CDE: Using System Call Interposition to Automatically Create Portable Software Packages

Philip J. Guo and Dawson Engler
Stanford University

April 5, 2011

(This technical report is an extended version of our 2011 USENIX ATC paper)

Abstract

It can be painfully difficult to take software that runs on one person's machine and get it to run on another machine. Online forums and mailing lists are filled with discussions of users' troubles with compiling, installing, and configuring software and their myriad of dependencies. To eliminate this dependency problem, we created a system called CDE that uses system call interposition to monitor the execution of x86-Linux programs and package up the Code, Data, and Environment required to run them on other x86-Linux machines. The main benefits of CDE are that creating a package is completely automatic, and that running programs within a package requires no installation, configuration, or root permissions. Hundreds of people throughout both academia and industry have used CDE to distribute software, demo prototypes, make their scientific experiments reproducible, run software natively on older Linux distributions, and deploy experiments to compute clusters.

1 Introduction

Most programmers want other people to run their software. Unfortunately, the path from having a piece of software running on the programmer's own machine to getting it running on someone else's machine is fraught with potential pitfalls. For instance, the programmer might have forgotten to document a crucial step in the magic incantation needed during the installation process. Or forgotten to list a library version dependency, leading to mysterious run-time errors when the wrong version gets silently run on the user's machine. Or listed the right library version, but one which is either hard to obtain or conflicts with a library needed by a different program on the user's machine. Or the software itself might require libraries that depend on many other libraries, which themselves need to be transitively obtained and installed by the user, leading to an aggravating experience known as *dependency hell*. Or the user might lack permissions to install software packages in the first place, a common occurrence on corporate machines and compute clusters that are administered by IT departments. Finally, the user (recalling bitter past experiences) may be reluctant to perturb a working environment by upgrading or downgrading library versions just to try out new software.

As a testament to the ubiquity of software deployment problems, consider the prevalence of online forums and mailing list discussions dedicated to troubleshooting installation and configuration issues. For example, almost half of the messages sent to the mailing lists of two mature research tools, the `graph-tool` mathematical graph analysis library [10] and the PADS system for processing ad-hoc data [17], were questions about how to compile and install them: 47% of 289 emails for `graph-tool`, and 44% of 87 emails for PADS. As an example from commercial software, the Google Chrome help forum for "install/uninstall issues" has 4501 threads (in Apr. 2011).

We have created a system named CDE that eases the pain of software deployment. 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. It works as follows:

1. Prepend any Linux command with the `cde` executable. `cde` executes your command and uses `ptrace` system call interposition to collect all the code, data files, and environment variables used during execution into a self-contained package.
2. Copy the resulting package to any modern x86-Linux machine.
3. Change into the package directory and prepend the original command with the `cde-exec` executable. `cde-exec` loads the stored environment variables and then uses `ptrace` to redirect file-related system calls so that executables and libraries can load the required dependencies from within the package.

The main benefits of CDE are that creating a package is completely automatic, and that running programs within a package requires no installation, configuration, or root permissions, thereby eliminating dependency hell.

Our experiments show that CDE packages created on a modern x86-Linux distribution (distro) can run on a variety of popular x86-Linux distros from the past 5 years. Although CDE only works on Linux, both the software dependency problem it addresses and the ideas embodied in our solution are relevant across all operating systems.

Contributions: This paper's main contribution is a novel use of system call interposition to automatically create portable software packages (§3). We validate our CDE system on packages that our users created (§4.1), showing that they are portable across Linux distros (§4.2), that disk space (§4.3) and performance (§4.4) overheads are acceptable, and that dynamic dependency tracking is necessary (§4.5) and works well in practice (§4.6).

1.1 Example real-world use cases

Since its initial release in Nov. 2010, over 2,000 people have downloaded CDE [2], and we have exchanged hundreds of emails with users in both academia and industry. Here are some representative use cases:

Reproducible research: A fundamental tenet of science is that colleagues should be able to reproduce the results of one's experiments. In the past few years, science journals and CS conferences (e.g., SIGMOD, FSE) have encouraged authors of published papers to put their code and datasets online, so that others can independently re-run, verify, and build upon their experiments. However, it can be hard for people to set up all of the (often-undocumented) dependencies required to re-run experiments. In fact, it can even be difficult to re-run *one's own experiments* in the future, due to inevitable OS and library upgrades. For example, to ensure that he could later re-run and adjust experiments in response to reviewer critiques for a paper submission [15], our groupmate Cristian took the hard drive out of his computer at paper submission time and archived it in his drawer.

With CDE, scientists can run the experiment once on their machine under CDE supervision to create a package, and colleagues can run that package on any modern Linux machine to repeat the experiment. For example, a robotics researcher used CDE to package up his experiments from a motion planning paper [26]. We were able to run his experiments and reproduce his results on six popular Linux distros from the past 5 years.

Distributing software: 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." [10] 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 woes. 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 compiling it. We were able to download and run his package on six popular Linux distros from the past 5 years.

Running software without perturbing the OS: A system administrator working with a proprietary firewall product used CDE to package up some network monitoring utilities on his desktop machine and ran them on a production server without the risk of breaking a live system by installing new software. He told us via email:

> "The reason I did not want to install additional libraries on the server in question was that it is a highly customized version of RedHat and we simply did not have the knowledge required to determine if this installation would somehow interfere with the proprietary software running on it. It was simply not worth disturbing a production system, and I was sure there had to be a better way of doing it. Luckily I found CDE and gave it a try."

Deploying computations to a cluster: People developing computational experiments on their desktop machines often want to run them on a cluster for greater performance and parallelism, but they might not have permission from the sysadmin to install anything on the cluster. A security analyst at McAfee used CDE to package up a CPU-intensive experiment on his desktop machine and deployed it to run on a compute cluster without needing to install dependencies on the cluster. He told us about the effort he saved by using CDE:

> "The most aggravating dependency that was missing [on the cluster machines] was the mongo client library which, to get it installed properly on the target systems, would require me to install an entire build environment (G++, scons, automake, etc) which is something CDE allowed me to successfully avoid."

Demoing a prototype to clients: An engineering contractor was creating a prototype analytics GUI for an enterprise database. His clients wanted to try running his prototype on their office machines, but it depended on a myriad of difficult-to-install GUI libraries (e.g., wxWidgets, PLplot). The engineer ran his prototype using CDE

on his machine to create a package and was able to have his clients run it without installing anything on their machines. He told us afterward that he estimated it might take a knowledgeable Linux user 2 to 3 days of trial-and-error to set up the same environment as his development machine so that they could install and run his prototype. That would have been too much effort for the clients to undertake, so they might not have agreed to demo his prototype if he had not used CDE.

Running software on an incompatible OS version: Even production-quality software might not run on some OS variants, most commonly due to library incompatibilities. For example, one user told us that the popular Google Earth 3D map application could not run on some Linux distros that seemed to meet its stated minimum system requirements. Thus, we installed Google Earth on our local distro, ran it once under CDE supervision to create a portable package, and sent that package to the user. He was able to run our package on distros on which Google Earth was not normally able to run.

1.2 Comparison to related work

We know of no published system that automatically creates portable software packages *in situ* from a live running machine like CDE does. Existing tools for creating self-contained applications all require the user to manually specify dependencies. For example, Mac OS X programmers can create self-contained application bundles using Apple's developer tools [7]. Mac OS X bundles are structurally similar to CDE packages, encapsulating all dependencies within an ordinary filesystem hierarchy.

VMware ThinApp is a commercial tool that automatically creates self-contained portable Windows applications. However, a user can only create a package by having ThinApp monitor the installation of new software [11]. Unlike CDE, ThinApp cannot be used to create packages from existing software already installed on a live machine, which is our most common use case.

Virtual machine snapshots achieve CDE's main goal of capturing all dependencies required to execute a set of programs on another machine. However, they require the user to always be working within a VM from the start of a project (or else re-install all of their software within a new VM). Also, VM snapshot disk images are (by definition) larger than the corresponding CDE packages, often by an order of magnitude, since they must also contain the OS kernel and other extraneous applications. CDE is a more lightweight solution because it enables users to create and run packages natively on their own machines rather than through a VM. Virtual machines are complementary to CDE, though, because a user can create a package using CDE and copy that package into a Linux VM image to make it portable across operating systems.

System call interposition using `ptrace` is a well-known technique that researchers have used for implementing tools such as secure sandboxes [19, 20], record-replay systems [21], and user-level filesystems [25].

Record-replay systems [14, 21, 24] are loosely related to CDE because both strive to reproduce program execution on other machines. However, CDE does not try to deterministically replay one exact execution path. Unlike record-replay, CDE is intended to let users easily run multiple paths in packaged programs without installation.

2 CDE system overview

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 dataset 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).

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). `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.

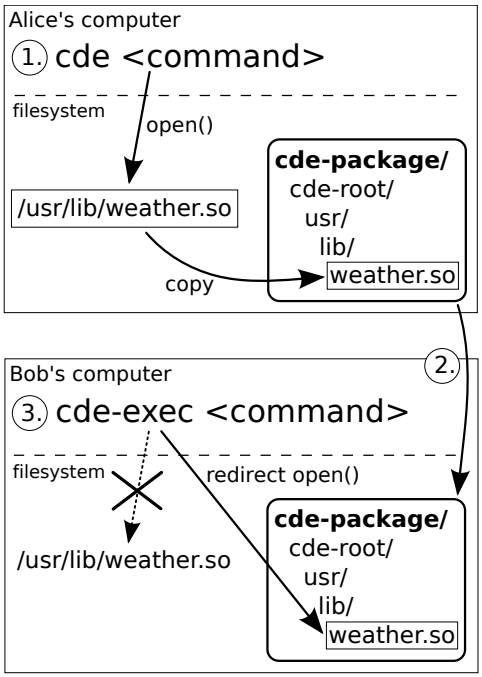

Figure 1: Example use of CDE: 1.) Alice runs her command with `cde` to create a package, 2.) Alice sends package to Bob's computer, 3.) Bob runs command with `cde-exec`, which redirects file accesses into package.

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 (also in the package):

```
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/weather.so` (see Figure 1). 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 dataset and then rerun to explore variations and alternative hypotheses, as long as he does not cause the script to import new Python extension modules that are not in the package.

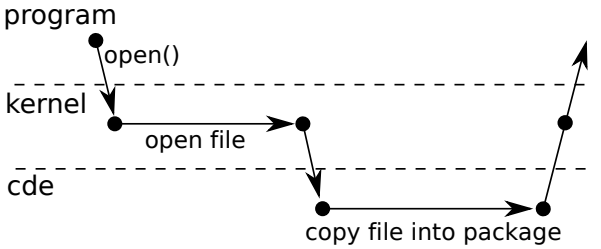

Figure 2: Timeline of control flow between target program, kernel, and `cde` process during an `open` syscall.

3 Design and implementation

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. We implemented CDE by adding 2500 lines of C code to the `strace` system call monitoring tool. Our implementation only works on x86-based Linux machines (32-bit and 64-bit) but should be straightforward to extend to other hardware architectures. Although implementation details are Linux-specific, the same ideas could be used to implement CDE for another OS such as Mac OS X or Windows.

3.1 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. `cde` only cares about the subset of syscalls that take a file path string as an argument, which are listed in the 'File path access' category in Table 1. 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 (Figure 2).

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 calls 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, `cde` will copy `/usr/lib/weather.so` into the package as `cde-package/cde-root/usr/lib/weather.so` (Figure 1). For efficiency, copies are done via 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,

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 dir.	Spoof current dir.
Change directory	chdir, fchdir	Update current working directory	
Spawn child	fork, vfork, clone	Track child process or thread	
Execute program	execve	Copy binary into package	Maybe run dynamic linker

Table 1: The 48 Linux system calls intercepted by cde and cde-exec, and actions taken for each category of syscalls. Syscalls with suffixes in [brackets] include variants with/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.

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 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.

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. 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 only copy *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:

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.

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 §3.2). cde remains single-threaded and responds to events queued by ptrace.

This feature is useful for packaging up workflows consisting of multiple program invocations, like a compilation job. For example, 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.

execve: 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.

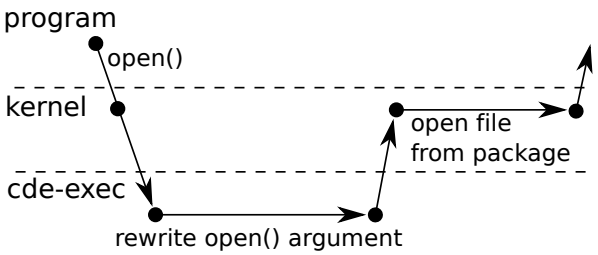

Figure 3: Timeline of control flow between target program, kernel, and `cde-exec` during an `open` syscall.

3.2 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, `cde-exec` rewrites their path argument(s) to refer to the corresponding path within `cde-package/cde-root/` (Figure 3). 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 3.3).

In our 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 version 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.

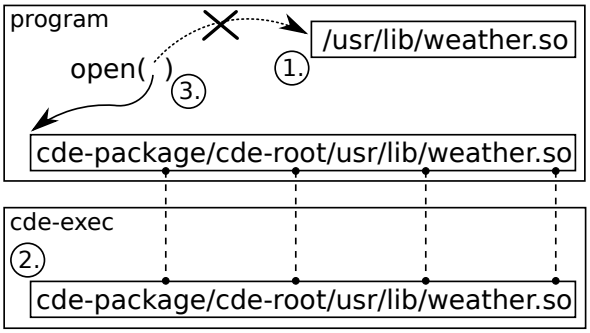

Figure 4: Example address spaces of target program and `cde-exec` when rewriting path argument of `open`. The two boxes connected by dotted lines are shared memory.

Instead, what `cde-exec` does is redirect the *pointer* to the buffer. When the target program (or one of its subprocesses) first makes a syscall, `cde-exec` forces it to make another syscall to attach a 16KB shared memory segment (a trick from [25]). 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 4 show the address spaces of the target program and `cde-exec`, respectively. Figure 4 illustrates the three steps involved in syscall argument rewriting:

1. `cde-exec` uses `ptrace` to read the original argument from the 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 argument 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 only take one filename argument, which is stored in `%ebx` on i386 and `%rdi` on x86-64. `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 buffer cannot be read-only, since the kernel must be able to mutate it. Some programs call `readlink("/proc/self/cwd")` to get current working directory, so we also spoof the return value for that particular syscall instance.

execve: 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.

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 having Linux execute the system's version. 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).

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.

3.3 Ignoring files and environment vars

By convention, Linux directories like `/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 like `$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 (customizable) blacklist of a dozen directories, files, and environment variables for CDE to ignore, so that packages can be portable across machines [1]. 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.

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 user 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.

This user-customizable blacklist is implemented as an options file. Figure 5 shows this file's default contents.

3.4 Non-goals

CDE only intercepts 14% of all Linux 2.6 syscalls (48 out of 338), but those are sufficient for creating portable self-contained Linux packages. CDE does not need to intercept more syscalls because it is not designed to perform:

- **Deterministic replay**: CDE does not try to exactly replay the original execution paths. Thus, it does not need to capture sources of randomness, thread scheduling, and other non-determinism [21, 24].
- **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-based) dependencies.
- **Security**: Although CDE isolates target programs in a chroot-like sandbox, it does not guard against attacks to circumvent such sandboxes [18]. Users should only run CDE packages from trusted sources.

3.5 Limitations

Executing a command within a CDE package will fail if:

- the Linux kernel or hardware architecture is incompatible with the binaries in the package. Only virtualization or emulation can overcome this limitation.
- the arguments or input change to make the program load a new shared library that the original execution did not load. We show in Section 4.6 that in the typical case, programs load the majority of their libraries at start-up, so the inputs can change a lot before the program needs to load additional libraries.
- the arguments or input change to make the program load another file that is not in the package. Since

```
# These directories often contain pseudo-files that shouldn't be tracked
ignore_prefix=/dev/
ignore_prefix=/proc/
ignore_prefix=/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

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

Figure 5: 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.

> a CDE package is just an ordinary directory tree, it is easy for users to directly add more files into the package if necessary. Also, if the user runs multiple commands in the same directory, `cde` will simply add additional files into the same `cde-package/`.

Note that a user who manually creates software packages by bundling together executables, libraries, and other files will face these exact same limitations, so CDE never performs worse than this manual approach. Of course, an expert user with the proper domain knowledge can create a more complete package than any automatic tool can (see Section 4.6); CDE can still aid these experts by creating a partially-complete package and then allowing them to manually fill in the remaining files.

In addition, CDE is limited by the limitations of `ptrace` and of executing binaries by explicitly invoking the dynamic linker: `ptrace` can cause subtle differences in the semantics of traced processes, most notably that a process being monitored by `ptrace` cannot itself `ptrace` another process, which precludes the use of CDE alongside applications like symbolic debuggers. Also, there is a known bug on certain Ubuntu distros where the `bash` shell non-deterministically crashes when invoked explicitly with a dynamic linker; a workaround is to have CDE use the machine's native `bash` shell on those distros.

4 Evaluation

To show that CDE is a practical and effective system, our evaluation addresses the following questions:

- How are people using CDE (§4.1)?
- How portable are CDE packages (§4.2)?
- How large are CDE packages (§4.3)?
- How much run-time slowdown is there (§4.4)?
- Is dynamic dependency tracking necessary (§4.5)?
- When running programs within a package, how much can inputs differ from the originals (§4.6)?

4.1 Use cases and benchmark packages

Since we released the first version of the CDE executable online on Nov 9, 2010, it has been downloaded at least 2,000 times (as of April 2011) [2]; we cannot track how many people have directly checked out its source code, though. We have exchanged hundreds of emails with CDE users and discovered six salient real-world use cases as a result of these discussions.

Table 2 summarizes the 16 CDE packages we used as benchmarks in our experiments. They contain software written in diverse programming languages. We have put

Package name	Description	Dependencies	Creator
Distributing research software			
`arachni`	Web app. security scanner framework [9]	Ruby (+ extensions)	security researcher
`graph-tool`	Lib. for manipulation & analysis of graphs [10]	Python, C++, Boost	math researcher
`pads`	Language for processing ad-hoc data [17]	Perl, ML, Lex, Yacc	self
`saturn`	Static program analysis framework [13]	Perl, ML, Berkeley DB	self
Running production software on incompatible distros			
`meld`	Interactive visual diff and merge tool for text	Python, GTK+	software engineer
`bio-menace`	Classic video game within a MS-DOS emulator	DOSBox, SDL	game enthusiast
`google-earth`	3D interactive map application by Google	shell scripts, OpenGL	self
Creating reproducible computational experiments			
`kpiece`	Robot motion planning algorithm [26]	C++, OpenGL	robotics researcher
`gadm`	Genetic algorithm for social networks [22]	C++, make, R	self
Deploying computations to cluster or cloud			
`ztopo`	Batch processing of topological map images	C++, Qt	graduate student
`klee`	Automatic bug finder & test case generator [15]	C++, LLVM, μClibc	self
Submitting executable bug reports			
`coq-bug-2443`	Incorrect output by Coq proof assistant [3]	ML, Coq	bug reporter
`gcc-bug-46651`	Causes GCC compiler to segfault [4]	gcc	bug reporter
`llvm-bug-8679`	Runs LLVM compiler out of memory [6]	C++, LLVM	bug reporter
Collaborating on class programming projects			
`email-search`	Natural language semantic email search	Python, NLTK, Octave	college student
`vr-osg`	3D virtual reality modeling of home appliances	C++, OpenSceneGraph	college student

Table 2: CDE packages used as benchmarks in our experiments, grouped by use cases. 'self' in the 'Creator' column means package created by first author; all other packages created by CDE users (mostly people we have never met).

all benchmark packages online, along with the command lines required to execute them [1]. We now summarize the use case categories and benchmarks (shown in **bold**).

Distributing research software: The creators of two research tools found CDE online and used it to create portable binary packages that they uploaded to their websites: **`arachni`**, a Ruby-based tool that audits web application security [9], 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, uploaded CDE packages and 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."* In Section 1.1, we already described how the creator of the **`graph-tool`** library used CDE to create portable Linux packages.

In addition, we used CDE to create portable binary packages for two of our Stanford colleagues' research tools, which were originally distributed as tarballs of source code: **`pads`** [17] and **`saturn`** [13]. 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 instructions were outdated), we created CDE packages by running their regression test suites, so that others do not need to suffer through the compilation process.

Running 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. 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. He sent us one of the packages, which we used as a benchmark: the **meld** visual diff tool.

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

Reproducible computational experiments: In Section 1.1, we described how CDE can make it easier for scientists to package up their experiments so that their colleagues can re-run and build upon them. In our experience, the results of many computational science experiments can be reproduced within CDE packages since the programs are output-deterministic [14], always producing the same outputs (e.g., statistics, graphs) for a given input. For instance, a robotics researcher used CDE to make the experiments for his motion planning paper (**kpiece**) [26] fully-reproducible. Similarly, we helped a social networking researcher create a reproducible package for his genetic algorithm paper (**gadm**) [22].

Deploying computations to cluster or cloud: It can be difficult to get root access to cluster machines. Instead, a user can create a self-contained package using CDE 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 instance, our colleague Peter wanted to use a department-administered 100-CPU cluster to run a parallel image processing job on topological maps (**ztopo**). 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 dataset on his desktop, transferred the package and the complete dataset to the cluster, and then ran 100 instances of it in parallel there.

Similarly, we worked with lab-mates to use CDE to deploy the CPU-intensive **klee** [15] 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 before it will compile.

Submitting executable bug reports: Bug reporting is a tedious manual process: Users submit reports by writing down the steps for reproduction, exact versions of executables and dependent libraries, and maybe attaching an input that triggers the bug. Developers often have trouble reproducing bugs based on these hand-written descriptions and end up closing reports as "not reproducible."

CDE offers an easier and more reliable solution: The bug reporter can simply run the command that triggers the bug under CDE supervision to create a CDE package, send that package to the developer, and the developer can re-run that same command on their machine to reproduce the bug. The developer can also modify the input file and command-line parameters and then re-execute, in order to investigate the bug's root cause.

To show that this technique works, we asked people who recently reported bugs to popular open-source projects to use CDE to create executable bug reports. Three volunteers sent us CDE packages, and we were able to reproduce all of their bugs: one that causes the Coq proof assistant to produce incorrect output (**coq-bug-2443**) [3], one that segfaults the GCC compiler (**gcc-bug-46651**) [4], and one that makes the LLVM compiler allocate an enormous amount of memory and crash (**llvm-bug-8679**) [6].

Since CDE is not a record-replay tool, it is not guaranteed to reproduce non-deterministic bugs. However, at least it allows the developer to run the exact versions of the faulting executables and dependent libraries.

Collaborating on class programming projects: Two users sent us CDE packages they created for collaborating on class assignments. Rahul, a Stanford grad student, was using NLTK [23], a Python module for natural language processing, to build a semantic email search engine (**email-search**) 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 and dependency hell. This meant that they could only run 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 (**vr-osg**).

4.2 CDE package portability

To demonstrate that CDE packages can successfully execute on a wide range of Linux distros and kernel versions, we tested our benchmark packages on popular distros from the past 5 years. We installed fresh copies of these distros (listed with the versions and release dates of their kernels) on a 3GHz Intel Xeon x86-64 machine:

Package name	Origin	Size (MB)	Sep 2006 CentOS 2.6.18	Oct 2007 Fedora 2.6.23	Oct 2008 openSUSE 2.6.27	Sep 2009 Ubuntu 2.6.31	Feb 2010 Mandriva 2.6.33	Aug 2010 Linux Mint 2.6.35
32-bit packages (tested on 32-bit Linux distributions)								
`bio-menace`	2.6.33 P	14	28%	29%	36%	48%	31%	43%
`pads`	2.6.24 U	23	14%	10%	29%	28%	20%	23%
`arachni`	2.6.35 U	27	47%	20%	20%	26%	22%	64%
`ztopo`	2.6.35 U	46	20%	15%	20%	21%	17%	31%
`klee`	2.6.32 D	61	3%	3%	3%	7%	3%	6%
`graph-tool`	2.6.26 D	192	15%	8%	17%	20%	22%	20%
`google-earth`	2.6.24 U	220	5%	4%	3%	2%	2%	2%
64-bit packages (tested on 64-bit Linux distributions)								
`coq-bug-2443`	2.6.32 D	18	7%	7%	7%	9%	9%	9%
`vr-osg`	2.6.35 U	24	27%	FAILED	28%	35%	35%	43%
`llvm-bug-8679`	2.6.35 U	30	7%	7%	7%	10%	8%	12%
`kpiece`	2.6.35 U	38	12%	13%	13%	15%	16%	18%
`arachni`	2.6.35 U	41	17%	17%	15%	20%	20%	41%
`saturn`	2.6.18 C	79	9%	8%	10%	10%	10%	72%
`meld`	2.6.35 U	82	20%	21%	25%	32%	31%	44%
`gcc-bug-46651`	2.6.36 G	125	10%	10%	33%	33%	62%	34%
`graph-tool`	2.6.26 D	209	15%	8%	17%	19%	25%	20%
`gadm`	2.6.18 C	281	3%	2%	2%	2%	2%	20%
`email-search`	2.6.32 U	476	4%	3%	5%	6%	10%	6%

Table 3: CDE benchmark packages, uncompressed package sizes, and percentage of disk space savings due to data deduplication. The 'Origin' column shows the kernel version and distro where a package was created: **U**buntu, **D**ebian, **C**entOS, **G**entoo, **P**uppy Linux. Creators of `arachni` and `graph-tool` provided both 32-bit and 64-bit packages.

- Sep 2006 — CentOS 5.5 (Linux 2.6.18)
- Oct 2007 — Fedora Core 8 (Linux 2.6.23)
- Oct 2008 — openSUSE 11.1 (Linux 2.6.27)
- Sep 2009 — Ubuntu 9.10 (Linux 2.6.31)
- Feb 2010 — Mandriva Free Spring (Linux 2.6.33)
- Aug 2010 — Linux Mint 10 (Linux 2.6.35)

We installed 32-bit and 64-bit versions of each distro and executed our 32-bit benchmark packages (those created on 32-bit distros) on the 32-bit versions, and our 64-bit packages on the 64-bit versions. Although all of these distros reside on one physical machine, none of our benchmark packages were created on that machine: CDE users created most of the packages, and we made sure to create our own packages on other machines. The 'Origin' column of Table 3 shows that packages were created on a variety of distros and kernel versions.

Results: Table 3 shows that out of the 108 configurations we tested (18 CDE packages each run on 6 distros), *all executions succeeded* except for one (`vr-osg` failed on Fedora Core 8 with a known error related to graphics drivers[1]). By 'succeeded' we mean that the programs ran correctly: Batch programs generated identical outputs across distros, regression tests passed, we could interact normally with the GUI programs, and we could reproduce the symptoms of the executable bug reports.

In addition, we were able to successfully execute all of our 32-bit packages on the 64-bit versions of CentOS, Mandriva, and openSUSE (the other 64-bit distros did not support executing 32-bit binaries).

In sum, we were able to use CDE to successfully execute a diverse set of programs (Table 2) 'out of the box' on a variety of Linux distributions from the past 5 years, without performing any installation or configuration.

Comparison against binary installer: To show that the level of portability that CDE enables is substantive, we compare CDE against a representative binary installer for a commercial application. We tried installing and running Google Earth (Version 5.2.1, Sep 2010) on our 6 test distros using the official 32-bit binary installer from Google. Here is what happened on each distro:

[1] OpenSceneGraph online forum discussion: `http://forum.openscenegraph.org/viewtopic.php?t=5653`

- CentOS (Linux 2.6.18) — installs fine but Google Earth crashes upon start-up with variants of this error message repeated several times, because the GNU Standard C++ Library on this OS is too old:

```
/usr/lib/libstdc++.so.6:
version 'GLIBCXX_3.4.9' not found
(required by ./libgoogleearth_free.so)
```

- Fedora (Linux 2.6.23) — same error as CentOS
- openSUSE (Linux 2.6.27) — installs and runs fine
- Ubuntu (Linux 2.6.31) — installs and runs fine
- Mandriva (Linux 2.6.33) — installs fine but Google Earth crashes upon start-up with this error message because a required graphics library is missing:

```
error while loading shared libraries:
libGL.so.1: cannot open shared object
file: No such file or directory
```

- Linux Mint (Linux 2.6.35) — installer program crashes with this cryptic error message because the XML processing library on this OS is *too new* and thus incompatible with the installer:

```
setup.data/setup.xml:1: parser error :
  Document is empty
setup.data/setup.xml:1: parser error :
  Start tag expected, '<' not found
Couldn't load 'setup.data/setup.xml'
```

To recap, on 4 out of our 6 test distros, a binary installer for the fifth major release of Google Earth (v5.2.1), a popular commercial application developed by a well-known software company, failed in its *sole goal* of allowing the user to run the application, despite advertising that it should work on any Linux 2.6 machine.

In contrast, once we were able to install Google Earth on just *one machine* (Dell desktop running Ubuntu 8.04), we ran it under CDE supervision to create a self-contained package, copied the package to all 6 test distros, and successfully ran Google Earth on all of them without any installation or configuration.

Absolute limit of portability: A CDE package can only run on a machine whose hardware architecture and Linux kernel version are compatible with the executables and shared libraries within the package. Every Linux executable and library indicates the architecture and minimum kernel version on which it can run (the `file` command shows this information). These hard limits are set at compile-time and cannot be overcome by anything except for emulation of the outdated ABI and system calls.

Some libraries (on distros like Gentoo) are compiled with aggressive machine-specific optimizations that hinder portability. However, most Linux distros our users have encountered contain libraries compiled with fairly generic and portable x86 optimizations. We have shown that CDE allows diverse types of programs to run on a variety of popular Linux distros within a 5-year range. Although we cannot predict the future, our intuition is that packages created today will continue to run fine on Linux distros from several years in the future, since kernel developers place high priority on maintaining backwards compatibility in the kernel-to-user ABI [5]. For reference, the `saturn` and `gadm` packages were created on 2006 Linux kernels and run fine on 2010 kernels. Users who desire greater portability or 'future-proofing' can pair CDE with a virtual machine or processor emulator.

4.3 CDE package size & data deduplication

The 'Size' column of Table 3 shows that our benchmark packages range from 14 to 476 MB uncompressed (file compression can make them 2X–5X smaller). Since disk space is plentiful on modern desktop machines and our users have not yet complained about package sizes, we have not attempted to optimize CDE for space usage.

However, to allay potential concerns about package sizes, we implemented a data deduplication algorithm and evaluated its impact on our benchmarks. Each CDE package contains all files the enclosed program needs in order to run on any contemporary x86-Linux machine, but intuitively, some of those files must already exist on any given target machine. For example, almost all CDE packages contain `libc` (GNU Standard C Library), and some variant of `libc` must exist on a target machine.

Our data deduplication algorithm searches for files that exist in both the CDE package and the target machine's system directories (e.g., `/lib`, `/usr`). For every pair of identical files, our algorithm deletes the copy within the CDE package and replaces it with a hard link to the copy in the system directory. For non-identical files with similar filenames (e.g., `libc-2.5.so` and `libc-2.8.so` are both variants of `libc`), our algorithm takes a binary diff using `bsdiff`, deletes the copy within the CDE package, and replaces it with the delta. `bsdiff` is an efficient binary diff algorithm optimized for executable files; the Google Chrome team uses it as the basis for creating binary security patches [12].

Results: Table 3 shows that disk space savings range widely from 2% to 72%, depending on how many files each Linux distro happens to share in common with the package's contents. The mean disk space savings was 18% (median was 15%), which indicates that the majority of package contents are files that do not already exist on most modern Linux systems. Deduplication can only be done after a package arrives on a particular target machine, not at package creation time, since each machine contains different files. In sum, deduplication might be

Command	Native time	CDE slowdown pack	CDE slowdown exec	Syscalls per sec
`gadm` (algorithm)	4187s	0%†	**0%**†	19
`pads` (inferencer)	18.6s	3%†	**1%**†	478
`klee`	7.9s	31%	**2%**†	260
`gadm` (make plots)	7.2s	8%	**2%**†	544
`gadm` (C++ comp)	8.5s	17%	**5%**	1459
`saturn`	222.7s	18%	**18%**	6477
`google-earth`	12.5s	65%	**19%**	7938
`pads` (compiler)	1.7s	59%	**28%**	6969

Table 4: Quantifying run-time slowdown of CDE **pack**age creation and **exec**ution within a package. Each entry reports the mean taken over 5 runs; standard deviations are negligible. Slowdowns marked with † are *not* statistically significant at $p < 0.01$ according to a t-test.

useful for space-conscious users, especially those who wish to run multiple CDE packages on one machine.

4.4 CDE run-time slowdown

To quantify the slowdown that CDE incurs, we measured running times for executing these commands in the five CDE packages that we created (first column in Table 4):

- `pads` — Compile a PADS [17] specification into C code (the 'compiler' row in Table 4), and infer a specification from a data file (the 'inferencer' row).
- `gadm` — Reproduce the GADM experiment [22]: Compile its C++ source code ('C++ comp'), run genetic algorithm ('algorithm'), and use the R statistics software to visualize output data ('make plots').
- `google-earth` — Measure startup time by launching it and then quitting as soon as the initial Earth image finishes rendering and stabilizes.
- `klee` — Use Klee [15] to symbolically execute a C target program (a STUN server) for 100,000 instructions, which generates 21 test cases.
- `saturn` — Run the regression test suite, which contains 69 tests (each is a static program analysis).

We measured the following on a Dell desktop (2GHz Intel x86, 32-bit) running Ubuntu 8.04 (Linux 2.6.24): number of seconds it took to run the original command ('Native time'), percent slowdown vs. native when running a command with `cde` to create a package ('pack'), and percent slowdown when executing the command from within a CDE package with `cde-exec` ('exec'). We ran each benchmark five times under each condition and report mean running times. We used an *independent two-group t-test* [16] to determine whether each slowdown was statistically significant (i.e., whether the means of two sets of runs differed by a non-trivial amount).

Results: Table 4 shows that the more system calls a program issues per second, the more CDE causes it to slow down. This makes sense because the kernel must context switch to the CDE process during every syscall. Creating a CDE package ('pack' column) is slower than executing a program within a package ('exec' column) because CDE must create new sub-directories and copy files into the package. The 'exec' column slowdowns are shown in **bold** since they are more important for our users: A package is only created once but executed multiple times.

CDE execution slowdowns ranged from negligible (not statistically significant) to ∼30%, depending on system call frequency. As expected, CPU-bound workloads like the `gadm` genetic algorithm and the `pads` inferencer machine learning algorithm had almost no slowdown, while those that were more I/O-intensive (e.g., the `pads` compiler) had the largest slowdowns.

When using CDE to run GUI applications, we did not notice any loss in interactivity due to the slowdowns. When we navigated around the 3D maps within the `google-earth` GUI, we felt that the CDE-packaged version was just as responsive as the native version. When we ran GUI programs from CDE packages that users sent to us (the `bio-menace` game, `meld` visual diff tool, and `vr-osg`), we also did not perceive any visible lag.

4.5 Importance of dynamic tracking

To show the importance of dynamic (run-time) dependency tracking, we compare CDE against a simple but representative static analysis. We wrote a script that runs the Linux `ldd` and `strings` utilities on an executable file to find all string constants representing shared libraries on which it depends, and then recursively runs `ldd` and `strings` on those libraries and their dependencies until the set of files converges. Although this basic static technique only finds libraries named by constant strings, it represents what people actually do in practice, since it automates the tedious manual process of "chasing down and copying over dependent libraries" that folk wisdom (e.g., blog posts and forums) suggests as the way to transport Linux binaries across machines [8]. It is difficult in general for a static analysis to model dynamically-generated strings; we know of no static dependency gathering tool that works in this way.

In contrast, since CDE actually executes the target executable in addition to statically searching for constant strings, CDE can find dependencies on shared libraries (and all other files) named by dynamically-generated strings, in addition to those that a static technique finds.

Package name	# shared library files Total	Statically found		# total files
google-earth	82	3	(4%)	243
graph-tool	149	9	(6%)	1909
meld	93	8	(9%)	507
arachni	48	6	(13%)	381
gcc-bug-46651	13	2	(15%)	114
email-search	138	28	(20%)	3052
gadm	18	4	(22%)	268
saturn	16	8	(50%)	455
pads	9	5	(56%)	150
ztopo	59	35	(59%)	164
vr-osg	39	28	(72%)	57
bio-menace	27	26	(96%)	107
coq-bug-2443	3	3	(100%)	29
klee	6	6	(100%)	18
llvm-bug-8679	8	8	(100%)	14
kpiece	30	30	(100%)	45

Table 5: Number of total shared library files in each CDE package, and number (and percent) of those found by a simple static analysis of binaries and dependent libs. Rightmost column is number of total files in package.

Results: Table 5 shows that in all but four benchmarks, the static technique found fewer libraries than CDE. Thus, it cannot be used to create a portable package since the program will fail if *even one library is missing*. (For similar reasons, static linking when compiling will not work either.) Even on the four benchmarks where the static technique found all required libraries, a user would still have to manually insert all input, configuration, and other data files into the package. The '# total files' column in Table 5 shows that packages contain dozens to thousands of files, often scattered across many directories, so this process can be tedious and error-prone.

Table 5 also shows why it is necessary for CDE to dynamically track dependencies, since most benchmarks load libraries that are not named by constant strings. At one extreme, the four benchmarks where the static technique performed worst (`google-earth`, `graph-tool`, `meld`, `arachni`) consist of scripts written in interpreted languages. The interpreter dynamically loads libraries and invokes other executables based on the contents of those scripts. A static analysis of the interpreter's executable (e.g., Python) can only find the libraries needed to start up the interpreter; however, the majority of libraries that each script requires are indirectly specified within the script itself. For example, executing a simple line of Python code "`import numpy`" in `graph-tool` causes the Python interpreter to import the NumPy numerical analysis module, which consists of 23 shared libraries scattered across 7 sub-directories. The interpreter dynamically generates the pathnames of those 23 libraries by processing strings read from environment variables and config files, so it is unlikely that any static analysis could ever generate those 23 pathnames and find the corresponding libraries. CDE easily finds those libraries since it monitors actual execution.

4.6 Running on different inputs

Since users create CDE packages by executing the target program on one or a few inputs, one might wonder how far programs executed from within packages can diverge from their original execution paths before they fail. In general, *no automatic tool* (static or dynamic) can find all the dependencies required to execute all possible program paths, since that problem is undecidable. Similarly, it is also impossible to automatically quantify how 'complete' a CDE package is or 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.

Despite the lack of formal guarantees, we feel that execution within CDE packages can diverge from their original path(s) to a degree that our users find to be practical. Packaged execution can diverge as long as it does not load any new libraries (or configuration files) that were not loaded by the original execution(s). We observed that many programs load the majority of required libraries at start-up, regardless of what paths are executed later.

Most of our benchmark packages were created by running one representative command (or a regression test suite); we list all the commands on our benchmark webpage [1]. Here are our experiences with executing those packages on different inputs using `cde-exec`:

- We can make compiler-like programs (e.g., `pads`, `saturn`, `coq`, `gcc`, `llvm`) process any legal input file, not just the files used to create the packages.
- We can reproduce the original results of computational experiments (e.g., `gadm`, `kpiece`, `email-search`), but more importantly, we can adjust parameters, recompile, and execute variants of those experiments to explore related hypotheses.
- We can interact normally with GUI applications like `google-earth`, `meld`, and `vr-osg`. Of course, it is possible that clicking some obscure nested submenu option causes these programs to load libraries not in the package, but common use cases work fine.
- We can play the `bio-menace` game following any path we wish, not just the path taken by its package creator, since the game contents are in a single file.

- We can use the `arachni` security tool to scan arbitrary URLs and the `graph-tool` Python library to do graph operations. Their creators had enough confidence in CDE to make those packages available for download on their respective project websites.

Since a CDE package is just an ordinary directory tree, it is easy for a creator to directly add more files into the package. A more convenient way to add files is to simply execute the program additional times using `cde` to exercise more paths that users might want to run. In sum, although it is impossible for any automatic tool to create complete self-contained binary packages, CDE works well in practice and makes it convenient for creators to augment their packages with additional files as needed.

5 Conclusion

We presented CDE, an open-source tool [2] that 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 or configuration. Hundreds of people in both academia and industry have used CDE to avoid the usual pains associated with software distribution and installation.

Our experiments showed that CDE packages are portable across popular Linux distros from the past 5 years, that size and run-time overheads are acceptable, and that dynamic dependency tracking is necessary and works well in practice. All of the benchmark packages used in our experiments are available online [1].

Acknowledgments

Thanks to Fernando Perez for the serendipitous discussion of reproducible research that planted the seeds of the idea for CDE, to Richard Spillane for sharing his Goanna code [25], to Imran Haque for the Slashdot publicity, to our users for bug reports and feedback, to {`ridddlr`, `paboonst`, `cbird`, `TomZ`, `ewencp`, `ihaque`, `daramos`} for editorial help, and to the NSF fellowship for funding Philip's graduate studies.

References

[1] CDE benchmark packages and documentation, `http://www.stanford.edu/~pgbovine/cde-usenix.html`.

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

[3] Coq proof assistant: Bug 2443, `http://coq.inria.fr/bugs/show_bug.cgi?id=2443`.

[4] GCC compiler: Bug 46651, `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=46651`.

[5] Linux: Ensuring Binary Compatibility, `http://kerneltrap.org/node/4006`.

[6] LLVM compiler: Bug 8679, `http://llvm.org/bugs/show_bug.cgi?id=8679`.

[7] Mac OS X Bundle Programming Guide: Introduction, `http://developer.apple.com/library/mac/#documentation/CoreFoundation/Conceptual/CFBundles/Introduction/Introduction.html`.

[8] Tutorial: Static, Shared Dynamic and Loadable Linux Libraries, `http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html`.

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

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

[11] VMware ThinApp User's Guide, `http://www.vmware.com/pdf/thinapp46_manual.pdf`.

[12] Google Chrome software updates: Courgette, `http://www.chromium.org/developers/design-documents/software-updates-courgette`.

[13] AIKEN, A., BUGRARA, S., DILLIG, I., DILLIG, T., HACKETT, B., AND HAWKINS, P. An overview of the Saturn project. PASTE '07, ACM, pp. 43–48.

[14] ALTEKAR, G., AND STOICA, I. ODR: output-deterministic replay for multicore debugging. SOSP '09, ACM, pp. 193–206.

[15] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI '08, USENIX Association, pp. 209–224.

[16] CHAMBERS, J. M. *Statistical Models in S*. CRC Press, Inc., Boca Raton, FL, USA, 1991.

[17] FISHER, K., AND GRUBER, R. PADS: a domain-specific language for processing ad hoc data. PLDI '05, ACM, pp. 295–304.

[18] GARFINKEL, T. Traps and pitfalls: Practical problems in system call interposition based security tools. NDSS '03.

[19] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. NDSS '04.

[20] JAIN, K., AND SEKAR, R. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. NDSS '00.

[21] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. SIGMETRICS '10, pp. 155–166.

[22] LAHIRI, M., AND CEBRIAN, M. The genetic algorithm as a general diffusion model for social networks. In *Proc. of the 24th AAAI Conference on Artificial Intelligence* (2010), AAAI Press.

[23] LOPER, E., AND BIRD, S. NLTK: The Natural Language Toolkit. In *In ACL Workshop on Effective Tools and Methodologies for Teaching NLP and Computational Linguistics* (2002).

[24] SAITO, Y. Jockey: A user-space library for record-replay debugging. In *AADEBUG* (2005), ACM Press, pp. 69–76.

[25] SPILLANE, R. P., WRIGHT, C. P., SIVATHANU, G., AND ZADOK, E. Rapid file system development using ptrace. In *Experimental Computer Science* (2007), USENIX Association.

[26] SUCAN, I. A., AND KAVRAKI, L. E. Kinodynamic motion planning by interior-exterior cell exploration. In *Int'l Workshop on the Algorithmic Foundations of Robotics* (2008), pp. 449–464.