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

Philip J. Guo and Dawson Engler
Stanford University

Abstract

It can be painfully hard 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. Creating a CDE package is completely automatic, and running programs within a package requires no installation, configuration, or root permissions. Hundreds of people in 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*. Finally, the user might lack the permissions or willingness to risk installing software packages as root in the first place, a common occurrence on corporate machines and clusters administered by IT staff.

To alleviate these frustrations, we have created an open-source tool named CDE that monitors program execution using `ptrace` and 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 [1].

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.

The main limitation of CDE is that it is not guaranteed to find all the dependencies required for a complete package, so it is up to the user to insert additional files if necessary. Also, packages are only portable across machines with a compatible architecture and Linux kernel version. Despite these limitations, CDE has been downloaded over 2,000 times, and we have received hundreds of emails from users who have used it to quickly test and deploy software without installing any dependencies.

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).

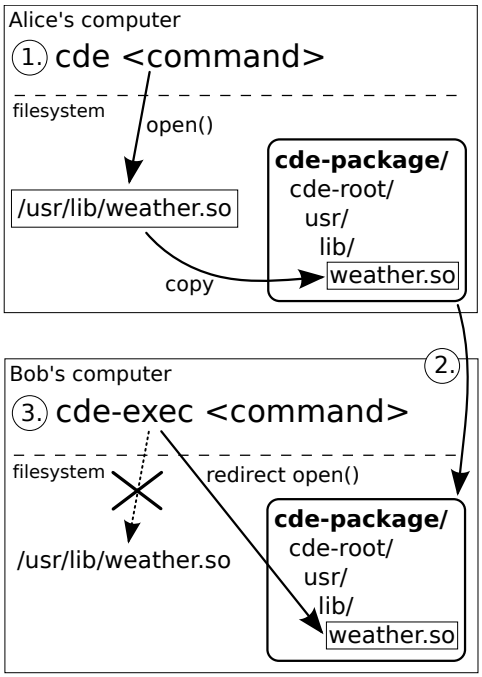

Figure 1: Example use of CDE: 1.) Alice runs her command with `cde` to create a package, 2.) Alice sends package to Bob's computer, 3.) Bob runs command with `cde-exec`, which redirects file accesses into package.

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

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 re-run 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.

3 Implementation

CDE uses the Linux `ptrace` system call to monitor the target program's processes, read and 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. The same ideas could be used to implement CDE for other architectures or operating systems.

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. After the kernel finishes executing a syscall that takes a file path string as an argument (the 'File path access' category in Table 1) 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.

Prior to copying a file into the package, `cde` creates all necessary sub-directories and symbolic links to exactly mirror that file's location. If a file is a symlink, then both it and its target must be copied into the package.

If the copied file is an ELF binary, then `cde` searches its contents for constant strings that are filenames and then recursively copies those files into the package. This simple hack works well in practice to partially overcome CDE's limitation of only being able to gather dependencies on executed paths, since many binaries dynamically load libraries named by constant strings.

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 IPC 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 bin & linker into pkg	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.

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.

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/. By doing so, cde-exec creates a chroot-like sandbox that fools the program into 'believing' that it is executing on the original machine.

To reliably rewrite syscall arguments, cde-exec redirects the pointer to the argument's buffer. When a target process first makes a syscall, cde-exec forces it to make another syscall to attach a 16KB shared memory segment (a trick from [16]). Prior to every file path access syscall, cde-exec computes and writes the redirected path into shared memory and uses ptrace to mutate the syscall's argument, stored in a register, to point to the start of the shared memory segment in the target's address space.

Spoofing current working directory: At the completion of the getcwd syscall, cde-exec mutates (truncates) its return value string to eliminate all absolute path components up to and including cde-root/.

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 rather than directly executing the binary. The dynamic linker on one 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, even a trivial "hello world" binary compiled on one distro (e.g., Ubuntu with Linux 2.6.35) will not run on an older one (e.g., Knoppix with Linux 2.6.17).

Ignoring files and environment vars: By convention, Linux directories like /dev, /proc, and /sys contain pseudo-files that do not make sense to include in a CDE package. To improve package portability, we have manually created a user-customizable blacklist of a dozen directories, files, and environment variables for CDE to ig-

nore. `cde` will not copy ignored files (or vars) into a package, and `cde-exec` will not redirect their paths and instead access the real versions on the target machine.

4 Limitations

Executing a command within a CDE package will fail if:

- the arguments or input change to make the program load a new file (e.g., library, config file) that the original execution did not load. 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. However, since 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 add additional files into the same `cde-package/`.
- the Linux kernel or hardware architecture on the target machine is incompatible with the binaries in the package. Mainstream distros contain libraries that are forwards- and backwards-compatible over several years. For example, the standard libs on 2010-era Ubuntu work on distros from as old as 2006 ($\geq$ 2.6.15 kernel), and the libs on 2007-era Fedora work on 2004-era distros ($\geq$ 2.6.9). Also, our intuition is that packages created today will run fine on Linux 2.6 distros from several years in the future, since kernel developers place high priority on maintaining backwards compatibility in the kernel-to-user ABI. Users who desire greater portability or 'future-proofing' can embed CDE packages within virtual machine or processor emulator images.

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.

5 Real-world use cases

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) [1]. We have exchanged hundreds of emails with CDE users and discovered six salient use cases as a result of our discussions. For our experiments (see Table 2), we used representative packages from each use case category (names in **`bold`**).

Distributing research software: The creators of two research tools — the **`arachni`** web app. security scanner [5] and the **`graph-tool`** math library [6] — used CDE to create portable binary packages that they uploaded to their project websites, so that their users do not have to go through the anguish of compiling them from source.

In addition, we used CDE to create portable binary packages for two of our Stanford colleagues' research tools, which were originally distributed as hard-to-compile source code tarballs: **`pads`** [11] and **`saturn`** [8].

Running software on incompatible distros: Even production-quality software might be hard to install on Linux distros with older kernel or library versions. For example, a Cisco engineer 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 (e.g., the **`meld`** visual text diff tool), 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 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: A fundamental tenet of science is that colleagues should be able to reproduce the results of one's experiments. Recently, some science journals and CS conferences are starting to encourage 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 to set up all of the (often-undocumented) dependencies required to re-run experiments.

Scientists can run the experiment once on their machine with CDE to create a package, and colleagues can run that package on any contemporary Linux machine to repeat the experiment. A robotics researcher used CDE to make the experiments for his motion planning paper (**`kpiece`**) [17] fully-reproducible. Similarly, we helped a social networking researcher create a reproducible package for his genetic algorithm paper (**`gadm`**) [15].

Deploying computations to cluster or cloud: 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** [10] bug finding tool from the desktop to Amazon's EC2 cloud computing service without needing to compile Klee on the cloud machines.

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 solution: The 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. Three bug reporters 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**) [2], one that segfaults the GCC compiler (**gcc-bug**) [3], and one that makes the LLVM compiler allocate an enormous amount of memory and crash (**llvm-bug**) [4].

Collaborating on class projects: Rahul, a Stanford grad student, was using the NLTK natural language processing module 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 machines due to conflicting library versions and dependency hell, so they only had one runnable copy. Rahul used CDE to create 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. Similarly, an undergrad used CDE to collaborate on and demo his virtual reality project (**vr-osg**).

6 Summary of experimental results

Due to space constraints, we summarize our main experimental results. Full details are in our tech report [12].

Package portability: To demonstrate that CDE packages can successfully execute on a range of Linux variants, we tested our benchmark packages on six popular distros, listed with the versions and release dates of their kernels:

Package name	Origin	Num libs	Slowdown
Distribute research software			
`arachni` [5]	2.6.35	48 (6)	
`graph-tool` [6]	2.6.26	149 (9)	
`pads` [11]	2.6.24⋆	9 (5)	28%
`saturn` [8]	2.6.18⋆	16 (8)	18%
Run production software on incompatible distros			
`meld`	2.6.35	93 (8)	
`bio-menace`	2.6.33	27 (26)	
`google-earth`	2.6.24 ⋆	82 (3)	19%
Create reproducible computational experiments			
`kpiece` [17]	2.6.35	30 (30)	
`gadm` [15]	2.6.18 ⋆	18 (4)	5%
Deploy computations to cluster or cloud			
`ztopo`	2.6.35	59 (35)	
`klee` [10]	2.6.32⋆	6 (6)	2%
Submit executable bug reports			
`coq-bug` [2]	2.6.32	3 (3)	
`gcc-bug` [3]	2.6.36	13 (2)	
`llvm-bug` [4]	2.6.35	8 (8)	
Collaborate on class programming projects			
`email-search`	2.6.32	138 (28)	
`vr-osg`	2.6.35	39 (28)	

Table 2: CDE packages by category. The 'Origin' column shows the kernel version where a package was created, and a star⋆ means it was created by the first author. The 'Num libs' column shows number of shared libraries (and number of statically-discoverable libs in parens).

1. CentOS 5.5 (Linux 2.6.18, Sep 2006)
2. Fedora Core 8 (Linux 2.6.23, Oct 2007)
3. openSUSE 11.1 (Linux 2.6.27, Oct 2008)
4. Ubuntu 9.10 (Linux 2.6.31, Sep 2009)
5. Mandriva Free Spring (Linux 2.6.33, Feb 2010)
6. Linux Mint 10 (Linux 2.6.35, Aug 2010)

Out of the 108 configurations we tested (18 CDE packages[1] each run on 6 Linux distros), *all executions succeeded* except for one (`vr-osg` failed on Fedora Core 8 with a known graphics-related error). By 'succeeded' we mean that the programs appeared to run correctly: Batch programs generated identical outputs across distros, and we could interact normally with GUI programs.

Necessity of dynamic tracking: We compared CDE against a static analysis that recursively runs the Linux `ldd` and `strings` utilities on executables files and libraries to find all string constants representing dependent

[1] Two of our benchmarks had both 32-bit and 64-bit versions.

libraries. Although this technique is simple, 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 suggests as the way to transport programs across machines.

The 'Num libs' column in Table 2 shows that in all but four benchmarks, the static technique found fewer libraries than CDE (the number of statically-discoverable libraries shown in parentheses). 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. This is why CDE's static+dynamic dependency tracking is necessary.

Run-time slowdown: We informally evaluated slowdowns on the five CDE packages we created (those marked with * in Table 2). Executing those programs within CDE packages were $2\% - 28\%$ slower than executing natively. The more system calls a program issues per second, the more CDE causes it to slow down, since the kernel must context switch to the CDE process during every syscall. We have heard that `ptrace` interposition can cause slowdowns of 10X or more, but we have not yet performed a rigorous performance stress test.

7 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. PDS is a prototype tool for creating self-contained Windows apps, which requires the user to manually specify a dependency list [9].

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 [7]. 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 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.

Finally, system call interposition using `ptrace` is a well-known technique that has been used for implementing tools such as secure sandboxes [13], record-replay systems [14], and user-level filesystems [16].

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 [16], to Imran Haque for the Slashdot publicity, to our users for their bug reports and feedback, and to {`ridddlr`, `paboonst`, `cbird`, `TomZ`, `ewencp`, `ihaque`, `daramos`} for editorial help. This research was supported by the NSF Graduate Research Fellowship and the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024.

References

[1] CDE project home page, `http://www.pgbovine.net/cde.html`.

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

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

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

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

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

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

[8] 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.

[9] ALPERN, B., AUERBACH, J., BALA, V., FRAUENHOFER, T., MUMMERT, T., AND PIGOTT, M. PDS: a virtual execution environment for software deployment. VEE '05, ACM, pp. 175–185.

[10] 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.

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

[12] GUO, P. J., AND ENGLER, D. CDE: Using system call interposition to automatically create portable software packages. Stanford University Computer Science Technical Report 2011-01.

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

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

[15] 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.

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

[17] 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.