Controlling Opaque-Component Effects with Semisolates and Ef

Evangelos Lamprou
Brown University

Tianyu (Ezri) Zhu
Stevens Institute of Technology

Di Jin
Brown University

Grigoris Ntousakis
Brown University

Georgios Liargkovas
Columbia University

Calvin Eng
Brown University

Konstantinos Kallas
University of California, Los Angeles

Michael Greenberg
Stevens Institute of Technology

Nikos Vasilakis
Brown University

Abstract

Many developers and systems today rely on opaque software components. When executing, these components affect each other and the broader environment in which they execute. Some of these effects are expected and desired; others not so. This paper introduces a novel abstraction and corresponding subsystem for controlling and manipulating the effects of opaque components. Available as a higher-order, language-agnostic command, `ef` interposes on a component's execution to automatically capture and control its effects. Effect control includes introspection, optional application, effect stacking, and further manipulation—all driven by several real-world case studies. Today `ef` is used in research and production applications across several organizations, mediating potentially undesired effects, maintaining full compatibility with real-world components, and incurring a modest performance overhead well within each case's acceptable levels.

1 Introduction

Modern software systems contain many intercommunicating components, including various subsystems, programs, commands, and packages [11, 41, 42, 80]. These components are often available in diverse programming languages, interact opaquely with the broader environment, and are developed by developers of varying skill and care. Even quick-and-dirty data-processing scripts pull together many polyglot components [41, 61] and simple update scripts or AI agents invoke other commands to complete their jobs [42, 80].

Unfortunately, the execution of these components—during *e.g.*, exploration or development—results in unpredictable and often-irreversible effects. These effects are a direct and necessary byproduct of the computation encoded by a component: executing `npm install` downloads and installs new software packages, directly altering several files and directories [1]. At times, effects are unintentional: naively running a `find -exec` as suggested by an LLM may result in deleting critical user files [45]. These and other cases (§2) hint at the necessity to control these effects during component execution.

Figure 1: Applying `ef`. Execute a command in its own `ef` semisolate to control its effects: inspect, selectively apply, revert, partially hide (all shown) or otherwise manipulate them—all within the *current* environment rather than a new isolated container.

This paper presents a new abstraction, the *semisolate* (§3), and corresponding implementation, `ef` (§4), for controlling the effects of such opaque components. Semisolates offer the ability to execute a component in the current environment via a language-agnostic, higher-order command that allows its users to inspect and manipulate a completely unmodified component's externally observable behavior. But rather than focusing on the behavior encoded in standard streams, easily controllable by the usual pipe-and-filter patterns, `ef` focuses on *side* effects such as filesystem modifications. And contrary to various forms of containment and virtualization [10, 47, 58], which create a new environment that fully isolates all effects, `ef` allows the executing components to access the current environment in which they execute and enables effect manipulation directly in that environment.

More specifically, `ef` allows introspecting, selectively applying, stacking, and further manipulating a component's effects to the filesystem and the broader environment. To automatically capture these effects, `ef` limits the component's view of the filesystem: it first sets up a *semisolate* to create a view of the filesystem and environment that retains partial isolation; it then executes the component inside the semisolate, collecting effects into a writable temporary directory; finally, either during its (streaming) execution or upon completion, the semisolate allows manipulating the collected effects by staging them, reverting them or applying them to the underlying filesystem—either wholly or partially (Fig. 1).

This opaque effect control is applicable broadly (§2): it can be used to inspect the effects of risky or cryptic LLM suggestions before applying them [3]; track the dependency graph

for systems composed of opaque components, useful for parallelization or reordering [56]; mediate undesired effects by third-party packages, including post-installation scripts [1]; and support the specification inference of opaque components [42]. These and other examples drive `ef`'s design requirements: effect control that allows selectively hiding state, stacking effects, and partially or fully applying or rolling them back. No current abstraction and corresponding subsystem offer granular effect control with `ef`'s wealth of features.

Applying `ef` to real-world instances (§5) indicates that `ef` controls all component effects, maintains behavior equivalence with non-semisolated execution, and incurs a modest performance overhead relative to the need of each use case. Semisolates and `ef` offer a distinct set of features from, and better performance than, containers and Docker [47], built using similar techniques arranged quite differently.

In summary, this paper makes several contributions: it

- identifies the shared needs of several case studies for which today's effect control is insufficient (§2);
- presents a novel effect-control abstraction, a semisolate, that meets the needs of these case studies (§3);
- describes the implementation of `ef`, a new effect-control subsystem built for Linux environments (§4);
- characterizes `ef` by applying it to several real-world components drawn from the earlier case studies (§5).

It also discusses `ef`'s limitations and the experiences of organizations using `ef` (§6), as well as prior related work (§7). Appendix A collects all LLM prompts used in the evaluation and B several testimonials on `ef`'s potential impact, included only for double-blind reviewing. The `ef` subsystem is open-source, MIT-licensed software, publicly available on GitHub [23].[1]

2 Examples and Goals

To understand the need for controlling component effects, this section describes several real-world case studies (§2.1). Combined, these case studies highlight several shared requirements for the design of a flexible effect-control abstraction and corresponding subsystem (§2.2).

2.1 Motivating Examples

Risky or cryptic LLM suggestions: Recent agentic systems have incorporated large language models (LLMs), allowing them to act on behalf of the developer by executing arbitrary commands inside the host system [2, 3, 7], with sometimes incorrect or catastrophic results [27,45,83]. Consider an LLM-generated snippet intended to "delete the directory containing a script, invoked from any directory" to later update it:

```
rm -fr "$(cd "${0%/*}" && echo $PWD)"/*
```

Unfortunately, called from some paths, the runtime expansion `${0%/*}` will result in the script name, `cd` will fail, and `rm` will delete everything user-writable. Prefixing this line with `ef` indicates the deletion of several critical files and offers the option to discard these changes.

```
ef rm -fr "$(cd "${0%/*}" && echo $PWD)"/*
  /home/user/photos/family_01.png (modified)
  /home/user/photos/match_01.png (modified)
  ...
  Apply these changes? [y/i/N]
```

A developer or system calling `ef` can decide whether to inspect these changes and choose to discard them, or inspect them individually and choose which ones to apply.

Dependency tracking: Speculative execution reordering offers significant speedups [56], but ensuring it respects read-after-write dependencies across opaque components is particularly challenging. Consider the following program that compiles three and links two `C` files:

```
gcc -c a.c -o a.o; gcc b.c -o a;
gcc -c c.c -o c.o; gcc a.o c.o -o b;
```

The second call to `gcc` does not depend on the first, thus it can be executed in parallel. Unfortunately, detecting this opportunity automatically at runtime requires detecting effects and postponing their application to the broader environment until they are confirmed to not violate read-after-write dependencies—at times discarding these effects to effectively roll back command misspeculation. Crucially, subsequent invocations must start their execution with the illusion that prior effects have been applied—necessitating chaining reversible effect application across component invocations.

Prefixing `gcc` with `ef` while explicitly asking it to stack effect environments solves both problems:

```
ef -x -N env1 gcc -c c.c -o c.o
ef -x -N env2 -L env1 gcc a.o c.o -o b
```

The `-N` option instructs `ef` to store, instead of directly applying, `gcc`'s effects for later use with a configurable name. The `-x` flag enables system-call interposition to capture all read-write effects. Next, `-L` injects `gcc`'s uncommitted effects from `env1` into its own view, making `c.o` visible to `ef`'s second invocation.

Third-party library risks: Public third-party libraries amplify developer capabilities, but risk introducing undesired effects [15,49,54]. Consider the popular `node-ipc` library [62], invoked as part of a larger codebase:

```
npm test node-ipc
```

Earlier versions of `node-ipc` included undesired effects modifying critical system files [17, 74]. Prefixing `npm test` with `ef` highlights files about to be modified without making any modifications prior to approval:

[1] System renamed to `ef` for double-blind reviewing; the project is worked on by several institutions, and is used by multiple downstream projects—with key experiences summarized in §6.

```
ef npm test node-ipc
  /etc/passwd (modified)
  /bin/bash (modified)
  /sbin/init (modified)
  ...
  Commit these changes? [y/i/N]
```

The output of `ef` highlights several such effects, allowing a developer to review unexpected changes and, if necessary, selectively apply only the relevant changes.

Cautious software installation: Package installations are common tasks performed daily, but are often risky due to accidental effects specific to the target environment. Consider the installation of `eslint-scope` [77]:

```
npm install eslint-scope
```

Unfortunately, the post-installation script for `eslint-scope` v.3.7.2 reads `_authToken` tokens from `.npmrc` files and communicates them over the network [1]. Prefixing `npm install` with `ef -I` instructs `ef` to include only a specific directory subtree, excluding everything else from the view of the command it prefixes:

```
ef -I /usr/local/lib/node_modules npm install
  Access to /home/user/.npmrc denied
```

Using `ef`, no process or underlying subprocesses under `ef` can access or modify files in `/.npmrc`. Not shown: Inversely, `ef`'s `-E` flag offers explicit exclusion, but this case would require enumerating a large number of directories, and `ef -x` limits network access altogether.

Partial-specification mining: Automated acceleration of programs that use opaque components [35, 61] depend on partial component specifications—*e.g.*, the order in which a component reads its input streams [31]. Generating such specifications automatically requires exploring a command's invocation space while carefully collecting the effects of different invocations. Consider the following invocation of `grep`:

```
grep -oE [a-zA-Z]+ one.txt two.txt
```

Unfortunately, understanding that `grep` first reads `one.txt` and *then* reads `two.txt` requires a more detailed view of its effects. Prefixing `grep` with `ef -t` offers a detailed, ordered log of `grep`'s read and write effects:

```
ef -t trace.log grep -oE [a-zA-Z]+ names.txt
```

The resulting `trace.log` file contains information that can then be compiled into a partial specification for `grep`.

```
r /usr/bin/grep
r /usr/lib/libc.so.6
r one.txt
r two.txt
w stdout
```

This order of effects extracted by `ef -t` reveals invariants (*e.g.*, `one.txt` preceding `two.txt`) that need to be maintained when optimizing programs that invoke `grep`.

2.2 Desiderata for the Missing Abstraction

The aforementioned case studies illustrate an underlying common need to abstract away effect control. A suitable effect control abstraction has several core requirements: (1, **I**) *effect introspection*, *i.e.*, the ability to detect, collect, present, and inspect a component's side effects in the same environment directly executing the component; (2, **A**) *selective application*, *i.e.*, the ability to decide whether to apply effects to the broader environment or roll them back; (3, **S**) *effect stacking*, allowing commands to execute in the context of earlier effects that have not been applied yet, potentially rolling back multiple stacks of effects; (4, **M**) *further effect manipulation* beyond direct application or discard, including partial application of only some effects, effect recording and delayed propagation, and selective hiding of the underlying environment. These requirements typically need to be met in the context of the current executing environment—*not* a virtualized or contained environment fully isolated from the current environment.

Apart from these core requirements, the abstraction's practical applicability and reach would further benefit from an implementation that features streaming support, allowing its users to manipulate the effects of long-running components as they are generated; configurable effect handlers for tuning the granularity of effects; and runtime performance overheads that do not affect interactive use. These features are not essential to the core abstraction—*i.e.*, a batch-only, non-configurable, slow implementation would still fit all of §2.1's case studies—but they broaden its applicability and reach.

Other characteristics such as complete mediation against arbitrary adversarial behavior are out of scope: the abstraction aims at the inconvenient or undesired effects of everyday mistakes or accidents—not actively malicious, `ef`-aware components actively trying to bypass it.

3 Abstracting Away Effect Control

To meet the aforementioned desiderata (§2.2) and abstract away effect control, `ef` offers a language-agnostic, higher-order, and configurable abstraction that launches one or more first-order components in a semisolate. A semisolate offers a constrained and configurable private view of the real host environment—*i.e.*, not a fresh environment in a container or virtual machine. In the example below, `sh` runs in the exact same environment as `curl`, but within a private view that controls, manipulates, and potentially discards the application of effects to the broader environment:

```
curl -sL https://sh.rustup.rs | ef sh
```

This private view can be configured to, for example, completely mirror the global filesystem, partially mirror it while hiding some of the paths, or correspond to one or more layers of as-yet-unapplied effects from previous `ef` executions.

Table 1: Control configuration and `ef` flags. Controlling effects with `ef` supports significant configurability during semisolate creation, execution, and conclusion—including core options (in black, *Cf.*§3) and peripheral features and subsystems (faded, *Cf.*§6).

Semisolate stage	Configuration	Flag
Creation	Stack over prior effects	`-L`
	Name current effect group	`-N`
	Ignore specific paths	`-i`
Execution	Disable the network	`-x`
	Create filesystem trace	`-t`
	Collect effect delta	`--diff`
Conclusion	Apply all effects	`-y`
	Discard all effects	`-n`
	Inspect effects	`-e`
	Include, exclude specific paths	`-I`, `-E`
	Summarize effects	`-s`
	Offer friendlier summary	`-h`

Semisolates: The abstraction of a semisolate provides the illusion of direct access to its outer environment and supports flexible effect manipulation. Semisolates collect the effects of opaque components in this private view and expose them to their callers for further manipulation. Using semisolates, `ef` allows its callers to (1) selectively apply or discard these effects, with options to view partial earlier state; (2) collect the effects for possible application later; and (3) stack and expose as-yet-unapplied effects across invocations. This design allows the effect-mediated component to execute in partial isolation, without affecting the global filesystem until the caller explicitly consents.

To achieve effect control, a semisolate operates in three configurable stages: (1) *creation*: the semisolate sets up the appropriate view for the executing component to ensure appropriate effect mediation; (2) *execution*: the semisolate launches the target component within a private view, collecting relevant effects; and (3) *conclusion*: the semisolate optionally and selectively applies the recorded effects back into the main environment. Components execute in semisolates until explicitly tasked to apply their effects, while supporting both sensitive path hiding and execution chaining for complex workflows.

All three stages typically operate on the current directory and are highly configurable to support the nuances of various real-world uses (§2). The next few paragraphs detail such tuning, both at the level of individual stages and the current `ef` flags summarized in Tab. 1, after first establishing which effects are controllable and their level of control.

Which effects are controllable? As Unix descendants already offer well-studied pipe-and-filter abstractions for manipulating standard streams such as `stdin`, `stdout`, and `stderr`, semisolates by default go beyond these streams. They add abstractions for manipulating effects around a child command—for example, the initial filesystem view, the final filesystem view, and the filesystem access trace. They can optionally be configured to support additional system and network effects and standard-stream inputs and outputs as part of a unified abstraction.

Effects are made available to the semisolate caller as a combination of (1) filesystem paths or stream identifiers, and (2) optional metadata such as effect type and success. Each path can be a valid path pointing to an existing file or directory; it can also be a *non*-existent path that might lead to access failure—still useful when a command relies on a file's absence to exhibit the intended behavior. The effect type includes information such as read-only, write-only, and read-write: a read-write effect is, for example, `open`ing a file with any flag other than `O_RDONLY`; a read-only effect includes operations that only interrogate a file's metadata through, for example, `stat`ing a file; a write effect includes modifying filesystem metadata through, for example, `chmod`ing or `unlink`ing a file. Each recorded effect instance is accompanied by metadata that indicates whether an effect succeeded; or—if it failed—the reason for its failure.

By default, `ef` manipulates effects across entire process subtrees. For example, if an `ef`-controlled process launches a subprocess that writes to the filesystem, then `ef` will include those write effects in the group of effects directly available for manipulation. A parent process can also use `ef` to wrap and programmatically control child effects.

Semisolate creation: Upon creation, a semisolate configures the private view of the executing component—to allow both stacking the effects of different semisolates and avoiding effects on data that should not be affected (including reads), useful for hiding sensitive information from the executing component. Configuring this initial view translates to setting up specific virtual-filesystem environments before any modifications to the filesystem.

Upon creation, semisolates may configure effect stacking and hiding. Stacking configures the component's initial view to import the effects produced from a previous semisolate. Selective hiding masks certain parts of the underlying filesystem from the executing component, given a specification that describes a filesystem subtree. These are implemented by `ef`'s interface as `-L`, which accepts the name of a previous semisolate, itself produced by a previous call to `ef` with the `-N` flag; and `-i`, which takes as input a regular expression that matches filesystem paths to be hidden from the component's view. The latter is semantically equivalent to stacking the effects of `ef -n N rm dir1 dir2...` over `ef -L N`—but `-i` is vastly more efficient as it directly sets up exclusions without first running an additional (and potentially expensive) command.

Semisolate execution: During execution, semisolates additionally configure the classes of effects to interpose on and the corresponding interposition granularity—for example, whether to include system-call tracing. The current classes of effects include filesystem effects, network effects, signals, and interprocess streams.

The current interface of `ef` supports: `-t`, which makes `ef` produce a filesystem access trace, containing an ordered list of effects; and `-x`, which prevents the component from accessing the network. Apart from the effect path, this detailed view consists of the mode and the return code, which can be inspected to infer negative effects like failed accesses.

Semisolate conclusion: Upon conclusion, semisolates offer the ability to decide which effects to apply and which to discard, eventually generating the final filesystem view. They allow selective effect application at multiple granularities, from binary all-or-nothing to per-effect application, while also supporting path-oriented filtering. The can also hold individual effects and allow inspecting them directly using conventional Unix abstractions.

These options can be configured when a semisolate concludes. The current interface supports inspecting individual effects (`-e`) and, based on pre-defined caller policies, programmatically filtering these changes (`-E` and `-I`). It also supports applying, discarding, or mixing them (`-y`, `-n`, and `-e`). And `-N` stores effects in a named effect directory, necessary earlier during creation with `-L`.

4 The Ef Subsystem

This section describes how `ef` implements the aforementioned abstraction for programs executing in Linux userspace. It first outlines `ef`'s use of union filesystems, process namespaces, and system-call tracing before diving into the details of semisolate creation (§4.1), execution (§4.2), and conclusion (§4.3), closing with a detailed walkthrough of multiple `ef` invocations put together (§4.4).

Union filesystems: To achieve effect control, `ef` needs to interpose on the component's view of the filesystem. This interposition is responsible for manipulating the underlying filesystem view, creating a partially private view that allows accessing the underlying contents while capturing and containing effects. This private view is implemented as a union filesystem, configured appropriately to either mirror the underlying filesystem, partially mirror it with sensitive paths hidden, or correspond to the as-yet-uncommitted state of a previously executed run of `ef`. To contain effects, `ef` uses the configured union filesystem to record filesystem changes into a temporary directory—without affecting the *host environment* until effects are applied.

To instantiate the concept of effect *layers*, `ef` leverages a union filesystem. Union filesystems are a class of filesystems that allow multiple directories to be presented as a single unified directory without modifying the underlying kernel structures [60]. A union filesystem layers an upper directory (*upperdir*) and one or more lower directories (*lowerdirs*) into a single merged view—without merging their underlying data. OverlayFS, which `ef` uses, is a specific implementation of a union filesystem which is part of the Linux kernel [22].

In the merged view, the content at a specific path `p` is determined by searching for `p` in the upperdir, followed by all lowerdirs in order until `p` is found. The first match is presented in the merged view; if `p` exists in multiple layers, content from the higher layer takes precedence.

Changes to the merged view are forwarded to the upperdir and persist inside it, even after unmounting the union filesystem directory. Changing a file at `p` in the merged view places the new version of the file at `p` in the upperdir. Removing `p` places a *whiteout file* at the corresponding upperdir location.

Process namespaces: To execute privileged actions—while executing in an un-privileged manner—and interpose on the execution context of the component, `ef` leverages Linux process namespaces. Namespaces isolate resources such as the network, pseudo-filesystems like `procfs`, permissions, and signals from the rest of the system, offering (groups of) processes the illusion of exclusive ownership of those resources—enabling the creation of a private (and controllable) view of the underlying system state. For example, the PID namespace assigns new PIDs to processes inside the namespace: processes in the namespace can only see and interact with each other using the PIDs assigned in the namespace.

Process-related interaction such as signaling between processes and adjustment of scheduling priorities are contained to the namespace. Components in `ef` are only partly able to interact with ambient resources and objects such as processes, users, or network devices—with `ef` interposing on all accesses to the broader system.

System-call tracing: To observe the order of a component's effects, and go beyond effects that modify the filesystem, `ef` leverages system-call tracing—interposing between the operating system kernel and user-space components to monitor and record their interactions. Tracing commands such as `strace` on Linux offer visibility into exact calls performed by a process, their order, and their arguments, as well as the signals received by the process. Combined, they offer a wealth of information about the process—including filesystem operations, process management, inter-process communication, and network access. By selectively enabling call tracing when necessary, `ef` extracts ordered, finer-grained information about component effects beyond filesystem modifications.

4.1 Semisolate creation

Each semisolate uses its own special writable directory to record effects and a second temporary directory—the 'workdir'—used by the union filesystem for internal bookkeeping (Alg. 1, lines 1–5). Each of these directories can be (1) a new, appropriately prefixed temporary directory created with `mktemp`; (2) a new directory whose name is provided by the `ef` caller; or (3) an existing directory, potentially used in previous `ef` invocations.

OverlayFS does not support certain filesystems to host the upperdir or the 'workdir'. One such filesystem is OverlayFS

Algorithm 1 Semisolate creation. Key steps in setting up a new semisolate, before running a component;

Require: *path*
1: **if** *path* is not provided **then**
2: *path* ← `mktemp -d`
3: **end if** ▷ Set up directories
4: **if** *path*'s filesystem is overlayfs **then**
5: `mount -t tmpfs tmpfs` *path*
6: **end if**
7: **for all** *dir* **in** *topLevelDirs* **do**
8: `mkdir -p` *path*`/{upperdir,workdir,tmp}/`*dir*
9: **end for**
10: `unshare ...` ▷ Prepare namespaces
11: **for all** *dir* **in** *topLevelDirs* **do**
12: `mount -t overlayfs ...`*path*`/tmp/`*dir*
13: **if** overlay failed **then**
14: `mergerfs` *path*`/tmp/`*dir*`...`
15: `mount -t overlayfs...`*path*`/tmp/`*dir*
16: **end if**
17: **end for**
18: `mount` *path*`/tmp/dev/...` ▷ Mount devices
19: `unshare ...` ▷ Enter environment

itself, a common scenario when using `ef` within Docker. To address this limitation, `ef` mounts an additional `tmpfs` over the semisolate directory (Alg. 1, line 5). This ensures that all of `ef`'s side effects prior to the conclusion stage are confined within the semisolate directory.

To allow hosting OverlayFS' `upperdir` and `workdir` across all filesystems and allow sharing effects with the broader environment outside the semisolate, `ef` mounts an additional `tmpfs` over the semisolate directory as necessary (Alg. 1, line 5).

Setting up mounting options: To bypass OverlayFS's restriction of overlapping the lower and upper directory, `ef` creates an individual union filesystem mount for each top-level directory—*e.g.*, `/bin`, `/usr`, and `/home`—instead of using the root directory `/` as the union filesystem's lower directory.

Specifically, `ef` first creates directories `upperdir`, `workdir`, and `tmproot` (Alg. 1, lines 5–12). In these top-level directories, it then creates a child directory for each top-level directory in the filesystem, such as `/etc`, `/usr`, and `/home`. Additional lower directories already containing effects are added to the mount options during this step—layered on top of the corresponding union filesystem merged views. The `tmproot` directory will become the final merged view presented to the executing component (§4.2).

Once inside the new namespace, `ef` iterates over the mount options constructed in the previous step and mounts the union filesystem (Alg. 1, lines 12–14). It mounts the `/proc`, `/dev`, and `/devpts` pseudo-filesystems within the `tmproot` directory and symlinks all special files through which the component interacts with its standard input and output streams—*i.e.*, `/proc/self/fd/{0,1,2}` and `/dev/{stdin,stdout,stderr}`. Using bind-mounts [21], it duplicates device files such as `zero`, `null`, and `random`, into the `tmproots`'s `/dev` directory.

To implement file hiding, `ef` creates an additional directory called `excldir`. This `excldir` directory is mounted as an intermediate layer right above the lower directory and contains whiteout files corresponding to files that will be hidden from the merged view.

Submount permissions: Unfortunately, filesystem submounts within any OverlayFS layer cannot be unmounted by the current user: mount points inside layers are ignored because these layers correspond to overlapping filesystem fragments. Consider an initial filesystem *A* that contains paths `/x`, `/x/y`, and `/x/y/z`, and a new filesystem *B* mounted at `/x/y`, effectively hiding `/x/y/z` as contents at that mount point by *B*. If `/x` is specified as an OverlayFS lower directory, it will include only the contents of *A* and ignore the submount to *B*—thus including `/x/y/z` in the merged view.

As `ef` aims to limit access to information otherwise inaccessible without its use, it prevents unprivileged access to directories with submounts as lower or upper layers. Although the fresh mount namespace grants `ef` the capability to create new overlay mounts, among other root-like capabilities, it is forbidden to unmount existing submounts.

To address this problem, `ef` uses an additional union filesystem to "flatten" all content within the directory into a single filesystem (Alg. 1, line 14) when submounts are present. The union filesystem used by `ef` for this purpose is mergerfs [53], which combines the entire directory, including its submounts, into a single, unified filesystem view. This contrasts with OverlayFS, which does not support submounts. By creating this flattened view of the directory, submounts are effectively eliminated, allowing OverlayFS to use the directory as a 'lowerdir' without issues.

4.2 Semisolate execution

After preparing a semisolate, `ef` launches the effect-controlled component in its own dedicated namespace and optionally traces its execution.

Execution namespacing: The semisolate executes components within a new user, PID, mount, and network namespace (Alg. 1, line 10). There are two distinct reasons for which `ef` uses namespaces: (1) namespaces offer controlled isolation from the broader environment, restricting the process's view so that all effects—together with the filesystem view shaped by the semisolate (§4.1)—are contained, (2) namespaces create the illusion of administrative privileges, allowing a semisolate to mount `/proc` and the union filesystem itself, which would otherwise require elevated privileges. The capability for user namespaces requires setting the `unprivileged_userns_clone` kernel flag, which is enabled by default in most modern Linux distributions.

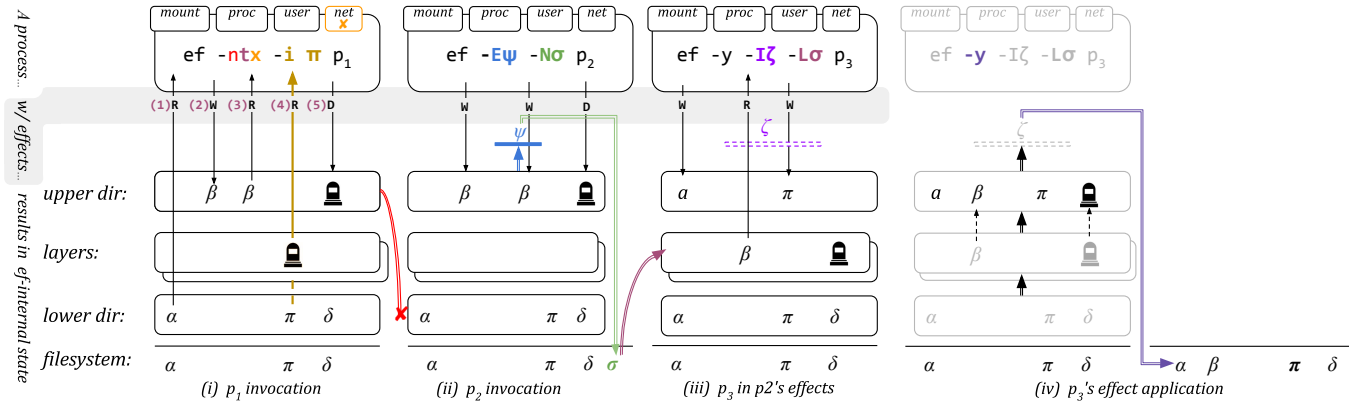

Figure 2: Detailed walkthrough of the internals of multiple `ef` invocations A detailed walkthrough of these invocations is presented in §4.4. From top to bottom: `ef`'s invocation, component effects, `ef` internal state, and host filesystem state. From left to right: different `ef` invocations on different component on the same host filesystem. Single arrows indicate effects; double arrows indicate effect discard, storage, stacking, and application; and certain colors correspond to certain `ef` invocation options.

These special files do not maintain persistent state after `ef` exits, containing any side effects.

Finally, `ef` enters a second new namespace (Alg. 1, line 19) to isolate the component's process tree from `ef`'s own process, setting the component's root directory to `tmproot` using `chroot`, and finally executes the component under optional system-call tracing.

Execution tracing: To extract additional information about filesystem effects, `ef` can interpose on relevant system-call invocations using `strace`. Rather than tracing all filesystem-related system calls when they occur, `ef` focuses collecting relevant information upon `open`—as this is typically a necessary step before subsequent operations on the file descriptor such as reads, writes, random accesses, or direct memory mapping. This approach lowers `ef`'s performance overhead, as it avoids tracing reads, writes, and other latency-critical data manipulation operations, omitting further tracing during subsequent use of file descriptor.

To improve its reach while still avoiding finer-grained tracing, `ef` additionally traces other system calls that take paths as arguments—*e.g.*, `open`, `mkdir`, and `unlink`. To trap only this preconfigured set of system calls, `ef` uses Seccomp-BPF [39] with a predefined filter—but allows users to pass additional filters if they need to track additional effects.

Non-existent paths: Non-existent or missing paths pose a challenge, as ignoring their accesses risks introducing effect holes: accessing non-existent paths generates an observable effects—*e.g.*, negative return value from `open`. Therefore, `ef` records such accesses as negative dependencies, carefully marking their presence using special metadata. Such negative dependencies are additionally useful for recording effects of searching dynamic library paths and configuration files.

4.3 Semisolate conclusion

Semisolates conclude their execution by going through all the effects and either applying pre-defined policies or logging effects to one of the standard streams. As all changes are captured in the effect directory, applying these effects amounts to carefully comparing the captured effects to the underlying filesystem and environment.

To apply filesystem-related effects, `ef` recursively processes the upper directory in a preorder fashion, moving files out of the semisolate and into the real filesystem by first handling directories before their contents. For regular files in the upper directory, `ef` replaces the original file with the new file. For directories, it either (1) replaces the original file with the new directory, if a regular file exists at the directory's original path; (2) replaces the original file or directory with the new file, if the directory has the extended attribute `user.overlay.opaque`, indicating that directories in lower layers have been shadowed by new definitions in the merged view; or (3) replaces the original directory with the new directory. If `ef` encounters a whiteout file—a character device with 0 major and minor numbers—in the upperdir, it deletes the original file or directory at that path.

If the semisolate directory resides on the same filesystem as the target location, `rename` is used to avoid unnecessary disk copying. Otherwise, `ef` falls back to the host's `mv` utility, which will typically fall back to efficient choices for the filesystems in question (*e.g.*, `copy_file_range`).

4.4 Putting it all together

Fig. 2 illustrates the internals of four `ef` invocations.

The first invocation wraps p_1, eliding all network access inside the semisolate. It prepares an overlay layer with a white-out mask at path π, and mounts it on top of the host

filesystem's lowerdir, ignoring the underlying π in the filesystem. It then enters the new namespace, and launches p_1 inside it. While p_1 is running, `ef` captures four effects in the upperdir, which it later processes to summarize the order of effects (`-t`): (1) reading α hits the lowerdir, which forwards the call to the underlying filesystem, (2) writing to β hits the upperdir, copying β there, (3) reading β hits the upperdir, which sees the latest write, (4) reading path π, masked in the overlay layer, thus failing and resulting in `ef` recording a negative dependency; and deleting path δ, which places a mask file at δ in the upperdir, effectively hiding it from the merged view. The sequence of collected effects (α_R, β_W, β_R, π_R, δ_D) is fed to standard out but the effects themselves are discarded alongside the semisolate (`-n`).

Next, `ef` wraps p_2, creating a new namespace and mounting the host filesystem inside a fresh overlay filesystem lowerdir, now without any path masking or network isolation. The three component effects it captures are: (1) writing β, with identical effects as before, (2) writing ψ, and (3) deleting δ—both operations captured in the upperdir. This time, `ef` stores the semisolate (and by extension, its effects) inside a named directory on the host filesystem (`-N` σ), after filtering out excluded effects (`-E` ψ) by iterating over the upperdir and deleting them.

Next `ef` wraps p_3 while stacking the effects of the latest `ef` invocation (`-L` σ). It captures: (1) writing α, which brings it to the upperdir, (2) reading β, which hits the stacked lowerdir from semisolate σ, and (3) writing π—recorded in the upperdir. The effects stored in the semisolate are the two writes to α and π, and the delete of δ inherited from σ.

Finally, `ef` commits the effects (`-y`). It iterates from bottom to top, first applying the effects from the stacked semisolate σ, followed by the current semisolate's effects, applying each one based on its type. It focuses on paths in ζ (`-I` ζ), *i.e.*, the writes to β and π, which are applied to the host system.

This approach implements the semisolate abstraction, allowing for effect control on arbitrary components without requiring any invasive instrumentation or modification.

5 Evaluation

This section reports on the results of applying `ef` in several instances of each case-study application presented earlier (§2). Key dimensions of each study's characterization (§5.1–5.5) include (1) `ef`'s ability to *control* all relevant effects; (2) `ef`'s behavioral *equivalence* to semisolate-free execution when effects are finally applied to the host environment; and (3) `ef`'s runtime *performance* relative to semisolate-free execution and full isolation (§2.2). A series of synthetic microbenchmarks zoom into `ef`'s sources of overhead (§5.6).

Control focuses on confirming that `ef` correctly identifies **I**, applies or discards **A**, selects **S**, or manipulates **M** all relevant effects during a component's execution in a semisolate.

Behavioral equivalence focuses on confirming two aspects of `ef`: (1) the in-semisolate behavior during execution, meaning that the computation in the semisolate executes as expected and produces the results identical to the non-semisolated execution; and (2) the eventual state of the filesystem, meaning that the filesystem results in state that is indistinguishable from `ef`-free execution without any undesirable effects. The former confirms that `ef` does not affect the internal computation, including its direct effects; and the latter confirms the application of manipulated effects operates as expected.

The performance characterization compares between completion timings with no isolation, with `ef`, and with Docker v28.4.0; performing container (not image) creation, input copy-in, execution, container teardown and copy-out. Experiments use an environment with a 64GB RAM, 8-core 3.70 GHz Intel Xeon W-2145. Software used is the Linux kernel v6.1 on Debian GNU/Linux 13 (trixie), GNU coreutils v9.7, Node.js v18.19.0, and Python v3.13.5.

Benchmark results are summarized in Table 2. In terms of effect control, `ef` mediates desired effects, propagates them appropriately, and hides sensitive environment state. In terms of behavioral equivalence, `ef` behaves equivalent to `ef`-free execution, confirmed both via manual inspection and automated checksum comparison between the entire filesystem tree after executing each benchmark with and without `ef`. In terms of performance, `ef` suffers a 1.0×–8.3× performance overhead over vanilla, `ef`-free execution and enjoys a 1.3×–225.7× performance improvement over containment in Docker. We do not provide a geomean or other statistical summary for overhead over vanilla execution or speedup over Docker, as each case study requires different semisolate configurations and corresponding `ef` flags to control effects—and thus incur different overheads.

We discuss the benchmarks grouped by use case below.

5.1 Risky or cryptic LLM suggestions

These benchmarks consist of five scripts generated by ChatGPT for the following tasks (prompts in Appendix §A): (1) **`crawl`**, updates the timestamps of 10,000 files in a directory, using `find`, `exec`, and `touch` [78]; (2) **`fresh`**, compressing 10,000 10-KB files that exceed a certain size, using `find`, `exec`, and `gzip` [32]; (3) **`archive`**, searching for files with a `.txt` extension in 100 directories of 10,000 files each, using `find` [34]; (4) **`logs`**, checking for errors in a 10-million-line log file using `grep` [37]; (5) **`order`**, sorting a 10-million-line file using `sort` [6]. Benchmarks were executed with both `ef -y` and `ef -n`.

Control: All computations executing in semisolates configured with `ef -n` mediated all effects that were applied to the filesystem by `ef`-free executions.

Equivalence: All computations executing in semisolates configured with `ef -y` produce correct results; and their effects

Table 2: Benchmark summary. The benchmark applications for the five use cases evaluated. The **Control** column indicates `ef`'s effect control between effect introspection (**I**), application and discard (**A**), stacking (**S**), and manipulation (**M**). The **Equivalence** columns indicate whether the output (O) and side effects (SE) match those of vanilla execution. The **Performance** columns show the **execution time ratio** relative to the baseline. Specifically, the **Performance** columns show the execution time ratio relative to the baseline (Docker or Vanilla), and whether `ef` comparatively has a speedup (↑) or slowdown (↓). The **Description** column describes each benchmark's purpose and `ef`'s invocation. The **Source** column provides references to the original benchmark sources.

Use case	Name	Control		Equivalence O	Equivalence SE	Performance vs. Docker	Performance vs. Vanilla	Description	Source
Risky or cryptic LLM suggestions (§5.1)	**crawl**	✓	**I A M**	✓	✓	1.4×↑	1.4×↓	LLM-generated shell pipelines (`ef`)	[6, 32, 34, 37, 78]
	fresh	✓	**I A M**	✓	✓	1.3×↑	1.8×↓		
	archive	✓	**I A M**	✓	✓	225.7×↑	1.3×↓		
	logs	✓	**I A M**	✓	✓	58.2×↑	1.9×↓		
	order	✓	**I A M**	✓	✓	2.0×↑	1.1×↓		
Dependency tracking (§5.2)	**covid**	✓	**I A S M**	✓	✓	1.5×↑	1.0×↓	Script with dependency tracking (`ef -t -L`)	[13, 38, 81, 82]
	nlp	✓	**I A S M**	✓	✓	3.4×↑	1.2×↓		
	spell	✓	**I A S M**	✓	✓	2.3×↑	1.0×↓		
	unixfun	✓	**I A S M**	✓	✓	18.7×↑	1.7×↓		
Third-party library risks (§5.3)	**LinOTP**	✓	**I A M**	✓	✓	2.0×↑	1.2×↓	Git repos with risky pre-commit hooks (`ef -i`)	[8, 28, 55, 57, 63]
	frogmouth	✓	**I A M**	✓	✓	4.2×↑	1.0×↓		
	kibble	✓	**I A M**	✓	✓	1.4×↑	1.2×↓		
	okteto	✓	**I A M**	✓	✓	1.7×↑	1.3×↓		
	uv-metrics	✓	**I A M**	✓	✓	3.1×↑	1.0×↓		
Cautious software installation (§5.4)	**eslint-scope**	✓	**I A M**	✓	✓	4.8×↑	1.2×↓	NPM packages with post-install scripts (`ef -I`)	[25, 26, 67, 71]
	coa	✓	**I A M**	✓	✓	5.8×↑	1.2×↓		
	node-sass	✓	**I A M**	✓	✓	5.7×↑	1.3×↓		
	ua-parser-js	✓	**I A M**	✓	✓	5.2×↑	1.2×↓		
Partial-specification mining (§5.5)	**cp**	✓	**I A S**	✓	✓	2.2×↑	8.3×↓	Standard unix utilities (`ef -t`)	[35, 61]
	ls	✓	**I A S**	✓	✓	1.9×↑	3.7×↓		
	rm	✓	**I A S**	✓	✓	1.6×↑	7.1×↓		
	sed	✓	**I A S**	✓	✓	1.9×↑	8.1×↓		
	xargs	✓	**I A S**	✓	✓	1.6×↑	1.7×↓		

on the filesystem are identical to those of `ef`-free executions.

Performance: When compared to vanilla execution, `ef` introduces runtime overheads 1.1×–1.9×. Variations come from the complexity of effects interposed during execution. For example, in **`order`**, `ef` achieves the lowest overhead (1.1×) as it interposes on a single write effect; writing the sorted file to disk. The **`crawl`**, **`fresh`**, and **`archive`** tasks exhibit higher overheads (up to 1.9×) because they operate on many files from the host system (use of `find`), which results in more upperdir copy-ups. Relative to Docker-contained execution, `ef` demonstrates substantial performance improvements across all five benchmarks, achieving performance improvements of 1.3×–225.7×. Two benchmarks incur outlier speedups: the **`archive`** benchmark only interacts with its standard output stream (`find`'s output) which is effectively a symlink to the same file descriptor in the host environment, meaning that `ef` boils down to vanilla execution, paying only the flat cost of semisolate creation, while Docker still pays the full container setup and teardown costs. Similarly, the **`logs`** benchmark reads from and writes to a single large file, meaning that again `ef` pays only a single creation and copy-up cost, while Docker with full isolation needs to pay the copy-in and copy-out costs as well.

5.2 Dependency tracking

These benchmarks consist of several real-world shell programs with many and complex runtime dependencies between their components from the Koala suite of benchmarks [41]: (1) the **`covid`** benchmark, which processes bus schedule data collected during COVID-19 to generate summary statistics [81]; (2) the **`nlp`** benchmark, which performs natural language processing tasks on a text corpus [38]; (3) the **`spell`** benchmark, which checks spelling in a large text file [13]; and (4) the **`unixfun`** benchmark, which solves challenges from Unix 50th anniversary game [82]. These programs contain a diverse set of commands that include GNU Coreutils, POSIX, and third-party commands. To stack effects across all component invocations without applying them on the host environment (§2.1), `ef` was configured with `ef -L`, while also

having tracing enabled to extract dependencies (`-t`).

Control: All computations executed with `ef -x -n -L` mediated all effects that were applied to the filesystem by `ef`-free executions. Manual inspection and multiple re-executions confirm that executing these benchmarks with `ef -t -L` identifies all true dependencies between components.

Equivalence: Tracking the dependencies discovered by `ef -t` and later applying them with `ef -y -L` produces correct results. Their effects on the filesystem are identical to the ones produced by `ef`-free executions.

Performance: When compared to vanilla execution, `ef` introduces absolute overheads 1.2×–1.7×, which is the flat cost of setting up the semisolate environment for each tracked command. However, while the semisolate setup cost is flat, tracing overheads scale with the number of effects a component produces. This is less evident in statically compiled components, such as commands from GNU coreutils, which are tightly packed binaries with very minimal dependencies (often just `libc`), but evident with components relying on interpreting runtimes (*e.g.*, a custom `lowercase.py` utility in **`nlp`**), which need to resolve their entire dependency tree during execution. The performance of `ef` significantly outperforms fully-isolated execution across all four dependency tracking benchmarks, demonstrating speedups 1.5×–18.7×.

5.3 Third-party library risks

These benchmarks consist of `git` repositories with pre-commit hooks that introduce undesirable effects: (1) **`LinOTP`** [55] multi-factor authenticator; (2) the **`frogmouth`** [28] markdown reader; (3) Apache **`kibble`** [8] aggregating and visualizing software data; (4) the **`okteto`** [57] container orchestrator; (5) **`uv-metrics`** [63], a machine-learning metrics reporter. The pre-commit hooks perform several operations on the filesystem, communicate with the network, and include a code-injection attack that leads to data exfiltration. They were prefixed with `ef` configured as `ef -E`, excluding the `/etc/passwd` file from the semisolate.

Control: All computations executed with `ef -E` mediated across all effects present in the `ef`-free executions, except `/etc/passwd`. This effect originated from the pre-commit hook at `.git/hooks`, which attempted to append `/etc/passwd` to `README.md`.

Equivalence: All computations executed with `ef -E` produce correct results correctly commiting the intended changes, and the resulting file-system state remains identical to that of `ef`-free execution. Across both the vanilla pre-commit hook workloads and the five code-injection vulnerabilities evaluated, `ef` preserves correctness.

Performance: When compared to vanilla execution, `ef` has performance impact, up to a 1.3× overhead. The performance of `ef` is comparable to vanilla execution for the **`frogmouth`** and **`uv-metric`**, while maintaining low overheads for **`LinOTP`** (1.2×), **`okteto`** (1.3×), and **`kibble`** (1.2×), whose pre-commit hooks perform short-running computations, comparable with `ef`'s semisolate setup overhead. `ef` outperforms Docker-contained execution across all five benchmarks, achieving speedups 1.4×–4.2×. With blocking enabled, `ef` reduces the effect surface and thus computation performed by the injected pre-commit hooks. This leads to `ef` improving the runtime of some of the workloads (**`frogmouth`** and **`kibble`**), by 0.8× and 0.9× respectively.

5.4 Cautious software installation

These benchmarks evaluate `ef` on four NPM packages whose installation scripts perform unexpected file-system operations. The benchmark set includes: (1) **`eslint-scope`** [25], a package for analyzing the scope of JavaScript code; (2) **`coa`** [71], a package for handling command-line options; (3) **`node-sass`** [67], a package for compiling Sass to CSS; and (4) **`ua-parser-js`** [26], a package for parsing user-agent strings. All workloads use both `ef -y` and `ef -n`.

Control: All computations executed with `ef -n` mediated all effects present in the `ef`-free executions and prevented them from being applied to the host filesystem.

Equivalence: All computations executed with `ef -y` produce correct outputs, and the eventual file-system state result is identical to that of `ef`-free execution.

Performance: When compared to vanilla execution, `ef` demonstrates up to a 1.3× overhead. Overheads and speedups against full isolation have little variance across these benchmarks, as the installation scripts create a similar number of files inside the semisolate's `upperdir` (7391–8565 files), with overheads being closely correlated to the number of filesystem effects; `eslint-scope` creates the most files and is the closest to Docker-contained execution—still achieving a significant 4.8× speedup. In general, Docker-contained execution is outperformed by `ef` across all five installation benchmarks, achieving speedups 4.8×–5.8×.

5.5 Partial-specification mining

These benchmarks evaluate `ef`'s capability to support automatic specification mining for shell commands, focusing on determining the order of their read and write effects to their standard streams or the filesystem. These properties correspond to the ones required by PaSh [35] and POSH [61] to parallelize or distribute shell scripts. The commands evaluated in this section are: (1) **`ls`**, a command that lists directory contents; (2) **`cp`**, a command that copies files and directories; (3) **`rm`**, a command that removes filesystem entries; (4) **`sed`**, a stream editor that filters and transforms text using a script of editing commands; and (5) **`xargs`**, a higher-command that executes command lines from standard input. The specification miner explores 11, 100, 40, 1408, and 3 configurations

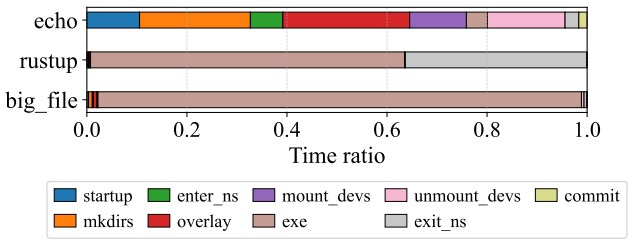

Figure 3: The performance breakdown of `ef`. The performance of `ef` in three microbenchmarks and its overhead breakdown across different execution stages.

for these commands respectively. For **`xargs`**, it uses the `rm`, `ls`, and `cat` commands as arguments. All workloads were executed using `ef -t`.

Control: All computations executed with `ef -t` mediate across all effects observed during command invocation, capturing every read and write operation performed by the evaluated commands.

Equivalence: All computations executed with `ef -t` produce correct results, and the resulting file-system state remains identical to that of `ef`-free execution. Across all commands evaluated, the inferred specifications match or extend the handwritten PaSh specifications. In three cases, `ef` enables the miner to detect effects omitted from the original PaSh specifications. For example, although **`cp`** is marked as side-effectful in the original specifications, its behavior in these evaluation scenarios is effectively pure, as it modifies only the files explicitly listed in its arguments.

Performance: Since the configuration miner needs to explore a large number of command invocations to infer their specifications, `ef`'s overheads compound significantly. Compared to vanilla execution, `ef` increases execution time 1.7×–8.3×, caused by `ef`'s flat cost of setting up the semisolate environment and its tracing overhead. Vanilla execution is not meaningful in this use case, as the specification miner needs to observe effects to infer specifications. The performance of `ef` improves relative to Docker-contained execution across all five mining benchmarks, achieving speedups 1.6×–2.2×. Benefits from `ef`'s lighter semisolate setup and teardown times compound over the many invocations required by the specification miner (here, 1562 in total).

5.6 Microbenchmarks

The goal of these benchmarks is to analyze `ef`'s sources of overhead by evaluating its performance on three synthetic workloads that create varying numbers and sizes of files. These benchmarks consist of three workloads: (1) **`echo`**, a Rust function that creates zero files, invoked 100 times; (2) **`rustup`**, a Rust function that creates 10,000 small files of size 16 bytes each; and (3) **`big_file`**, a Rust function that creates five large files of size 1GB each. All workloads were executed using `ef -y` and `ef -n`, and yield complete control and equivalence, hence we focus on performance breakdown (Fig. 3).

The performance analysis breaks down the `ef` runtime into nine stages: (i) *startup* parses command-line arguments and enumerates top-level directories; (ii) *mkdirs* provisions the upper, work, and temporary directories; (iii) *enter_ns* creates a fresh namespace; (iv) *overlay* establishes the OverlayFS mount points; (v) *mount_devs* bind-mounts selected device files; (vi) *exe* executes the target process; (vii) *unmount_devs* unmounts the device files; (viii) *exit_ns* exits the namespace, triggering the automatic unmount of remaining OverlayFS layers; and (ix) *commit* moves output files from the upper directory to the host filesystem. For the baseline **`echo`** command, which completes in 165ms, the overhead is caused by file-system and namespace operations. Creating the OverlayFS (*overlay*) accounts for 25% (42ms), followed by directory creation (*mkdirs*) at 22% (37ms). Device management also creates measurable overhead, with *unmount_devs* taking 16% (26ms) and *mount_devs* taking 11% (19ms). In longer-running benchmarks, such as **`big_file`** (9.2s) and **`rustup`** (11s), these setup and teardown costs remain roughly constant, making the relative overhead comparable to the actual *exe* time. However, **`rustup`** presents an outlier in the teardown phase as *exit_ns* takes 6.3s. This latency is caused because the kernel is unmounting the extensive number of OverlayFS mounts created during the installation process.

6 Discussion, Limitations & Experiences

Limitations: Current `ef` limitations primarily impact compatibility in specific scenarios: restrictions and altered application behavior due to user namespaces, challenges in handling pseudo-filesystems and physical devices, and inability to fully disable certain isolation mechanisms.

Due to user namespaces, `ef`-semisolated processes cannot interact with other users. Even when a process appears to be root in the namespace, it cannot switch UID or operate on files owned by other users. Components that rely on creating or switching users may behave unexpectedly.

Processes inside semisolates bypass writable permissions without needing to change the file permissions (a special feature of UID 0 in Linux). This may affect components that rely on write permissions or ones running without elevated privileges, such as `npm`, when the user ID is 0.

As `ef` requires user and PID namespaces for unprivileged OverlayFS mounting, it creates a non-configurable isolation barrier for IPC mechanisms such as signals, complicating interprocess coordination when such isolation is not desired.

External modules: Two additional modules offer functionality that augments or supplants `ef`.

To allow gaining root privileges, `ef` configures `unshare`

to map the current effective user and group ID (UID and GID) to the superuser UID and GID. Unfortunately, `unshare` does not support mapping multiple ranges of UIDs and GIDs, thus risking failure when traversing directories owned by a different GID, but within the permissions of the user entering the unshare. Such cases result in the Linux kernel throwing an `EOVERFLOW` error, as the group ID is not known in the new user namespace. To resolve this, `ef` introduces an external module that handles effective GID mapping. This module, called `ef-gidmapper`, identifies other GIDs that (1) exist in the system, and (2) belong to the calling user in the current namespace. As it extends the authority of the user namespace, the module incurs security implications that go beyond `ef`'s current scope.

As the size of effects can overwhelm a user, an external `ef` module summarizes effects in a human-friendlier form. This module, called `ef-summarize`, queries an opaque large-language model with the output from `ef` and additional instructions requesting (1) a non-expert summary of the effects, starting from the most important ones; (2) information about whether these effects touch sensitive directories. Both prompts offer functionality `ef`'s users could directly implement, but proved useful during recent user interactions with `ef`.

Experiences: Colleagues and students using `ef` in the past nine months have shared several experiences, a few of which are worth mentioning (see Appendix B for other testimonials completely unrelated to the `ef` authors).

Several colleagues used `ef` successfully in their grading infrastructure. A colleague grading distributed-systems projects was particularly happy that `ef` warned of buggy executions about to accidentally modify the grades of portions already graded—all executing in the same environment.

After a minicourse at the Data Science Institute of a large U.S. institution, `ef` was used by several scholars developing their Python skills. It saved a scholar developing a script for processing hieroglyphic images using a vision-language model, avoiding several image deletions from her own laptop caused by the model assigning duplicate names.

7 Related Work

Containment and virtualization: There has been a significant amount of work on full isolation, including in containers [9, 47, 72, 75] and virtual machines [10, 12, 16, 46, 58]. These primitives create an isolated environment that is different from the host environment they execute in, completely avoiding sharing of any sort. Semisolates instead run a program in a simulacrum of the current environment—as if on the same environment, but with significantly lower risk—and allow for further and selective manipulation of its effects.

Software isolation and confinement: Earlier work on isolation and confinement [4, 33, 36, 59, 64, 66], focused on controlling and if necessary protecting against harmful component effects. These approaches offer structures at the operating system level, often not supporting unmodified Linux environments. In contrast, `ef` leverages existing subsystems available in modern operating systems—requiring no modifications.

Other systems achieve confinement via system call interposition [14, 40]. For example, MBOX traces and interposes on system calls to inform the user of unintended or undesired effects. Such interposition meets only some of `ef`'s requirements (**I**, partly); it does not, for example, support stacking or further manipulation. Semisolates and `ef` take a different approach, offering several more features—and compatibility across several modern environments.

Language-based wrapping and proxying: Semisolates bear resemblance to earlier research that attempts to control effects and capabilities using dynamic effect wrapping [5, 19, 44, 48, 50, 51]. In contrast to `ef`, these approaches are not language-agnostic, as they are tightly integrated into a language and its runtime environment [5, 18, 19, 49], and often require developer effort through code modifications, wrapping mechanisms, or isolation policies [43, 50–52, 79]. Semisolates offer language-agnostic effect control and support completely unmodified programs.

Filtering and access prevention: Another class of sandboxing tools is based on software fault isolation [84], including NaCl [70], WASM [30], and RLBox [54]. These systems focus on intraprocess memory-level isolation. Instead, `ef` supports effect control (not isolation) outside the process—including effects such as file modifications.

Other systems [65, 85, 86] support effect control by preventing file accesses, terminating processes that access files outside of an allowed set. And system call filtering [20, 29, 39, 73] can also be used to control effects by preventing untrusted applications from accessing dangerous or unnecessary system calls. These approaches prevent effects completely—possibly breaking the application if it needs to perform the effects to complete its execution. In contrast, semisolates allow the target application to perform all of its effects and can later selectively prevent some of them from being committed.

Transactional filesystems: Earlier research on transactional filesystems [24, 68, 69, 76] has developed support for atomically commiting or reverting changes to multiple files in a filesystem. This is contrary to `ef`, which allows calling programs to inspect and apply some of the effects—or further manipulate them at will.

8 Conclusion

Software systems consist of many components, written in many languages, and interacting with each other and the environment. This paper showcases the need for and feasibility of building a new abstraction, the semisolate, for controlling potentially undesired effects, maintaining full compatibility with real-world components, and incurring a performance

overhead that is aligned with several practical applications.

References

[1] eslint-scope vulnerability. https://security.snyk.io/vuln/SNYK-JS-ESLINTSCOPE-11120, 2024. Accessed: 2024-09-29.

[2] Claude code. https://claude.com/product/claude-code, 2025. Accessed: 2025-12-10.

[3] Codex: Lightweight coding agent that runs in your terminal. https://github.com/openai/codex, 2025. Accessed: 2025-08-29.

[4] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Technical Conference*, 1986.

[5] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In *Annual Computer Security Applications Conference (ACSAC)*, pages 1–10, 2012.

[6] Ali Sajid. How to sort a file in-place?, 2015. Accessed: 2025-12-04.

[7] Cursor / Anysphere. Cursor: The best way to code with ai, 2025. Accessed: 2025-09-25.

[8] Apache Kibble Developers. Apache kibble.

[9] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[11] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2021.

[12] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[13] Bentley, Jon and Knuth, Don and McIlroy, Doug. Programming pearls: a literate program. *CACM*, 29(6):471–483, June 1986.

[14] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 27–38, 2015.

[15] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[16] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 2–13, New York, NY, USA, 2008. Association for Computing Machinery.

[17] MITRE Corporation. Cve-2022-23812: Vulnerability details, 2025. Accessed: 2025-01-02.

[18] Tom Van Cutsem. Membranes in javascript. https://tvcutsem.github.io/js-membranes, 2012. Accessed: 2020-03-16.

[19] Willem De Groef, Fabio Massacci, and Frank Piessens. NodeSentry: Least-privilege Library Integration for Server-Side JavaScript. In *Annual Computer Security Applications Conference (ACSAC)*, pages 446–455, 2014.

[20] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 459–474, 2020.

[21] Linux Kernel Documentation. mount(2) — linux manual page, 2024. Available at https://man7.org/linux/man-pages/man2/mount.2.html.

[22] Linux Kernel Documentation. Overlayfs - linux kernel documentation, 2024. Available at https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html.

[23] The ef authors and contributors. Ef: Controlling side effects with semisolates, 2025. Accessed: 2025-09-25.

[24] Robert Escriva and Emin Gun Sirer. The design and implementation of the warp transactional filesystem. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 469–483, Santa Clara, CA, March 2016. USENIX Association.

[25] ESLint. eslint-scope - ECMAScript scope analyzer, 2025. Accessed: 2025-12-04.

[26] Faisal Salman. UAParser.js - The Essential Web Development Tool for User-Agent Detection, 2025. Accessed: 2025-12-04.

[27] fegome90-cmd. Cursor ai executes destructive command during development session. https://forum.cursor.com/t/cursor-ai-executes-destructive-command-rm-rf-during-development-session/129401, 2025. Accessed: 2025-09-24.

[28] Frogmouth Developers. Frogmouth.

[29] Alexander J Gaidis, Vaggelis Atlidakis, and Vasileios P Kemerlis. Sysxchg: Refining privilege with adaptive system call filters. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1964–1978, 2023.

[30] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[31] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin C. Rinard. An order-aware dataflow model for parallel unix pipelines. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021.

[32] IAspireToBeGladOS. How to compress files larger than a certain size in a directory?, 2017. Accessed: 2025-12-04.

[33] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan M. Smith. Sub-operating systems: A new approach to application security. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 108–115, New York, NY, USA, 2002. ACM.

[34] john. How can I recursively find all files in current and subfolders based on wildcard matching?, 2011. Accessed: 2025-12-04.

[35] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. Practically correct, Just-in-Time shell script parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 769–785, Carlsbad, CA, July 2022. USENIX Association.

[36] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[37] KCD. Regex with grep, 2021. Accessed: 2025-12-04.

[38] Kenneth Ward Church. Unix for Poets, 1994.

[39] The Linux Kernel. Seccomp bpf (secure computing with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html, 2023.

[40] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 139–144, 2013.

[41] Evangelos Lamprou, Ethan Williams, Georgios Kaoukis, Zhuoxuan Zhang, Michael Greenberg, Konstantinos Kallas, Lukas Lazarek, and Nikos Vasilakis. The koala benchmarks for the shell: Characterization and implications. In *2025 USENIX Annual Technical Conference (USENIX ATC '25)*, pages 449–64, Boston, MA, July 2025. USENIX Association.

[42] Lukas Lazarek, Seong-Heon Jung, Evangelos Lamprou, Zekai Li, Anirudh Narsipur, Eric Zhao, Michael Greenberg, Konstantinos Kallas, Konstantinos Mamouras, and Nikos Vasilakis. From ahead-of- to just-in-time and back again: Static analysis for unix shell programs. In *2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, page 88–95, New York, NY, USA, 2025. Association for Computing Machinery.

[43] Sergio Maffeis, John C Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *2010 IEEE Symposium on Security and Privacy*, pages 125–140. IEEE, 2010.

[44] Jonas Magazinius, Daniel Hedin, and Andrei Sabelfeld. Architectures for Inlining Security Monitors in Web applications. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 141–160, 2014.

[45] Vahid Majdinasab, Michael Joshua Bishop, Shawn Rasheed, Arghavan Moradidakhel, Amjed Tahir, and Foutse Khomh. Assessing the security of github copilot's generated code - a targeted replication study. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 435–444, 2024.

[46] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, page 6–es, New York, NY, USA, 2007. Association for Computing Machinery.

[47] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[48] Leo A Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *2010 IEEE Symposium on Security and Privacy*, pages 481–496. IEEE, 2010.

[49] James Mickens. Pivot: Fast, synchronous mashup isolation using generator chains. In *2014 IEEE Symposium on Security and Privacy*, pages 261–275. IEEE, 2014.

[50] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized javascript, 2008. *Google white paper*, 2009.

[51] Mark S Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in javascript. In *European Symposium on Programming*, pages 1–20. Springer, 2013.

[52] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. Shill: a secure shell scripting language. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 183–199, USA, 2014. USENIX Association.

[53] Antonio SJ Musumeci. mergerfs: a featureful union filesystem, 2012–2025. Accessed: 2025-01-14.

[54] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716, 2020.

[55] netgo software GmbH. LinOTP.

[56] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. *ACM SIGOPS Operating Systems Review*, 39(5):191–205, 2005.

[57] Okteto. Okteto - Develop your applications directly in your Kubernetes Cluster, 2025. Accessed: 2025-12-04.

[58] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, November 2015.

[59] NSA Peter Loscocco. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track:... USENIX Annual Technical Conference*, page 29. The Association, 2001.

[60] David Quigley, Josef Sipek, Charles P Wright, and Erez Zadok. Unionfs: User-and community-oriented development of a unification filesystem. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 349–362, 2006.

[61] Raghavan, Deepti and Fouladi, Sadjad and Levis, Philip and Zaharia, Matei. POSH: a data-aware shell. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[62] RIAEvangelist. node-ipc: Inter-process communication for node.js, 2025. Accessed: 2025-01-02.

[63] Sam Ritchie. UV Metrics: Metric reporting and experiment management for ml workflows., 2020.

[64] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[65] Mickaël Salaün. Landlock lsm: toward unprivileged sandboxing. *Linux Security Summit*, 2017.

[66] Jerome H Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

[67] Sass. Node-sass - Node.js bindings to LibSass, 2025. Accessed: 2025-12-04.

[68] Frank Schmuck and Jim Wylie. Experience with transactions in quicksilver. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 239–253, 1991.

[69] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 29–44, 2006.

[70] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[71] Sergey Berezhnoy. Command-Option-Argument: Yet another parser for command line options, 2025. Accessed: 2025-12-04.

[72] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.

[73] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. Draco: Architectural and operating system support for system call security. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 42–57. IEEE, 2020.

[74] Snyk. Snyk vulnerability: Snyk-js-nodeipc-2426370, 2025. Accessed: 2025-01-02.

[75] Stephen Soltesz, Herbert Potzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 275–287, New York, NY, USA, 2007. Association for Computing Machinery.

[76] Richard P Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P Wright. Enabling transactional file access via lightweight kernel extensions. In *FAST*, volume 9, pages 29–42, 2009.

[77] ESLint Team. eslint-scope: Scope analysis for javascript, 2025. Accessed: 2025-01-02.

[78] Teekin. Unix shell script: Update timestamp on all subdirectories and sub-files, including those with spaces, 2010. Accessed: 2025-12-04.

[79] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. Javascript in javascript (js. js): sandboxing third-party scripts. In *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*, pages 95–100, 2012.

[80] Lillian Tsai and Eugene Bagdasarian. Contextual agent security: A policy for every purpose. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, page 8–17, New York, NY, USA, 2025. Association for Computing Machinery.

[81] Tsaliki, Eleftheria and Spinellis, Diomedes. The Real Numbers for Athens Buses, 2020.

[82] Unix Game. The Unix Game - 50 Challenges to Master the Command Line, 2024. Accessed: 2025-10-19.

[83] u/Zashuiba. Tifu by copypasting code from ai. lost 20 years of memories. https://www.reddit.com/r/homelab/comments/1jmlj9f/tifu_by_copypasting_code_from_ai_lost_20_years_of/, 2023. Accessed: 2025-09-24.

[84] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[85] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, volume 8032, pages 6–16, 2002.

[86] Wenhui Zhang, Peng Liu, and Trent Jaeger. Analyzing the overhead of file protection by linux security modules. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 393–406, 2021.

A LLM-Generated Scripts

This appendix lists the exact prompts used to generate the LLM-based benchmarks in §5.1, along with the scripts returned by ChatGPT.

`crawl`: Find & Touch

PROMPT

Write a simple Bash script that searches a directory (for example ./data) and runs a command on every file it finds. Use find with -exec and include comments explaining what the script does. The command should just touch each file.

GENERATED SCRIPT

```
directory="./data"
find "$directory" -type f -exec touch "{}" \;
```

`fresh`: Compress Files Larger Than 1 KB

PROMPT

Write a Bash script that finds all files larger than 1 KB in a directory (e.g., ./data) and compresses them using gzip. Use find with the -size filter and an -exec command. Include comments.

GENERATED SCRIPT

```
directory="./data"
find "$directory" -type f -size +1k \
  -exec sh -c 'gzip "{}"' \;
```

`archive`: Find All .txt Files

PROMPT

Give me a command to find all .txt files in the current directory and its subdirectories.

GENERATED SCRIPT

```
find . -name "*.txt"
```

`logs`: Extract Error Lines From Log

PROMPT

Give me a command to extract only ERROR, WARNING, or CRITICAL lines from a log file and save them to a new file. Make sure the command doesn't fail even if nothing is matched.

GENERATED SCRIPT

```
grep -E "ERROR|WARNING|CRITICAL" file.log \
  > filtered_errors.log || true
```

`order`: Sort a File

PROMPT

Give me a command to sort the contents of book.txt and save the result to another file.

GENERATED SCRIPT

```
sort book.txt > sortedBook.txt
```

B Testimonials

This appendix includes testimonials from users who have tried `ef` and shared their feedback on social media platforms. They offer a snapshot of real-world experiences and community reception of the tool. The feedback was primarily gathered from platforms such as personal blogs and Hackernews.

Looks nifty—basically it's the "let me try that in a container first" except on your live system with no setup to get it going.

— **@mikepurvis**, *Hackernews*

This is a much needed software.

— **@johnsbullings**, *Hackernews*

Very interesting! This feels like something that should have always existed.

— **@Benjamin Oakes**, *benjaminoakes.com*

This is cool I probably will use this.

— **@jagtstronaut**, *Hackernews*

I love that their demo is to finally give me a dry run command for pip.

— **@gdevenyi**, *Hackernews*

The script is mounting an overlay filesystem for each root directory on the current system, then looking at what has been added as a new layer... Pure genius.

— **@MichaelMoser123**, *Hackernews*

I can see this being incredibly useful.

— **river**, *lobste.rs*

`ef` is clever.

— **Jim Fowler**, *x.com*

I look forward to it helping me prevent programs polluting my laptop.

— **bfiedlera**, *lobste.rs*

This tool can bring a level of control that is typically only available in databases and VCSs, and make it a commodity.

— **@klabb3**, *Hackernews*