APPENDIX A
CLEARS IMPLEMENTATION DETAILS

In this section, we elaborate on the system-level details of the design used to implement a prototype of CLEARS. This implementation operates entirely within the user space of the UNIX system, without altering the kernel space or any of its system programs.

`clears` Command Line Tool. We have developed a command line interface (CLI) tool named '`clears`' that encompasses all functionalities necessary to support the concepts proposed in our model, detailed in Section IV. This CLI serves as the primary interaction point for both system administrators and users within the system. The '`clears`' tool includes several commands, each corresponding to specific actions supported by the model. Snapshots of all executed actions can be found in the code repository.A concise overview of the CLI's functionalities and required parameters can be found in Table IV. As the tool interacts with some of the system configuration files, such as scheduler configuration, user and group information, etc, it must be owned by the *root* user with *setuid* enabled, which allows the model to run with elevated privileges irrespective of who is executing it. It is crucial to note that, according to the model, non-administrative users are limited to performing share and unshare privileges. All other actions require administrative privileges and are intended for system administrators.

Reference Files. `clears` interacts with several different configuration files and services to execute the desired behaviors of the proposed model for efficient privilege management using the per-project collaboration network.

To begin with, the per-project collaboration network and its associated shared privileges are maintained in JSON format, stored in the directory `/etc/project/`. Each ongoing project has a corresponding JSON file named after the project. This directory is created by system administrators, with default ownership assigned to the *root* user. Default permissions are configured so that the directory and its files are world-readable but only writable by the root user. The JSON files for each project are automatically generated by the model when the administrator executes the *start* command. Subsequently, for all other commands, `clears` either reads from or writes to the JSON files located under `/etc/project/` to maintain the shared privileges within each project. Thus, inspecting the JSON file at any given point provides a clear and comprehensive overview of the shared privileges within that project.

The JSON format is chosen for its compatibility with Python3 and its ease of readability. However, future work may involve generating visualization graphs based on these JSON files to enhance the understanding of shared privileges. An example of a collaboration network JSON file can be found in Listing 1. To optimize space, we only dynamically maintain the 'active' collaboration contexts — those within which at least one privilege has been shared — instead of creating and maintaining all possible contexts. Moreover, each collaboration context is designated by an *id* created by concatenating the *uid*s of all involved users. The snapshot in Listing 1 captures the status of the collaboration network for `Project1`, involving users $alex(uid = 10001), bailey(uid = 10002)$, and $cathy(uid = 10003)$. In this scenario, *alex* has shared access to `/scratch/alex` with both *bailey* and *cathy*, and `alex_partition1` with just *bailey*.

Listing 1: Sample Collaboration Network JSON File

```
# File: /etc/project/Project1.json
{
    "project_id": "Project1",
    "all_user_ids": ["alex","bailey","cathy"],
    "contexts": {
        "100011000210003": {
            "id": "100011000210003",
            "user_ids": ["alex","bailey","cathy"],
            "resource_ids": [
                [1, "/scratch/alex"]
            ]
        },
        "1000110002": {
            "id": "1000110002",
            "user_ids": ["alex","bailey"],
            "resource_ids": [
                [2, "alex_partition1"]
            ]
        }
    }
}
```

As the core idea of CLEARS is to assign privileges to the collaborations within a project, it is crucial to represent the collaborations in a way that allows for privilege assignment. Conceptually, collaborations are collections of users with added context, so we represent them using UNIX groups. Since the same collaboration can appear in multiple projects, we represent these groups by prefixing them with the *project_id*. For example, referring to Listing 1, the context 10011002 within $Project1$ represents a UNIX group in the system named $Project110011002$. Therefore, the corresponding entry for this group would look like this, where 10009 is a system-assigned *gid* for this UNIX group.

```
Project110011002::10009:alex,bailey
```

Since we are dealing with RCI clusters, which consist of multiple interconnected Unix systems, maintaining uniform group information across the cluster is essential. Instead of relying on local system files like /etc/group, a centralized LDAP server is adopted to synchronize group information. Therefore, `clears` interfaces with the LDAP server to perform various group management tasks. It dynamically creates new groups, assigns users to these groups, and removes groups as needed.

Lastly, to enforce these shared privileges, `clears` directly interacts with the necessary configurations depending on the resource type. For files and directories, it creates a new entry in the access control list (ACL) of the file/directory to assign the shared privileges to a collaboration (UNIX group, in implementation). For computing resources, such as partitions managed by schedulers like SLURM, it updates the scheduler's configuration to ensure proper identification of the shared

Command	Function	Required Parameters	'sudo' access
start	Start a new project	project_id	Required
end	End an existing project	project_id	Required
add	Assign users to an existing project	project_id, set of usernames	Required
remove	Remove users from an existing project	project_id, set of usernames	Required
share	Share privileges with a collaborator to access resources within a project	project_id, resource_type, resource_name, set of usernames	Not Required
unshare	Retract previously shared privileges from a collaborator within a project	project_id, resource_type, resource_name, set of usernames	Not Required

TABLE IV: `clears` CLI commands, their functions, and required parameters.

privileges. For instance, considering the example provided in Listing 1, the two shared privileges would be translated into the following entries:

```
# ACL: /scratch/alex
group:Project1100110021003:rwx
...
# /etc/slurm/slurm.conf
PartitionName=alex_partition1
AllowGroups=alex,Project110011002
```

This mechanism ensures that there is always a collaboration context between the users and the privileges. Therefore, any subsequent actions such as *unshare*, *remove*, or *end* will either remove these collaboration contexts or the privileges assigned to them, automatically revoking the privileges from the user.

Figure 5 shows the sequence diagram of interactions between the Admin, the `clears` , and the system configuration files during the execution of administrative actions: *start*, *add*, *remove*, and *end*. Similarly, Figure 6 illustrates the interactions for user actions *share* and *unshare*. The `clears` processes each action by interacting with system files and configurations, running necessary algorithms, and making decisions to return `SUCCESS` or `ERROR` responses based on the conditions evaluated. For user actions, it also internally evaluates the preconditions using *can_share* and *can_unshare* functions to authorize the actions before executing them.

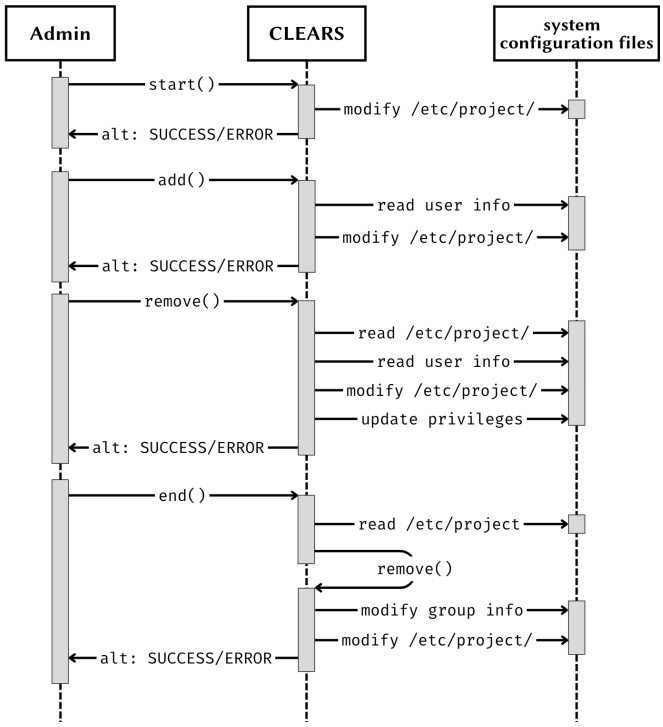

Fig. 5: Sequence Diagram for Admin-only Actions in the Authorization Model.

APPENDIX B
EXPERIMENTAL ENVIRONMENT SETUP

To emulate the actual RCI we studied, we created a small-scale controlled environment to test the feasibility of our implemented access control framework. This environment consists of four virtual machines—*linux*0, *linux*1, *linux*2, and *linux*3—each running Ubuntu 22.04.03 LTS Server and connected through a bridged network using an Intel PRO/1000 MT Desktop adapter.

Shared Network File System. The *linux*0 machine serves as the host node for the shared network file system (NFSv4) to keep necessary files and directories synchronized across the nodes. The directories hosted in *linux*0 are therefore mounted on $linux[1-3]$ to keep these resources and configurations synchronized. The following directories are hosted on the NFS:

- `/home/`: Home directories of each user in the system.
- `/scratch/`: Scratch directories of each user in the system.
- `/data/`: Additional data storage directories associated with a specific user or group in the system.
- `/etc/project/`: The project-specific collaboration network files for active projects in the system.

Shared User and Group Information. To maintain consistency in users and groups across VMs and streamline user authentication, we employ LDAP in our environment setup, with a centralized LDAP server hosted at `linux0`, and LDAP clients configured on `linux[1-3]`. Within these systems, we modify the entries associated with `passwd` and `group` in the local Name Service Switch configuration file `/etc/nsswitch.conf` to be `compat ldap`. This configuration ensures that the systems first look for users and group information in local files (like `/etc/passwd` and

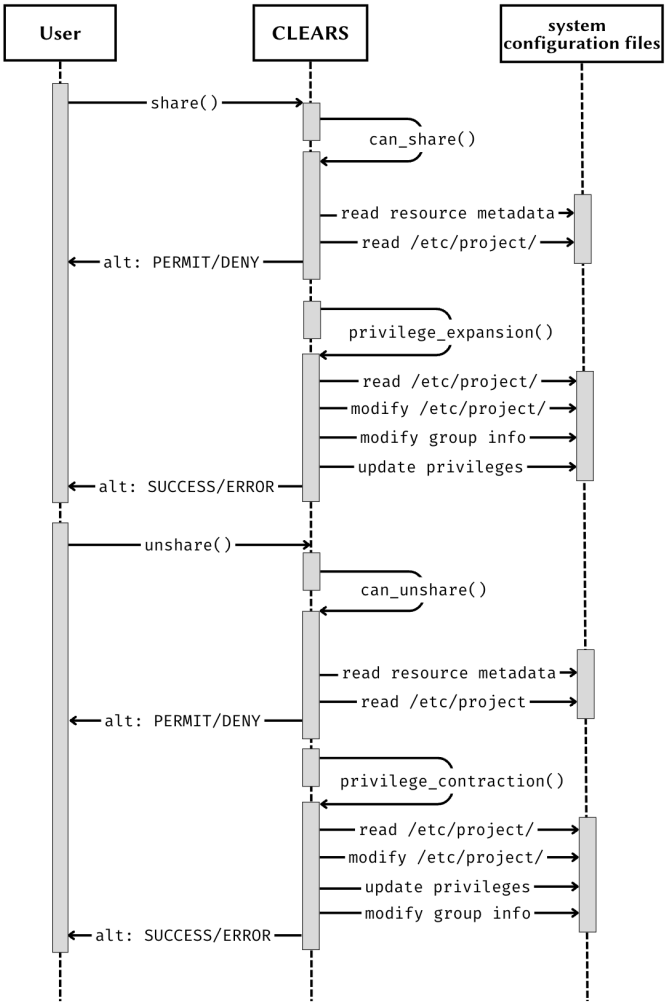

Fig. 6: Sequence Diagram for Owner-centric Actions in the Authorization Model.

`/etc/group`), and if not found, they consult LDAP.

Job Scheduler Configuration. SLURM is employed as the job scheduler, chosen for its widespread adoption in RCIs, including the infrastructure under study. On `linux0`, both the SLURM controller daemon (`slurmctld`) and the SLURM database daemon (`slurmdbd`) run. Meanwhile, `linux1` and `linux2` function as 'compute' nodes within the SLURM configuration. These nodes are configured to restrict remote access via PAM. Additionally, `linux3` is set up as a 'login' node, accessible directly through remote login. All of these nodes run the SLURM daemon (`slurmdbd`). According to official SLURM documentation, the SLURM configuration file `/etc/slurm/slurm.conf` is synchronized across all nodes. The following way the computing nodes are configured:

- `linux1` is configured within a public partition, with parameters `AllowGroups`, `AllowAccounts`, and `AllowQos` all set to `ALL`, ensuring no restrictions on job requests in this partition, aligning with its public nature.
- `linux2` is configured within an exclusive access partition, featuring default settings of `AllowGroups=owner_group`, and both `AllowAccounts`, and `AllowQos` as `ALL`. This setup restricts access to the partition, ensuring only users in the `owner_group` by default, can request jobs in this partition.
- `linux3` is configured within a hidden login partition, ensuring the scheduler does not admit any jobs to this node.

Within the configuration, only three parameters — `AllowGroups`, `AllowAccounts`, and `AllowQos` — are explicitly configured, while all other parameters remain at their default values.

Users, Resources, and Default Permissions. One administrative account, `sysadmin`, is created with administrative (*sudo*) privileges, along with several non-administrative user accounts. It is assumed that each account is used by a single individual, allowing the terms "user" and "account" to be used interchangeably. Resources within the system are managed by the system administrator (*sysadmin*), who can further explicitly designate a single user as the owner of a particular resource (§ IV), meaning each resource can have only one designated owner. For files and directories, ownership is specified in the file metadata (using the `chown` command). For computational partitions, ownership is denoted by prefixing the partition name with the username. For instance, a partition named `alex_partition1` indicates that the user *alex* owns the partition and can share access privileges to it according to the model.

Each user has a *home* directory and a *scratch* directory named after their username. Some users also have a *data* directory and a computational partition they own. This setup mirrors real-world scenarios as observed in our study. Default permissions for files and directories are configured to allow access only to the respective owner and the root user with elevated privileges. Additionally, the *umask* is set to ensure that newly created files and directories inherit these default permissions. Default permissions for the computational partitions are previously discussed.