Practical and effective sandboxing for non-root users

Taesoo Kim and Nickolai Zeldovich
MIT CSAIL

Abstract

MBOX is a lightweight sandboxing mechanism for non-root users in commodity OSes. MBOX's sandbox usage model executes a program in the sandbox and prevents the program from modifying the host filesystem by layering the sandbox filesystem on top of the host filesystem. At the end of program execution, the user can examine changes in the sandbox filesystem and selectively commit them back to the host filesystem. MBOX implements this by interposing on system calls and provides a variety of useful applications: installing system packages as a non-root user, running unknown binaries safely without network accesses, checkpointing the host filesystem instantly, and setting up a virtual development environment without special tools. Our performance evaluation shows that MBOX imposes CPU overheads of 0.1–45.2% for various workloads. In this paper, we present MBOX's design, efficient techniques for interposing on system calls, our experience avoiding common system call interposition pitfalls, and MBOX's performance evaluation.

1 Introduction

In this paper, we present MBOX, a lightweight sandboxing mechanism for non-root users in commodity OSes. MBOX provides two attractive benefits as a sandbox; first, protection of the host filesystem from modifications by sandboxed programs; and second, flexibility in controlling the execution of the sandboxed program.

To protect the host system, MBOX overlays the host filesystem with a sandbox filesystem and confines all modifications made by the sandboxed program to the sandbox filesystem. As MBOX stores the sandbox filesystem as a regular directory in the host filesystem, users can use standard Unix tools to examine the modifications, commit them back to the host filesystem, or even archive them for later use as a layered sandbox filesystem for other programs.

MBOX implements the layered sandbox filesystem with system call interposition. By interposing on system calls, MBOX can provide additional features missing from commodity OSes, which are useful to non-root users in a variety of real-world scenarios: enabling non-root users to install system packages with standard package managers, checkpointing the whole filesystem instantly, running unknown binaries safely without network access, and setting up virtual development environments without special tools. More importantly, all use cases neither require root privilege nor require modification to the OS kernel and applications.

Overview MBOX aims to make running a program in a sandbox as easy as running the program itself. For example, one can sandbox a program (say `wget`) by running as below:

```
$ mbox -- wget google.com
...
Network Summary:
 > [11279] -> 173.194.43.51:80
 > [11279] Create socket(PF_INET,...)
 > [11279] -> a00::2607:f8b0:4006:803:0
 ...
Sandbox Root:
 > /tmp/sandbox-11275
 >  N:/tmp/index.html
[c]ommit, [i]gnore, [d]iff, [l]ist, [s]hell, [q]uit ?>
```

`wget` is a utility to download files from the web. In the above example, MBOX prevents `wget` from writing the downloaded `index.html` to the host filesystem, and instead redirects it to the sandbox filesystem (stored at `/tmp/sandbox-11275`). Since the sandbox filesystem is just a regular directory in the host filesystem, the user can use standard Unix tools to perform operations on the files modified by the program. For example, the user can commit the `index.html` file back to the place where `wget` would have downloaded the file if it was not sandboxed.

The advantages of using MBOX come from the fact that we can restrict the sandboxed program or change its behavior while protecting the host filesystem. For example, we can enable interesting use cases like monitoring where `wget` connects to and what it downloads, or restricting its remote network accesses (see §2).

Contributions In this paper, we

- describe the MBOX abstraction, usage model, and a wide range of use cases.
- present `seccomp/BPF` as an efficient system call interposition technique, and our experience with avoiding common system call interposition mistakes [4].
- implement and evaluate these ideas in MBOX, a Linux-based open source tool that requires no changes to the OS kernel or applications.

Outline §2 provides practical use cases of MBOX. §3 describes its design. §4 explains MBOX's interposition

technique. §5 discusses its implementation, §6 evaluates, §7 compares MBOX with related work, and §8 concludes.

2 Use cases

We motivate the usefulness of MBOX by describing five real-world use cases that are difficult to achieve in commodity OSes as a non-root user.

2.1 Installing packages without root access

```
$ mbox -R -- apt-get install git
  (-R: emulate a fakeroot environment)
```

Installing packages requires root privilege in Linux because normal users do not have write access to system directories such as `/bin` and `/lib`; so, to install a package, non-root users need to perform tedious jobs like resolving dependencies manually and compiling source code, even though package managers already perform these jobs. With MBOX, users can instead install packages with standard package managers by running them in a sandbox with a writable sandbox filesystem. As package managers often check for root privilege, MBOX optionally emulates a root-like environment (fakeroot) so users can execute them without any modification. After installing a package with MBOX, the sandbox filesystem contains not only newly installed files, but also the corresponding package databases, separate from the host filesystem. Users, therefore, can even install or remove packages by reusing the same sandbox filesystem (see §2.4). We tested that MBOX supports Ubuntu's `apt-get`, Debian's `dpkg`, and Python's `pip` package managers.

2.2 Running unknown binary safely

```
$ mbox -n -- wget google.com
  (-n: disable remote network accesses)
```

When running unknown binaries, users can protect the host filesystem from modification by running them with MBOX. However, if these binaries misbehave or are compromised, they still can access a user's private data and disclose it to attackers. To prevent this, MBOX provides a way to restrict or monitor remote network accesses of sandboxed processes. If users want to restrict network accesses, MBOX blocks all socket-related system calls; for example, the above command kills `wget` at the first `socket()` system call. However, by default, MBOX interprets socket-related system calls and summarizes network activity, as in the `wget` example in §1.

2.3 Checkpointing filesystem

```
$ mbox -i -- sh
  (-i: enable interactive commit-mode)
```

Using MBOX, one can instantly branch out a new filesystem from the current host filesystem by running a new shell. The shell and all subsequent processes created from the shell run in the same sandbox, and share the same layered filesystem view. For example, editing `emacs` configuration files often requires killing and rerunning `emacs` to check if it works with the new configuration. When it fails with an error, we might need to run vanilla `emacs` to continue fixing the error. With MBOX, one can checkpoint the host filesystem and edit configuration files with `emacs` running in the sandbox; `emacs` instances on the host system still function correctly, even if the edited file has an error. When done with editing, users can commit the modified configuration files to the host filesystem, revert them by discarding changes, or stash them for later use. These workflows are what make users feel comfortable when using SCM tools like Git; with MBOX, users get similar safety and convenience for filesystem data.

2.4 Build/development environment

```
$ mbox -r outdir -- make
  (-r dir: specify a sandbox directory)
```

When building a project's source tree, we often see the directory entangled with both original source files and generated object files. By running a build script with MBOX, we can redirect all generated object files to the sandbox filesystem; also, cleaning up the project directory (say `make clean`) becomes a simple `rm -rf outdir`. Combined with package installations (§2.1), any user can conveniently setup a development environment that is safely separated from the system libraries. For example, without using `virtualenv` for Python and `cabal-dev` for Haskell, we can create virtual environments with the `pip` and `cabal` tools that major distributions come with.

2.5 Profile-based sandbox

```
$ mbox -p build.prof -- ./configure
  (-p prof: enable profile-based policy)
```

MBOX supports another important use case poorly supported by commodity OSes. In Unix-like OSes, a process created by a user runs with that user's privilege, and can access the user's private files. In some cases, the process needs access to the user's files to do useful work; however, often there are cases where the user does not want to expose sensitive data to the process. For example, when a user executes a `./configure` script, she does not want the script to read her private ssh key stored in the `$HOME/.ssh` directory. With MBOX, users can easily hide private directories, and allow access to only the necessary parts of a filesystem by describing them as below.

```
# build.prof
[fs]
    allow: .
    hide: ~
```

If a user runs the `./configure` script with the above profile, MBOX hides the user's home directory yet allows access to the current working directory. Therefore, the script cannot steal the user's private files, but can still

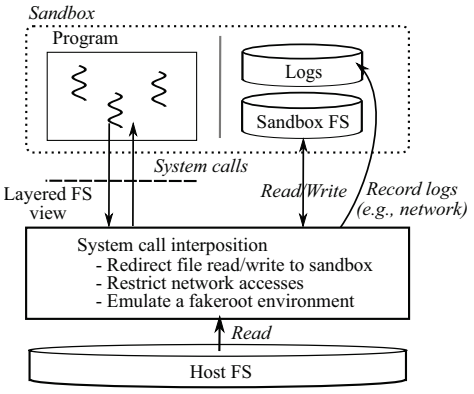

Figure 1: Overview of MBOX's design. MBOX interposes on a sandboxed program's system calls to provide a sandbox filesystem overlaid on the host filesystem; to restrict network accesses; and to emulate a fakeroot environment.

configure properly by accessing the system libraries and header files. In addition, for scripts that are never expected to access the network, users can additionally specify an option to restrict network accesses (§2.2).

3 MBOX abstraction

When a user runs a program with MBOX, MBOX creates a layered sandbox filesystem, where all modifications by the program take place, on top of the host filesystem. The host filesystem remains intact and is never modified by the sandboxed program. When the sandboxed program terminates, users can examine modified files in the sandbox filesystem, and commit them back to the host filesystem if they want. Since the sandbox filesystem is stored in persistent storage, users have complete control over files and directories afterward, and can even reuse them later as a sandbox layer of other programs. We call this usage model the MBOX abstraction. Figure 1 provides an overview of the MBOX design.

3.1 Layered filesystems

Unlike traditional filesystems in which every process has the same namespace, MBOX needs to provide a private filesystem to each process running in different sandboxes. MBOX stacks a private filesystem layer on top of the host filesystem, and provides a logically unified view of both filesystems to a sandboxed program. We call the private filesystem layer, where all modification happens by the program, the *sandbox filesystem*, and call both the sandbox and host filesystems together the *layered filesystem*. To provide a layered filesystem, MBOX interposes on system calls of a sandboxed program. On every system call entry, MBOX decides which system call arguments should be rewritten so that changes by the system call redirect to the sandbox filesystem, rather than affecting the host filesystem.

Copy-on-write The sandbox filesystem is created with no content when a user executes a program with MBOX. Since the sandbox filesystem is empty, all reads by the program will be forwarded to the host filesystem. Once the sandboxed program writes to a file, the sandbox filesystem will contain the modified file and subsequent reads will be redirected to the sandbox filesystem. Thus, the application running inside the sandbox is able to access the modified file and works as it would without the sandbox. The layered filesystem in effect implements copy-on-write: MBOX duplicates the file into the sandbox filesystem and protects the original file from modifications.

Persistent storage The sandbox filesystem is not a filesystem, but is a regular directory in the host filesystem, so it can persist even after the sandboxed program terminates. The persistent sandbox gives users more freedom to examine, archive, and even duplicate the sandbox filesystem, as normal files and directories, with familiar utilities. Also, users can reuse the previous sandbox filesystem as a sandbox layer of any other program, so that users can consider the layered filesystem persistently branched out of the host filesystem, yet easy to discard.

3.2 Committing changes

When a sandboxed program terminates, users can commit modified files back to the host filesystem with tools that MBOX provides. To help users decide what files to commit, MBOX allows the user to check the differences of files in host and sandbox filesystems before committing.

When committing a modified file back to the host filesystem, the original file that the sandbox branched out from might have been changed by programs running on the host filesystem. Faced with such concurrent modifications to the same file in both the host and the sandbox filesystem, MBOX flags a conflict, and requires the user to decide how to merge the changes, much like any version control system.

To detect conflicts, MBOX records a hash of the original file contents when creating a copy of the file in the sandbox filesystem, and checks if the contents of the file in the host filesystem still match the hash before committing any changes from the sandbox. For conflicts in text files, standard Unix tools like `diff` and `patch` can often resolve the conflict, but in other cases like custom or binary files, users should manually merge them with application-specific tools.

4 Interposing system calls

In this section, we describe the recently introduced `seccomp/BPF` [1] as a means for interposing system calls; common pitfalls of using `ptrace` and `seccomp/BPF` for sandboxing; and how to use them to restrict network accesses and construct a fakeroot environment.

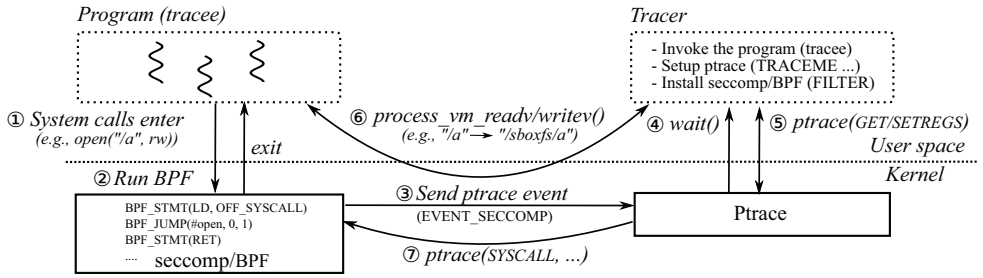

Figure 2: Interposing a system call with `seccomp/BPF` and `ptrace`. At startup, the tracer invokes the tracee, sets up `ptrace` and installs a `BPF` program. When the tracee generates a system call ①, the `BPF` program runs and decides whether to intercept or not ②. If the system call needs to be handled by MBOX, `BPF` will send a `seccomp` event ③ to the tracee waiting for a `ptrace` event ④. Then, the tracer queries the states of the tracee via the `ptrace` interface ⑤, or overwrites the tracee's memory with the `process_vm_writev()` system call ⑥. To continue the tracee, but stop at the exit of the current system call, the tracer needs to invoke `ptrace` with `SYSCALL`.

4.1 Using `seccomp/BPF`

`seccomp` [2] is a mechanism for isolating a process by allowing only a certain set of system calls. Linux 3.5 further introduced support for using Berkeley Packet Filter (`BPF`) bytecode to examine system calls when using `seccomp` [1]; for example, the `BPF` bytecode can decide whether the process can invoke the `socket()` system call. In `seccomp/BPF`, the input to the `BPF` program is the system call number, its arguments, and the instruction pointer, and it is invoked on every entry and exit of a system call. The `BPF` program decides whether to allow the system call to proceed or not; an additional option is to generate a `ptrace` event to the tracer, if the current process has one. Using `seccomp/BPF`, the tracer can download a `BPF` program and wait for a `ptrace` event, as described in Figure 2, instead of stopping on every tracee system call. This allows MBOX to interpose on just the necessary system calls, improving overall performance, as we show in §6.

4.2 Avoiding common pitfalls

It is easy to make mistakes when implementing a sandbox mechanism, making the resulting implementation vulnerable to adversaries due to minor mistakes. In particular, `ptrace` and `seccomp/BPF` are difficult to use correctly for interposing on system calls. We will now describe our experience in trying to avoid some of the pitfalls in using `ptrace` and `seccomp/BPF` for system call interposition.

4.2.1 Time-of-check-to-time-of-use (TOCTTOU)

Using `ptrace` to intercept system call entry allows us to examine, sanitize, and rewrite the system call's arguments. If an argument points to process memory, we can read remote memory and interpret it as the system call handler does. However, the read value can be different from what the system call handler will see in the kernel. For example, an adversary's thread can overwrite the memory that the current argument points to, right after the tracer checks the argument. Even verifying that sanitized arguments still point to the right value at system call exit does not help, because an adversary can restore it by that time.

To avoid TOCTTOU problems in rewriting memory arguments, MBOX takes advantage of two properties of `ptrace`. First, system call arguments examined using `PTRACE_GETREGS` are the actual values that the handler will see, because `x86-64` uses registers to pass system call arguments, and copies them to kernel space when entering the system call handler. Second, `ptrace` allows the tracer to write to read-only memory in the tracee with `PTRACE_POKEDATA`.

MBOX avoids TOCTTOU problems by mapping a page of read-only memory in the tracee process. When MBOX needs to examine, sanitize, or rewrite an in-memory data structure, such as a path name, used as a system call argument, MBOX copies the data structure to the read-only memory (using `PTRACE_POKEDATA` or the more efficient `process_vm_writev()`), and changes the system call argument pointer to point to this copy. For example, at the entry of an `open(path, O_WRONLY)` system call, the tracer first gets the system call's arguments, rewrites the path argument to point to the read-only memory, and updates the read-only memory with a new path pointing to the sandbox filesystem. Since no other threads can overwrite the read-only memory without invoking a system call (e.g., `mprotect()`), MBOX avoids TOCTTOU problem when rewriting path arguments. To ensure that the sandboxed process cannot change this read-only virtual memory mapping (e.g., using `mprotect()`, `mmap()`, or `mremap()`), MBOX intercepts these system call and kills the process if it detects an attempt to modify MBOX's special read-only page.

4.2.2 Replicating OS state

Another common mistake is to improve performance by replicating some state of the tracee process in the tracer. For example, in handling an `openat(fd, ...)` system call, one might think that keeping track of a path for `fd` whenever opening a path can improve performance, instead of reading the actual path for `fd`. However, it is

impractical to correctly emulate in userspace all subtle system calls that can change the state of a file descriptor. In MBOX, we design a set of stateless rules for deciding whether to rewrite a path argument of the current system call, by paying the cost of examining states at its entry. By controlling how a process can obtain a file descriptor in the first place, MBOX does not need to interpose on system calls that take only file descriptor arguments.

Rules for rewriting path arguments MBOX rewrites a path argument as follows:

- If `path` exists in the sandbox filesystem, then it was already modified by previous write operations. MBOX rewrites `path` to point at the sandbox filesystem, so that subsequent read/write should see the one in the sandbox filesystem.
- If `path` was deleted before, then to pretend that `path` in the host filesystem is deleted, `path` is rewritten to the non-existent path in the sandbox filesystem.
- If the current system call will modify the host filesystem, then, since `path` does not exist in the sandbox filesystem, MBOX copies the file from the host filesystem to the sandbox filesystem. The subsequent read/write will see the duplicated copy in the sandbox filesystem, by the first rule.

As one example, at the entry of an `open(path, O_RDWR)` system call, if `path` does not exists in the sandbox filesystem and was not deleted before, MBOX will copy the file from the host filesystem to the sandbox filesystem by the last rule, and rewrite the path to point to the sandbox filesystem, where any later `write()`s will be reflected. Any subsequent `open(path, O_RDONLY)` on the same path will also be rewritten to access the sandbox filesystem, by the first rule.

5 Implementation

We implemented a prototype of MBOX for Linux by extending `strace 4.7`, which is a system utility to trace system calls. To improve performance, we modified `strace` to use `seccomp/BPF`. For OSes that do not support `seccomp/BPF` yet, MBOX falls back to using `ptrace` as the main system call interposition mechanism (`seccomp/BPF` is supported on Linux 3.5 and above). MBOX has been tested on the x86-64 Arch distribution with the 3.8.10 Linux kernel, and the Ubuntu 12.04.1-LTS distribution with the 3.2.0-36 Linux kernel.

6 Evaluation

To analyze the performance characteristics of MBOX, we ran benchmarks used in Apiary [10] in three environments: without a sandbox, with MBOX using `ptrace`, and with MBOX using `seccomp/BPF` to intercept system calls. We carried out all experiments on a system with an Intel Core i7-2640M CPU, using one core with hyperthreads disabled, and 16GB RAM, running Arch Linux with kernel 3.8.10, if not stated specifically. Table 1 summarizes the results.

6.1 End-to-end performance overhead

In the computation-heavy Octave benchmark, Octave [6] in Table 1, MBOX exhibits negligible performance overheads, 0.1%, because it spends 98% of its execution time in userspace, with few system calls. However, when compressing files (Zip), decompressing files (Untar) or building the Linux kernel (Build Linux), MBOX incurs more significant overheads, 12.0%–20.9%, because these benchmarks invoke a lot of file-related system calls.

6.2 Interposing system calls

In the Zip and Untar benchmarks in Table 1, using `seccomp/BPF` was a lot more efficient than using `ptrace`. With `seccomp/BPF`, MBOX can intercept just the system calls that it needs to examine, and skip system calls such as `read()` and `write()` that take a file descriptor as an argument. Untar generates a total of 543k system calls, out of which 330k (60.8%) are `read()` and `write()`. Using `seccomp/BPF`, MBOX interposes on just 90k system calls (16.5%). These results show that `seccomp/BPF` helps MBOX reduce interposition overhead.

6.3 Concurrency

With `seccomp/BPF`, we can improve concurrency by avoiding unnecessary serialization of system calls, which enables each process to invoke system calls without being interleaved by the tracer. For example, `ptrace` imposed 110.1% overhead when building the Linux kernel in parallel, but using `seccomp/BPF` incurred 45.2% overhead, because the tracer interposed only on the necessary system calls, thereby allowing multiple system calls to execute simultaneously.

7 Related work

Layered filesystems UnionFS [8, 11] strongly influenced the design of MBOX; we follow its namespace unification rules and strategies for copy-on-write. However, MBOX enables them for non-root users by using `seccomp/BPF` in Linux, and also provides a variety of applications without requiring any modification of existing software. Cowdancer [12] and FL-COW [7] similarly provide a way to redirect modifications by a process, but since they use `LD_PRELOAD`, they cannot isolate a malicious process, unlike MBOX. Apiary [10] confines applications using UnionFS, but its main purpose of using the layered filesystem is to save storage by sharing package dependencies of confined applications.

Task	Normal	Sandbox				Description
		Ptrace		Seccomp/BPF		
Zip	15.6s	21.2s	36.5%	17.4s	12.0%	Compressing all files of linux-3.8
Octave	2.1s	2.1s	0.1%	2.1s	0.1%	Octave Benchmark [6] calculating matrix
Untar	13.6s	19.0s	40.3%	16.4s	20.9%	Decompressing linux-3.8 source files
Build Linux (-j1)	43.6s	53.2s	21.9%	49.7s	13.9%	Compiling linux-3.8 kernel
Build Linux (-j4)	21.7s	45.6s	110.1%	31.5s	45.2%	Compiling linux-3.8 kernel with 4 parallel jobs

Table 1: Performance benchmark results. Following the benchmark from Apiary [10], we measure the total execution time of each benchmark in normal execution, and in the sandbox using either `ptrace` and `seccomp/BPF`; for sandbox execution times, we also report the percent overhead on top of normal execution. We used two cores with hyperthreads enabled for the last benchmark, building the Linux kernel with 4 parallel jobs.

System call interposition Garfinkel used the system call interposition technique for enforcing security policies in Ostia [5], and studied common mistakes and pitfalls when using it for implementing a security tool [4]. In this paper, we summarized our experiences of avoiding those mistakes, especially the TOCTTOU attack, when using `seccomp/BPF` as a means for rewriting system calls.

Namespace The effectiveness of MBOX comes from the fact that every process can have a private namespace, detached from the host filesystem. Plan9 [9] originally proposed this idea; MBOX implements private namespaces by using `ptrace`, which commodity OSes provide to all users for debugging. MBOX, therefore, can use private namespaces for sandboxing without changing the kernel or applications. Docker [3] provides a container for applications by using namespaces, newly introduced in Linux 3.8, as a means to migrate processes transparently between OSes. We expect that the `mnt`, `net` and `ipc` namespaces, combined with Aufs [8], can be used for implementing an efficient layered filesystem, but without enabling all applications that MBOX provides with system call interposition.

8 Summary

We presented MBOX, a lightweight sandboxing mechanism for non-root users in commodity OSes. MBOX protects the host filesystem by layering the sandbox filesystem on top of it using efficient system call interposition based on `seccomp/BPF`. We showed that MBOX is effective in a variety of applications, and incurs reasonable CPU overhead. MBOX is available for download at `http://pdos.csail.mit.edu/mbox/`.

Acknowledgments

We thank Silas Boyd-Wickizer, Ramesh Chandra, Cody Cutler, Kavya Joshi, Meelap Shah, Keith Winstein, the anonymous reviewers, and our shepherd, David Presotto, for their feedback. This research was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

[1] Dynamic seccomp policies (using BPF filters). `http://lwn.net/Articles/475019`, January 2012.

[2] A. Arcangeli. Seccomp: secure computing mode. `http://en.wikipedia.org/wiki/Seccomp`. January 2013.

[3] dotCloud. Docker: The Linux container engine. `http://www.docker.io`, 2013.

[4] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2003.

[5] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2004.

[6] P. Grosjean. Octave benchmark 2: speed comparison of various number crunching packages (version 2). `http://sciviews.org/benchmark`. January 2013.

[7] D. Libenzi. FL-COW 0.10. `http://xmailserver.org/flcow.html`. January 2013.

[8] J. R. Okajima. Aufs3: Advanced multi layered unification filesystem version 3.x. `http://aufs.sf.net`. January 2013.

[9] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990.

[10] S. Potter and J. Nieh. Apiary: Easy-to-use desktop application fault containment on commodity operating systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 103–116, June 2010.

[11] D. Quigley, J. Sipek, C. Wright, and E. Zadok. Unionfs: User-and community-oriented development of a unification filesystem. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 349–362, 2006.

[12] J. Uekawa. Cowdancer: copy-on-write data access completely in userland. `http://www.netfort.gr.jp/~dancer/software/cowdancer.html.en`. January 2013.