt, gts, md5, quazip, and glib and all of their dependent libraries, none of which ParaView uses."*

Finally, we used CDE to create portable binary packages for two of our Stanford[5] colleagues' research tools, which were originally distributed as tarballs of source code: PADS [10] and Saturn [7]. 44% of the messages on the PADS mailing list (38 / 87) were questions related to troubles with compiling it (22% for Saturn). Once we successfully compiled these projects (after a few hours of improvising our own hacks since the documentation was grossly outdated), we created CDE packages by running their regression test suites. Now our fellow researchers no longer need to suffer through the compilation process.

To see the benefits of CDE here, note that the Saturn team leader admitted in a public email, "As it stands the current release likely has problems running on newer systems because of bit rot — some libraries and interfaces have evolved over the past couple of years in ways incompatible with the release" [3]. In contrast, CDE packages are largely immune to "bit rot" (until the user-kernel ABI changes) because they contain all dependencies.

Deploying computations to cluster or cloud: People working on computational experiments on their desktop machines often want to run them on a cluster for greater performance and parallelism. However, before they can deploy their computations to a cluster or cloud computing (e.g., Amazon EC2), they must first install all of the required executables and dependent libraries on the cluster machines. At best, this process is tedious and time-consuming, since cluster/cloud machines run older versions of software and libraries due to both slow upgrade cycles and concerns about security and stability. At worst, installation can be impossible, since regular users often do not have root access on cluster machines.

Using CDE, a user can create a self-contained package on their desktop machine and then execute that package on the cluster or cloud (possibly many instances in parallel), without needing to install any dependencies or to get root access on the remote machines.

For example, our Stanford colleague Peter wanted to use a department-administered 100-CPU cluster to run a parallel image processing job on topological maps. However, since he did not have root access on those older machines, it was nearly impossible for him to install all of the dependencies required to run his computation, especially the image processing libraries. Peter used CDE to create a package by running his job on a small data set on his desktop, transferred the package and the complete data set to the cluster, and then ran 100 instances of it in parallel there.

Similarly, we worked with labmates to use CDE to deploy the CPU-intensive Klee [9] automated bug finding tool from the desktop to Amazon's EC2 cloud computing service without needing to compile Klee on the cloud machines. Klee can be hard to compile since it depends on LLVM, which is very picky about specific versions of GCC and other build tools being present on the machine before it will compile.

Researchers have also used CDE to deploy computational experiments to internal compute clusters within several software companies, to the European Grid distributed computing infrastructure, and to the iPlant[6] cloud infrastructure (NSF-funded cyberinfrastructure for plant biologists).

On a related note, several researchers have used CDE to deploy their research software not to a cluster but rather to a webserver; that way, users can interact with their code

[5]CDE originated as a research project in the Computer Science Department at Stanford University.
[6]http://www.iplantcollaborative.org/

via a web interface. Since researchers often do not have root access on shared web hosting machines, it can be impossible to install all of the required dependencies on there.

Running production software on incompatible distros: Even production-quality software might be hard to install on Linux distros with older kernel or library versions, especially when system upgrades are infeasible. A user can run software under CDE supervision on a modern distro to create a package and then run that package on an older distro, regardless of what libraries are present on there.

For example, an engineer at Cisco wanted to run some new open-source tools on his work machines, but the IT department mandated that those machines run an older, more secure enterprise Linux distro. He could not install the tools on those machines because that older distro did not have up-to-date libraries, and he was not allowed to upgrade. Therefore, he installed a modern distro at home, ran CDE on there to create packages for the tools he wanted to port, and then ran the tools from within the packages on his work machines.

Hobbyists applied CDE in a similar way: A game enthusiast could only run classic games within a DOS emulator on one of his Linux machines, so he used CDE to create a package and can now play the games on his other machines. We also helped a user create a portable package for the Google Earth 3D map application, so he can now run it on older distros whose libraries are incompatible with Google Earth.

Class programming projects: A teaching assistant for Stanford's Parallel Computing course (CS 149) used CDE to package up the toolchain required to compile and run class programming projects; thus, students can focus on the actual programming rather than the drudgery of installation and configuration.

In addition, two users sent us CDE packages they created for collaborating on class assignments: Rahul, a Stanford grad student, was using NLTK [19], a Python module for natural language processing, to build a semantic email search engine for a machine learning class. Despite much struggle, Rahul's two teammates were unable to install NLTK on their Linux machines due to conflicting library versions. This meant that they could run only one instance of the project at a time on Rahul's laptop for query testing and debugging. When Rahul discovered CDE, he created a package for their project and was able to run it on his two teammates' machines, so that all three of them could test and debug in parallel. Joshua, an undergrad from Mexico, emailed us a similar story about how he used CDE to collaborate on and demo his virtual reality class project.

1.4 Implementation Details

This section describes the implementation of CDE in some detail, so it is relevant only for readers who are either curious about those details or who want to implement a similar tool.

CDE uses the Linux `ptrace` system call to monitor the target program's processes and threads, read/write to its memory, and modify its system call arguments, all without requiring root permission. System call interposition using `ptrace` is a well-known technique that computer systems researchers have used for implementing tools such as secure sandboxes [12, 16], record-replay systems [17], and user-level filesystems [21].

We implemented CDE by adding 3000 lines of C code to the `strace` system call monitoring tool. CDE works only on x86-based Linux machines (32-bit and 64-bit) but should be easy to extend to other hardware architectures. Although implementation details are

Category	Linux syscalls	`cde` action	`cde-exec` action
File path access	`open[at]`,`mknod[at]`, `fstatat64`,`access`, `faccessat`,`readlink[at]`, `truncate[64]`,`stat[64]`, `creat`,`lstat[64]`, `oldstat`,`oldlstat`, `chown[32]`,`lchown[32]`, `fchownat`,`chmod`,`fchmodat`, `utime`,`utimes`,`futimesat`	Copy file into package	Redirect path into package
Local sockets	`bind`,`connect`	None	Redirect path into package†
Mutate filesystem	`link[at]`,`symlink[at]`, `rename[at]`,`unlink[at]`, `mkdir[at]`,`rmdir`	Repeat in package	Redirect path into package
Get current dir.	`getcwd`	Update current directory	Spoof current directory
Change directory	`chdir`,`fchdir`	Update current directory	Update current directory
Spawn child	`fork`,`vfork`,`clone`	Track child process or thread	Track child process or thread
Execute program	`execve`	Copy binary into package	Maybe run dynamic linker

TABLE 1.1
The 48 (out of 338 total) Linux 2.6 system calls intercepted by `cde` and `cde-exec`, and actions taken for each category of syscalls. Syscalls with suffixes in [brackets] include variants both with and without the suffix: e.g., `open[at]` means `open` and `openat`. †For `bind` and `connect`, `cde-exec` only redirects the path if it is used to access a file-based socket for local IPC.

Linux-specific, these same ideas could be used to implement CDE for another OS such as Mac OS X or Windows.

1.4.1 Creating A New Package With `cde`

Primary action: The main job of `cde` is to use `ptrace` to monitor the target program's system calls and copy all of its accessed files into a self-contained package. The only relevant syscalls here are those that take a file path string as an argument, which are listed in the "File path access" category in Table 1.1 (and also `execve`). After the kernel finishes executing one of these syscalls and is about to return to the target program, `cde` wakes and observes the return value. If the return value signifies that the indicated file exists, then `cde` copies that file into the package (see Figure 1.5).

Note that many syscalls operate on files but take a file descriptor as an argument rather than a file path (e.g., `mmap`); `cde` does not need to track those, since it already tracks the preceding syscalls (e.g., `open`) that create file descriptors from file paths.

Copying files into package: Prior to copying a file into the package, `cde` creates all necessary sub-directories and symbolic links to mirror the original file's location. In our example from Figure 1.3, `cde` copies `/usr/lib/weather.so` into the package as `cde-package/cde-root/usr/lib/weather.so`. For efficiency, copies are done via Linux hard links if possible.

If a file is a symlink, then both it and its target must be copied into the package. Multiple levels of symlinks, to both files and directories, must be properly handled. More subtly, *any component* of a path may be a symlink to a directory, so the exact directory structure must be replicated within the package for `cde-exec` to work. For example, we once encountered a path `/usr/lib/gcc/4.1.2/libgcc.a`, where `4.1.2` is a symlink to a directory named `4.1.1`. We observed that some programs are sensitive to exact filesystem layout, so `cde` must faithfully replicate symlinks within the package, or else those programs will fail with cryptic errors when run from within the package.

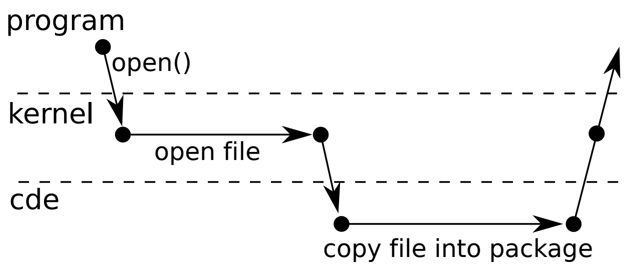

FIGURE 1.5
Timeline of control flow between the target program, kernel, and `cde` process during an `open` syscall.

Finally, if the file being copied is an ELF binary (executable or library code), then `cde` searches through the binary's contents for constant strings that are filenames and then copies those files into the package. Although this hack is simplistic, it works well in practice to partially overcome CDE's limitation of only being able to gather dependencies on executed paths (see Section 1.5.1 for more details). It works because many binaries dynamically load libraries whose filenames are constant strings. For example, we encountered a Python extension library that dynamically loads one of a few versions of the Intel Math Kernel Library based on the current CPU's capabilities. Without this hack, any given execution will copy only *one* version of the Intel library into the package, so packaged execution will fail when running on another machine with different CPU capabilities. Finding and copying all versions of the Intel library into the package makes the program more likely to run on machines with different hardware.

Here is how `cde` handles the other syscalls in Table 1.1:

Mutate filesystem: After each call that mutates the filesystem, `cde` repeats the same action on the corresponding copies of files in the package. For example, if a program renames a file from `foo` to `bar`, then `cde` also renames the copy of `foo` in the package to `bar`. This way, at the end of execution, the package's contents mirror the "post-state" of the original filesystem's contents, not the "pre-state" before execution.

Updating current working directory: At the completion of `getcwd`, `chdir`, and `fchdir`, `cde` updates its record of the monitored process's current working directory, which is necessary for resolving relative paths.

Tracking sub-processes and threads: If the target program spawns sub-processes, `cde` also attaches onto those children with `ptrace` (it attaches onto spawned threads in the same way). `cde` keeps track of each monitored process's current working directory and shared memory segment address (needed for Section 1.4.2). `cde` remains single-threaded and responds to events queued by `ptrace`.

This feature is useful for packaging up workflows consisting of multiple program invocations, such as a compilation job. Running "`cde make`" will track all sub-processes that the Makefile spawns and package up the source files and compiler toolchain. Now you can edit and compile the given project on another Linux machine by simply running "`cde-exec make`" without needing to install any compilation tools or header files on that machine.

execve syscall: `cde` copies the executable's binary into the package. For a script, `cde` finds the name of its interpreter binary from the shebang (`#!`) line. If the binary is dynamically-linked, `cde` also finds its dynamic linker (e.g., `ld-linux.so.2`) and copies it into the package. The dynamic linker is responsible for loading the shared libraries that a program needs at start-up time.

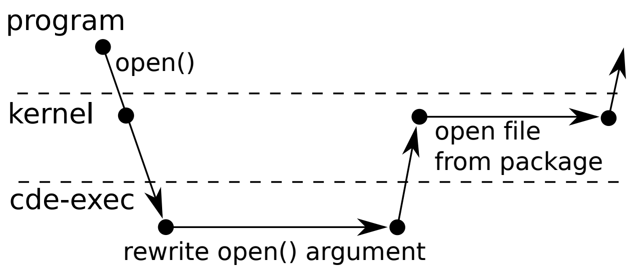

FIGURE 1.6
Timeline of control flow between the target program, kernel, and `cde-exec` process during an `open` syscall.

1.4.2 Executing A Package With `cde-exec`

Primary action: The main job of `cde-exec` is to use `ptrace` to redirect file paths that the target program requests into the package. Before the kernel executes most syscalls listed in Table 1.1, `cde-exec` rewrites their path argument(s) to refer to the corresponding path within `cde-package/cde-root/` (Figure 1.6). By doing so, `cde-exec` creates a chroot-like sandbox that fools the target program into "believing" that it is executing on the original machine. Unlike chroot, this sandbox does not require root access to set up, and it is user-customizable (see Section 1.2.4).

In our running example, suppose that Alice runs her experiment within the `/expt` directory on her computer:

```
cd /expt
cde python weather_sim.py tokyo.dat
```

She then sends the package to Bob's computer. If Bob unzips it into his home directory (`/home/bob`), then he can run these commands to execute her Python script:

```
cd /home/bob/cde-package/cde-root/expt
cde-exec python weather_sim.py tokyo.dat
```

Note that Bob needs to first change into the `/expt` sub-directory within the package, since that is where Alice's scripts and data files reside. When `cde-exec` starts, it finds Alice's `python` executable within the package (with the help of `$PATH`) and launches it. Now if her program requests to open, say, `/usr/lib/weather.so`, `cde-exec` rewrites the path argument of the `open` call to `/home/bob/cde-package/cde-root/usr/lib/weather.so`, so that the kernel opens the library version from within the package.

Implementing syscall rewriting: Since `ptrace` allows `cde-exec` to directly read and write into the target program's memory, the easiest way to rewrite a syscall's argument is to simply override its buffer with a new string. However, this approach does not work because the new path string is always longer than the original, so it might overflow the buffer. Also, if the program makes a system call with a constant string, the buffer would be read-only.

Instead, what `cde-exec` does is redirect the *pointer* to the buffer. When the target program (or one of its sub-processes) first makes a syscall, `cde-exec` forces it to make another syscall to attach a 16KB shared memory segment (a trick from Spillane et al. [21]). Now `cde-exec` can write data into that shared segment and have it be visible in the target program's address space. The two large rectangles in Figure 1.7 show the address spaces of the target program and `cde-exec`, respectively. Figure 1.7 illustrates the three steps involved in syscall argument rewriting:

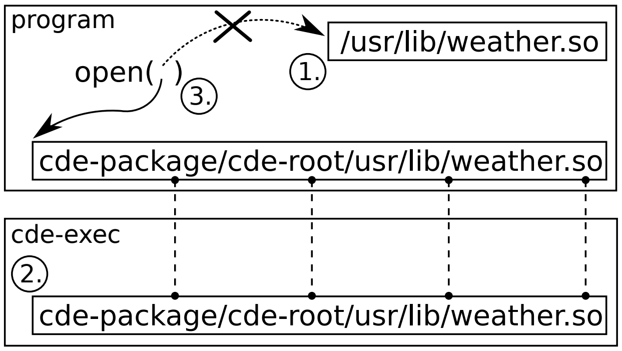
program
open( )
1.
3.
/usr/lib/weather.so
cde-package/cde-root/usr/lib/weather.so
cde-exec
2.
cde-package/cde-root/usr/lib/weather.so

FIGURE 1.7
Example address spaces of target program and `cde-exec` when rewriting path argument of `open`. The two boxes connected by dotted lines are shared memory.

1. `cde-exec` uses `ptrace` to read the original argument from the traced program's address space.
2. `cde-exec` creates a new string representing the path redirected inside of the package and writes it into the shared memory buffer. This value is immediately visible in the target program's address space.
3. `cde-exec` uses `ptrace` to mutate the syscall's filename `char*` argument(s) to point to the start of the shared memory buffer (in the target program's address space). x86-Linux syscall arguments are stored in registers, so `ptrace` mutates the target program's registers prior to executing the call. Most syscalls take only one filename argument, which is stored in `%ebx` on i386 and `%rdi` on x86-64. Syscalls such as `link`, `symlink`, and `rename` take two filename arguments; their second argument is stored in `%ecx` on i386 and `%rsi` on x86-64.

Spoofing current working directory: At the completion of the `getcwd` syscall, `cde-exec` mutates the return value string to eliminate all path components up to `cde-root/`. For example, when Bob runs Alice's script:

```
cd /home/bob/cde-package/cde-root/expt
cde-exec python weather_sim.py tokyo.dat
```

If her Python script requests its current working directory using `getcwd`, the kernel will return the true full path: `/home/bob/cde-package/cde-root/expt`. Then `cde-exec` will truncate that string so that it becomes `/expt`, which is the value it would have returned if it were running on Alice's machine. We have encountered many programs that break when `getcwd` is not spoofed.

There is no danger of buffer overflow here since the new string is always shorter, and the `char*` buffer passed into `getcwd` cannot be read-only, since the kernel must be able to update its contents. Some programs call `readlink("/proc/self/cwd")` to get the current working directory, so `cde-exec` also spoofs the return value for that particular syscall instance.

execve syscall: When the target program executes a dynamically-linked binary, `cde-exec` rewrites the `execve` syscall arguments to execute the dynamic linker stored in the package (with the binary as its first argument) rather than directly executing the binary. For example, if Bob invokes "`cde-exec python weather_sim.py tokyo.dat`", `cde-exec` will prepend the dynamic linker filename to the `argv` array argument of the `execve` syscall:

```
argv[0]: cde-package/cde-root/lib/ld-linux.so.2
argv[1]: /usr/bin/python
argv[2]: weather_sim.py
argv[3]: tokyo.dat
```

(Also note that although `argv[1]` is `/usr/bin/python`, that path will get redirected into the version of the binary file within the CDE package during the `open` syscall.)

Here is why `cde-exec` needs to explicitly execute the dynamic linker: When a user executes a dynamically-linked binary, Linux first executes the system's default dynamic linker to resolve and load its shared libraries. However, we have found that the dynamic linker on one Linux distro might not be compatible with binaries created on another distro, due to minor differences in ELF binary formats. Therefore, to maximize portability across machines, `cde` copies the dynamic linker into the package, and `cde-exec` executes the dynamic linker from the package rather than the target machine's builtin dynamic linker. Without this hack, we have noticed that even a trivial "hello world" binary compiled on one distro (e.g., Ubuntu with Linux 2.6.35) will not run on an older distro (e.g., Knoppix with Linux 2.6.17)[7].

A side-effect of rewriting `execve` to call the dynamic linker is that when a target program inspects its own executable name, the kernel will return the name of the dynamic linker, which is incorrect. Thus, `cde-exec` spoofs the return values of calls to `readlink("/proc/self/exe")` and `readlink("/proc/<$PID>/exe")` to return the original executable's name. This spoofing is necessary because some narcissistic programs crash with cryptic errors if their own names are not properly identified!

1.5 Advanced Features

We now describe three advanced CDE features that are relevant for power users: semi-automated package completion, seamless execution mode, and application streaming mode.

1.5.1 Semi-Automated Package Completion

CDE's main limitation is that it packages only the files accessed on executed program paths. Thus, programs run from within a CDE package will fail when executing paths that access new files (e.g., libraries, configuration files) that the original execution(s) did not.

Unfortunately, *no automatic tool* (static or dynamic) can find and package up all of the files required to successfully execute all possible program paths, since that problem is undecidable in general. Similarly, it is also impossible to automatically quantify how "complete" a CDE package is or determine what files are missing, since every file-related system call instruction could be invoked with complex or non-deterministic arguments. For example, the Python interpreter executable has only one `dlopen` call site for dynamically loading extension modules, but that `dlopen` could be called many times with different dynamically-generated string arguments derived from script variables or configuration files.

There are two ways to cope with this package incompleteness problem. First, if the user executes additional program paths, then CDE will add new files into the same `cde-package/` directory. However, making repeated executions can get tedious, and it is unclear how many or which paths are necessary to complete the package[8].

[7]It actually crashes with a cryptic "Floating point exception" error message.
[8]similar to trying to achieve 100% coverage during software testing

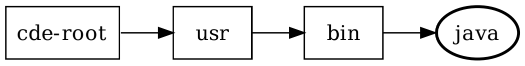

FIGURE 1.8
The result of copying a file named `/usr/bin/java` into `cde-root/`.

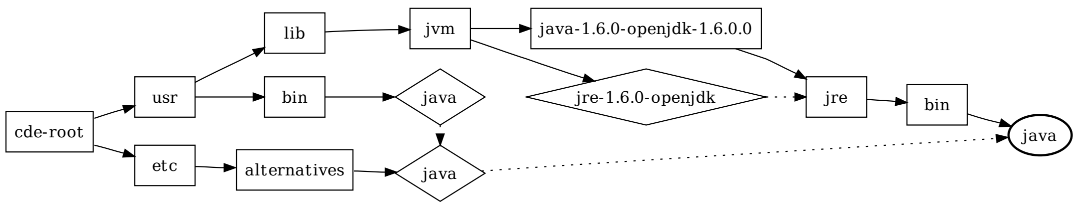

FIGURE 1.9
The result of using OKAPI to deep-copy a single `/usr/bin/java` file into `cde-root/`, preserving the exact symlink structure from the original directory tree. Boxes are directories (solid arrows point to their contents), diamonds are symlinks (dashed arrows point to their targets), and the bold ellipse is the real `java` executable.

Another way to make CDE packages more complete is by manually copying additional files and sub-directories into `cde-package/cde-root/` (see Section 1.5.1.3 for more details). For example, while executing a Python script, CDE might automatically copy the few Python standard library files it accesses into, say, `cde-package/cde-root/usr/lib/python/`. To complete the package, the user could copy the entire `/usr/lib/python/` directory into `cde-package/cde-root/` so that *all* Python libraries are present.

However, programs also depend on shared libraries that reside in system-wide directories such as `/lib` and `/usr/lib`. Copying the entire contents of those directories into a package results in lots of wasted disk space. In Section 1.5.1.2, we present an automatic heuristic technique that finds nearly all shared libraries that a program requires and copies them into the package.

1.5.1.1 OKAPI: Deep File Copying

Before describing our heuristics for completing CDE packages, we first introduce a utility library we built called OKAPI (pronounced "*oh-copy*"), which performs detailed copying of files, directories, and symlinks. OKAPI does one seemingly simple task that turns out to be tricky in practice: copying a filesystem entity (i.e., a file, directory, or symlink) from one directory to another while fully preserving its original sub-directory and symlink structure (a process that we call *deep-copying*). CDE uses OKAPI to copy files into the `cde-root/` sub-directory when creating a new package, and the support scripts of Sections 1.5.1.2 and 1.5.1.3 also use OKAPI.

For example, suppose that CDE needs to copy the `/usr/bin/java` executable file into `cde-root/` when it is packaging a Java application. The straightforward way to do this is to use the standard `mkdir` and `cp` utilities. Figure 1.8 shows the resulting sub-directory structure within `cde-root/`, with the boxes representing directories and the bold ellipse representing the copy of the `java` executable file located at `cde-root/usr/bin/java`. However, it turns out that if CDE were to use this straightforward copying method, the Java application would *fail to run* from within the CDE package! This failure occurs because the `java` executable introspects its own path and uses it as the search path for finding the Java standard libraries. On our Fedora Core 9 machine, the Java standard libraries are actually installed in `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0`, so when `java` reads its own path as `/usr/bin/java`, it cannot possibly use that path to find its standard libraries.

For Java applications to run from within CDE packages, all of their constituent files must be "deep-copied" into the package while replicating their original sub-directory and symlink structures. Figure 1.9 illustrates the complexity of deep-copying a single file, `/usr/bin/java`, into `cde-root/`. The diamond-shaped nodes represent symlinks, and the dashed arrows point to their targets. Notice how `/usr/bin/java` is a symlink to `/etc/alternatives/java`, which is itself a symlink to `/usr/lib/jvm/jre-1.6.0-openjdk/bin/java`. Another complicating factor is that `/usr/lib/jvm/jre-1.6.0-openjdk` is itself a symlink to the `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/` directory, so the actual `java` executable resides in `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/bin/`. Java can only find its standard libraries when these paths are all faithfully replicated within the CDE package.

The OKAPI utility library automatically performs the deep-copying required to generate the filesystem structure of Figure 1.9. Its interface is as simple as ordinary `cp`: The caller simply requests for a path to be copied into a target directory, and OKAPI faithfully replicates the sub-directory and symlink structure.

OKAPI performs one additional task: rewriting the contents of symlinks to transform absolute path targets into relative path targets within the destination directory (e.g., `cde-root/`). In our example, `/usr/bin/java` is a symlink to `/etc/alternatives/java`. However, OKAPI cannot simply create the `cde-root/usr/bin/java` symlink to also point to `/etc/alternatives/java`, since that target path is outside of `cde-root/`. Instead, OKAPI must rewrite the symlink target so that it actually refers to `../../etc/alternatives/java`, which is a relative path that points to `cde-root/etc/alternatives/java`.

The details of this particular example are not important, but the high-level message that Figure 1.9 conveys is that deep-copying even a single file can lead to the creation of over a dozen sub-directories and (possibly-rewritten) symlinks. The problem that OKAPI solves is not Java-specific; we have observed that many real-world Linux applications fail to run from within CDE packages unless their files are deep-copied in this intricate way.

Aside from being an integral part of CDE, OKAPI is also available as a free standalone command-line tool [1]. To our knowledge, no other Linux file copying tool (e.g., `cp`, `rsync`) can perform the deep-copying and symlink rewriting that OKAPI does.

1.5.1.2 Heuristics For Copying Shared Libraries

When Linux starts executing a dynamically-linked executable, the dynamic linker (e.g., `ld-linux*.so*`) finds and loads all shared libraries that are listed in a special `.dynamic` section within the executable file. Running the `ldd` command on the executable shows these start-up library dependencies. When CDE is executing a target program to create a package, CDE finds all of these dependencies as well because they are loaded at start-up time via `open` system calls.

However, programs sometimes load shared libraries in the middle of execution using, say, the `dlopen` function. This run-time loading occurs mostly in GUI programs with a plug-in or extension architecture. For example, when the user instructs Firefox to visit a web page with a Flash animation, Firefox will use `dlopen` to load the Adobe Flash Player shared library. `ldd` will not find that dependency since it is not hard-coded in the `.dynamic` section of the Firefox executable, and CDE will only find that dependency if the user actually visits a Flash-enabled web page while creating a package for Firefox.

We have created a simple heuristic-based script that finds most or all shared libraries that a program requires[9]. The user first creates a base CDE package by executing the target program once (or a few times) and then runs our script, which works as follows:

1. Find all ELF binaries (executables and shared libraries) within the package using the Linux `find` and `file` utilities.
2. For each binary, find all constant strings using the `strings` utility, and look for strings containing "`.so`" since those are likely to be shared libraries.
3. Call the `locate` utility on each candidate shared library string, which returns the *full absolute paths* of all installed shared libraries that match each string.
4. Use OKAPI to copy each library into the package.
5. Repeat this process until no new libraries are found.

This heuristic technique works well in practice because programs often list all of their dependent shared libraries in string *constants* within their binaries. The main exception occurs in dynamic languages such as Python or MATLAB, whose programs often dynamically generate shared library paths based on the contents of scripts and configuration files. Of course, our technique provides no completeness guarantees since the package completeness problem is undecidable in general.

1.5.1.3 OKAPI-Based Directory Copying

In general, running an application once under CDE monitoring only packages up a subset of all required files. In our experience, the easiest way to make CDE packages complete is to copy entire sub-directories into the package. To facilitate this process, we created a script that repeatedly calls OKAPI to copy an entire user-specified directory into `cde-root/`, automatically following symlinks to other directories and recursively copying as needed. (Note that simply running "`cp -aR`" is not sufficient since that does not follow and preserve symlinks.)

Although this approach might seem primitive, it is effective in practice because applications often store all of their files in a few top-level directories. When a user inspects the directory structure within `cde-root/`, it is usually obvious where the application's files reside. Thus, the user can run our script to copy those directories into the package.

1.5.2 Seamless Execution Mode

When executing a program from within a CDE package, `cde-exec` redirects all file accesses into the package by default, thereby creating a chroot-like sandbox with `cde-package/cde-root/` as the pseudo-root directory (see Figure 1.3, Step 3).

This default chroot-like execution mode is fine for running self-contained GUI applications such as games or web browsers, but it is a somewhat awkward way to run most types of UNIX-style command-line programs that researchers often prefer. If users are running, say, a compiler or command-line image processing utility from within a CDE package, they would need to first move their input data files into the package, run the target program using `cde-exec`, and then move the resulting output data files back out of the package, which is a cumbersome process.

Let's consider a modified version of the Alice-and-Bob example from Section 1.2. Suppose Alice is a researcher who is developing a Python script to detect anomalies in network log files. She normally runs her script using this Linux command:

```
python detect_anomalies.py net.log
```

[9]always a superset of the shared libraries that `ldd` finds

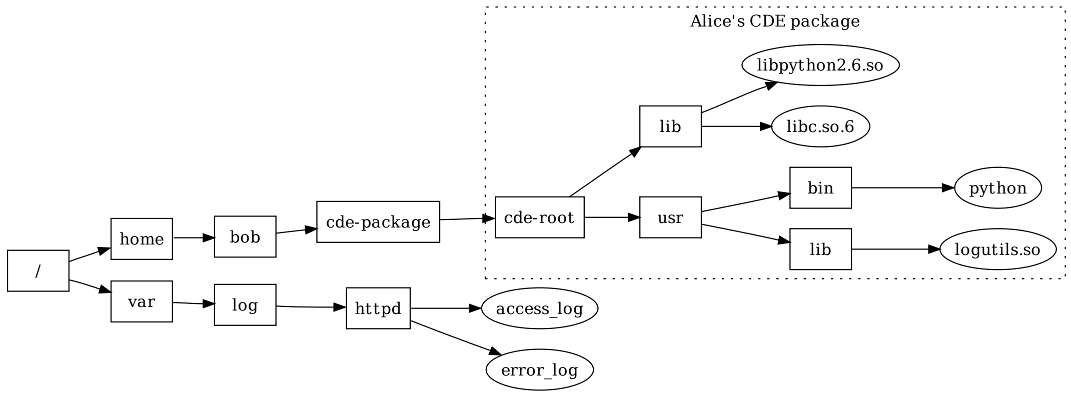

FIGURE 1.10
Example filesystem layout on Bob's machine after he receives a CDE package from Alice (boxes are directories, ellipses are files). CDE's seamless execution mode enables Bob to run Alice's packaged script on the log files in `/var/log/httpd/` without first moving those files inside of `cde-root/`.

Let's say she packages up her command with CDE and sends that package to Bob, who can now re-run her original analysis on the `net.log` file from within the package. However, if Bob wants to run Alice's script on his own log data (e.g., `bob.log`), then he needs to first move his data file inside of `cde-package/cde-root/`, change into the appropriate subdirectory deep within the package, and run:

```
cde-exec python detect_anomalies.py bob.log
```

In contrast, if Bob had actually installed the proper version of Python and its required extension modules on his machine, then he could run Alice's script from *anywhere* on his filesystem with no restrictions.

Some CDE users wanted CDE-packaged programs to behave just like regularly-installed programs rather than requiring input files to be moved inside of a `cde-package/cde-root/` sandbox, so we implemented a *seamless execution mode* that largely achieves this goal.

Seamless execution mode works using a simple heuristic: If `cde-exec` is being invoked from a directory *not* in the CDE package (i.e., from somewhere else on the user's filesystem), then only redirect a path into `cde-package/cde-root/` if the file that the path refers to actually exists within the package. Otherwise simply leave the path unmodified so that the program can access the file normally. No user intervention is needed in the common case.

The intuition behind why this heuristic works is that when programs request to load libraries and other mandatory components, those files must exist within the package, so their paths are redirected. On the other hand, when programs request to load an input file passed via, say, a command-line argument, that file does not exist within the package, so the original path is used to load it from the native filesystem.

In the example shown in Figure 1.10, if Bob ran Alice's script to analyze an arbitrary log file on his machine (e.g., his web server log, `/var/log/httpd/access_log`), then `cde-exec` will redirect Python's request for libraries (e.g., `/lib/libpython2.6.so` and `/usr/lib/logutils.so`) inside of `cde-root/` since those files exist within the package, but `cde-exec` will *not* redirect `/var/log/httpd/access_log` and instead load the real file from its original location.

Seamless execution mode fails when the user wants the packaged program to access a file

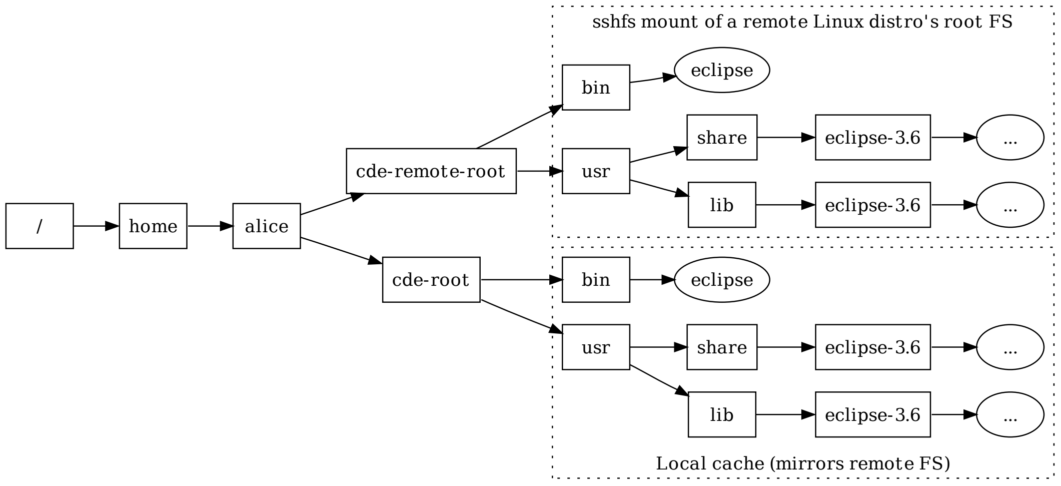

FIGURE 1.11
An example use of CDE's streaming mode to run Eclipse 3.6 on any Linux machine without installation. `cde-exec` fetches all dependencies on-demand from a remote Linux distro and stores them in a local cache.

from the native filesystem, but an identically-named file actually exists within the package. In the above example, if `cde-package/cde-root/var/log/httpd/access_log` existed, then that file would be processed by the Python script instead of `/var/log/httpd/access_log`. There is no automated way to resolve such name conflicts, but `cde-exec` provides a "verbose mode" where it prints out a log of what paths were redirected into the package. The user can inspect that log and then manually write redirection/ignore rules in a configuration file (see Figure 1.4) to control which paths `cde-exec` redirects into `cde-root/`. For instance, the user could tell `cde-exec` to *not* redirect any paths starting with `/var/log/httpd/*`.

1.5.3 On-Demand Application Streaming

With CDE's streaming mode, users can instantly run any Linux application on-demand without having to create, transfer, or install any packages. Figure 1.2 shows a high-level architectural overview. The basic idea is that a system administrator first installs multiple versions of many popular Linux distros in a "distro farm" in the cloud (or an internal compute cluster). When a user wants to run some application that is available on a particular distro, they use sshfs (an ssh-based network filesystem [4]) to mount the root directory of that distro into a special `cde-remote-root/` mount point. Then the user can use CDE's streaming mode to run any application from that distro locally on their own machine.

Implementation and Example: Figure 1.11 shows an example of streaming mode. Let's say that Alice wants to run the Eclipse 3.6 IDE on her Linux machine, but the particular distro she is using makes it difficult to obtain all the dependencies required to install Eclipse 3.6. Rather than suffering through finding, installing, and configuring all dependent libraries and software, Alice can simply connect to a distro in the farm that contains Eclipse 3.6 and then use CDE's streaming mode to "harvest" the required dependencies on-demand.

Alice first mounts the root directory of the remote distro at `cde-remote-root/`. Then she runs "`cde-exec -s eclipse`" (`-s` activates streaming mode). `cde-exec` finds and executes `cde-remote-root/bin/eclipse`. When that executable requests shared libraries, plug-ins, or any other files, `cde-exec` will redirect the respective paths into `cde-remote-root/`, thereby executing the version of Eclipse 3.6 that resides in the cloud distro. However, note that the application is running locally on Alice's machine, not in the cloud.

Astute readers will recognize that running applications in this manner can be slow, since files are being accessed from a remote server. While sshfs performs some caching, we have found that it does not work well enough in practice. Thus, we have implemented our own caching layer within CDE: When a remote file is accessed from `cde-remote-root/`, `cde-exec` uses OKAPI to make a deep-copy into a local `cde-root/` directory and then redirects that file's path into `cde-root/`. In streaming mode, `cde-root/` initially starts out empty and then fills up with a subset of files from `cde-remote-root/` that the target program has accessed.

To avoid unnecessary filesystem accesses, CDE's cache also keeps a list of file paths that the target program tried to access from the remote server, even keeping paths for *non-existent files*. On subsequent runs, when the program tries to access one of those paths, `cde-exec` will redirect the path into the local `cde-root/` cache. It is vital to track non-existent files since programs often try to access non-existent files at start-up while, say, searching for shared libraries by probing a list of directories in a search path. If CDE did not track non-existent files, then the program would still access the directory entries on the remote server before discovering that those files *still* do not exist, thus slowing down execution.

With this cache in place, the first time an application runs, all of its dependencies must be downloaded, which could take several seconds to minutes. This one-time delay is unavoidable. However, subsequent runs simply use the files already in the local cache, so they execute at regular `cde-exec` speeds. Even running a *different* application for the first time might still result in some cache hits for, say, generic libraries such as `libc`, so the entire application does not need to be downloaded.

Finally, the package incompleteness problem faced by regular CDE (see Section 1.5.1) no longer exists in streaming mode. When the target application needs to access new files that do not yet exist in the local cache (e.g., Alice loads a new Eclipse plug-in for the first time), those files are transparently fetched from the remote server and cached.

Synergy With Package Managers: Nearly all Linux users are currently running one particular distro with one default package manager that they use to install software. For instance, Ubuntu users must use APT, Fedora users must use YUM, SUSE users must use Zypper, and Gentoo users must use Portage. Moreover, different releases of the *same* distro contain different software package versions, since distro maintainers add, upgrade, and delete packages in each new release[10].

As long as a piece of software and all of its dependencies are present within the package manager of the exact distro release that a user happens to be using, then installation is trivial. However, as soon as even one dependency cannot be found within the package manager, then users must revert to the arduous task of compiling from source (or configuring a custom package manager).

CDE's streaming mode frees Linux users from this single-distro restriction and allows them to run software that is available within the package manager of any distro in the cloud distro farm. The system administrator is responsible for setting up the farm and provisioning access rights (e.g., ssh keys) to users. Then users can directly install packages in any cloud distro and stream the desired applications to run locally on their own machines.

Philosophically, CDE's streaming mode maximizes user freedom since users are now free to run any application in any package manager from the comfort of their own machines,

[10]We once tried installing a machine learning application that depended on the `libcv` computer vision library. The required `libcv` version was found in the APT repository on Ubuntu 10.04, but it was not found in the repositories on the two neighboring Ubuntu releases: 9.10 and 10.10.

regardless of which distro they choose to use. CDE complements traditional package managers by leveraging all of the work that the maintainers of each distro have already done and opening up access to users of all other distros. This synergy can potentially eliminate quasi-religious squabbles and flame-wars over the virtues of competing distros or package management systems. Such fighting is unnecessary since CDE allows users to freely choose from among all of them.

1.6 Discussion

Our design philosophy underlying CDE is that people should be able to package up their Linux software and deploy it to run on other Linux machines with as little effort as possible. However, we are not proposing CDE as a replacement for traditional software installation. CDE packages have a number of limitations. Most notably,

- They are not guaranteed to be complete.
- Their constituent shared libraries are "frozen" and do not receive regular security updates. (Static linking also shares this limitation.)
- They run slower than native applications due to `ptrace` overhead. We measured slowdowns of up to 28% in our experiments [13], but slowdowns can be worse for I/O-heavy programs.

Software engineers who are releasing production-quality software should obviously take the time to create and test one-click installers or integrate with package managers. But for the millions of research scientists, prototype designers, system administrators, programming course students and teachers, and hobby hackers who just want to deploy their ad-hoc software as quickly as possible, CDE can emulate many of the benefits of traditional software distribution with much less required labor: In just minutes, users can create a base CDE package by running their program under CDE supervision, use our *semi-automated heuristic tools* (Section 1.5.1) to make the package complete, deploy to the target Linux machine, and then execute it in *seamless execution mode* (Section 1.5.2) to make the target program behave like it was installed normally.

Practical Lessons Learned: Here are some generalizable lessons that we have learned in the past two years of developing CDE and supporting thousands of diverse users.

- First and foremost, start with a conceptually clear core idea, make it work for basic non-trivial cases, document the still-unimplemented tricky cases, launch your tool, and then get feedback from real users. User feedback is by far the easiest way for you to discover what bugs are important to fix and what new features to add next.
- A simple and appealing quick-start webpage guide and screencast video demo are essential for attracting new users. No potential user is going to read through dozens of pages of an academic research paper before deciding to try your tool. In short, even hackers need to learn to be great salespeople.
- To maximize your tool's usefulness, you must design it to be easy-to-use for beginners but also to give advanced users the ability to customize it to their liking. One way to accomplish this goal is to have well-designed default settings, which can be adjusted via command-line options or configuration files. The defaults must work effectively "out-of-the-box" without any tuning, or else new users will get frustrated.

- Resist the urge to add new features just because they're "cool," interesting, or potentially useful. Only add new features when there are compelling real users who demand it. Instead, focus your development efforts on fixing bugs, writing more test cases, improving your documentation, and, most importantly, attracting new users.
- Users are by far the best sources of bug reports, since they often stress your tool in ways that you could have never imagined. Whenever a user reports a bug, send them a sincere *thank you* note, try to create a representative minimal test case, and add it to your regression test suite.
- If a user has a conceptual misunderstanding of how your tool works, then think hard about how you can improve your documentation or default settings to eliminate this misunderstanding.

Reflecting on the past two years of serving CDE's users, we believe that its success thus far is largely due to it being a conceptually simple tool that has been meticulously engineered to do one thing well—eliminating Linux software dependency problems.

1.7 Future Vision: Ph.D.-In-A-Box

Despite the fact that so much modern scientific research is being done on computers, research papers are still the primary means of disseminating new knowledge. Dead trees are an impoverished communications medium, though. The ideas in this chapter could be extended to build a richer electronic medium where one's colleagues, apprentices, and intellectual adversaries can interactively explore the results of one's experiments. Imagine enhancing CDE packages with fine-grained versioning history and notes, perhaps hosted on a cloud service where readers can visit a web site to re-run and tweak those experiments. If we could capture and present a rich history of a project's progression over time, then readers can learn from the *entire research process*, not merely digest the final products.

To convey the potential benefits of learning from research *processes* rather than just end results, we will make an analogy to mathematics. Mathematics research papers are written in a concise manner presenting a minimal set of proofs of lemmas and theorems. Readers unfamiliar with the process of discovery in mathematics might mistakenly assume that some lofty genius must have dreamt up the perfect proof fully-formed and scribbled it down on paper. The truth is far messier: Much like computational researchers, mathematicians explore numerous hypotheses, go down dead-ends, backtrack repeatedly, talk through ideas with colleagues, and gradually cobble together a finished proof. Then they painstakingly whittle down that proof until it can be presented as elegantly as possible, and only then do they write up the final paper. The vast majority of intellectual wisdom lies in the *process* of working through the problems, and such knowledge is not represented anywhere in the published paper. Mathematicians learn their craft not just by reading papers, but by actually watching their colleagues and mentors at work. Imagine if there was a way to capture and present the entire months-long process of a famous mathematician coming up with a particular proof. Aspiring mathematicians could learn far more from such an interactive presentation than from simply reading the final polished paper.

We believe that such a goal of "total recall" is easier to accomplish in the context of computational research. Since most of the work is being done on the computer, it is easier to trace the entire workflow history and provide user interfaces for in-context annotations and notetaking [14]. First-class support for branching and backtracking are vital needs in such a system, since much of the wisdom gained from a research apprenticeship is learning from what did not work and what dead-ends to avoid. In this vision, all Ph.D. students would maintain a hard disk image containing the complete trials and tribulations of their five to seven (or more) years' worth of computational experiments. This "Ph.D.-In-A-Box" could be used to train new students and to pass down all of the implicit knowledge, experiences, tricks, and wisdom that are often lost in a dead-tree paper dissertation.

Extending this analogy further, imagine an online library filled with the collected electronic histories of all research projects, not just their final results in published form. It now becomes possible to perform pattern recognition and aggregation across multiple projects to discover common "tricks of the trade." Someone new to a field, say machine learning, can now immersively learn from the collective wisdom of thousands of expert machine learning researchers rather than simply reading their papers. One could argue that, in the limit, such a system would be like "indexing" all of those researchers' brains and making that knowledge accessible. We speculate that such a system can be more effective than "brain indexing," since people subconsciously apply tricks from their intuitions and often forget the details of what they were working on (especially failed trials). In this vision of the future, a paper is merely a facade for the real contributions of the full research process.

Such a dream "Ph.D.-In-A-Box" system can approach the holy grail of universally reproducible research in three main ways:

1. Researchers can revisit and reflect upon their old experiments, perhaps uncovering hidden biases that might have skewed results (e.g., multiple comparison problems in statistics).
2. Colleagues and students can reproduce, learn from, and scrutinize the entire workflow history of fellow researchers, not just the final products in the form of published papers. If properly authenticated, such an audit trail can be used to detect evidence of "cherry picking" and inappropriate "slicing and dicing" of data and code parameters to achieve statistically significant results.
3. "Meta-researchers" can perform meta-analyses of research processes, methodologies, trends, and findings within a field by data mining the computational archives of all researchers in that field.

The good news is that the technology to realize this dream already exists. A properly motivated team can use much of the ideas and tools described in this book to create a "Ph.D.-In-A-Box" system that makes the still-elusive ideal of reproducible research into a pervasive reality. Just like how we now take for granted that emails and documents are stored and synchronized in the cloud, perhaps in a decade or two, computational researchers will take for granted that all of their experiments are versioned, annotated, archived, curated, and fully reproducible.

Bibliography

[1] CDE public source code repository, `https://github.com/pgbovine/CDE`.

[2] List of software package management systems, `http://en.wikipedia.org/wiki/List_of_software_package_management_systems`.

[3] Saturn online discussion thread, `https://mailman.stanford.edu/pipermail/saturn-discuss/2009-August/000174.html`.

[4] SSH Filesystem, `http://fuse.sourceforge.net/sshfs.html`.

[5] `arachni` project home page, `https://github.com/Zapotek/arachni`.

[6] `graph-tool` project home page, `http://projects.skewed.de/graph-tool/`.

[7] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the Saturn project. PASTE '07, pages 43–48. ACM, 2007.

[8] Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. SOSP '09, pages 193–206. ACM, 2009.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI '08, pages 209–224. USENIX Association, 2008.

[10] Kathleen Fisher and Robert Gruber. PADS: a domain-specific language for processing ad hoc data. In *PLDI '05*, pages 295–304. ACM, 2005.

[11] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. NDSS '03, 2003.

[12] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. NDSS '04, 2004.

[13] Philip J. Guo. CDE: Run any Linux application on-demand without installation. In *Proceedings of the 2011 USENIX Large Installation System Administration Conference*, LISA '11. USENIX Association, 2011.

[14] Philip J. Guo and Margo Seltzer. Burrito: Wrapping your lab notebook in computational infrastructure. In *TaPP '12: Proceedings of the 4th USENIX Workshop on the Theory and Practice of Provenance*, 2012.

[15] Markus Heinonen, Huibin Shen, Nicola Zamboni, and Juho Rousu. Metabolite identification and molecular fingerprint prediction via machine learning. *Bioinformatics*, 2012.

[16] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. NDSS '00, 2000.

[17] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. SIGMETRICS '10, pages 155–166, 2010.

[18] Mayank Lahiri and Manuel Cebrian. The genetic algorithm as a general diffusion model for social networks. In *Proc. of the 24th AAAI Conference on Artificial Intelligence*. AAAI Press, 2010.

[19] Edward Loper and Steven Bird. NLTK: The Natural Language Toolkit. In *In ACL Workshop on Effective Tools and Methodologies for Teaching NLP and Computational Linguistics*, 2002.

[20] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *AADEBUG*, pages 69–76. ACM Press, 2005.

[21] Richard P. Spillane, Charles P. Wright, Gopalan Sivathanu, and Erez Zadok. Rapid file system development using ptrace. In *Experimental Computer Science*. USENIX Association, 2007.

[22] Ioan A. Sucan and Lydia E. Kavraki. Kinodynamic motion planning by interior-exterior cell exploration. In *Int'l Workshop on the Algorithmic Foundations of Robotics*, pages 449–464, 2008.