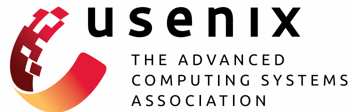

Application-Defined Decentralized Access Control

Yuanzhong Xu and Alan M. Dunn, ***The University of Texas at Austin;*** **Owen S. Hofmann,** ***Google, Inc.;*** **Michael Z. Lee, Syed Akbar Mehdi, and Emmett Witchel,** ***The University of Texas at Austin***

https://www.usenix.org/conference/atc14/technical-sessions/presentation/xu

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

Application-Defined Decentralized Access Control

Yuanzhong Xu Alan M. Dunn Owen S. Hofmann * Michael Z. Lee
Syed Akbar Mehdi Emmett Witchel
The University of Texas at Austin *Google, Inc.**

Abstract

DCAC is a practical OS-level access control system that supports application-defined principals. It allows normal users to perform administrative operations within their privilege, enabling isolation and privilege separation for applications. It does not require centralized policy specification or management, giving applications freedom to manage their principals while the policies are still enforced by the OS. DCAC uses hierarchically-named attributes as a generic framework for user-defined policies such as groups defined by normal users. For both local and networked file systems, its execution time overhead is between 0%–9% on file system microbenchmarks, and under 1% on applications.

This paper shows the design and implementation of DCAC, as well as several real-world use cases, including sandboxing applications, enforcing server applications' security policies, supporting NFS, and authenticating user-defined sub-principals in SSH, all with minimal code changes.

1. Introduction

Continued high-profile computer security failures and data breaches demonstrate that computer security for applications is abysmal. While there is extensive research into novel security and access control models little of this work has an impact on practice. Instead of applications consistently reimplementing security vulnerabilities, they need a practical and expressive way to use thoroughly debugged system-level primitives to achieve best security practices.

DCAC (DeCentralized Access Control) is our attempt to make modern security mechanisms practical for access control. It has three distinguishing characteristics: it is decentralized in privilege, decentralized in policy specification, and allows application-defined principals and synchronization requirements. Although DCAC greatly increases the flexibility of access control, it retains a familiar model of operation, with per-process metadata checked against per-object ACLs to determine the allowed access. It relies on the standard OS infrastructure of a hierarchical file namespace, extended file attributes, and file descriptors. It is practical for distributed environments because it avoids requiring centralized storage, consistency, or management.

Decentralized privilege. In Linux and Windows, users and groups are principals, and can be assigned privileges. A user might consider creating another user (a "sub-principal") and assigning it a subset of her privileges. This allows an application to run as the sub-principal, and thus with restricted privileges compared to the case where the user directly runs the application. However, on Linux and Windows, administrative functions on users and groups require root privilege. As a result, current OS-level access control does not allow many applications to run with least privilege.

DCAC decentralizes administrator privilege: a normal user can perform administrative operations within her privilege, like creating principals with subsets of her privilege. Privilege separation makes complex applications more difficult to exploit. But current systems require administrative involvement to install and deploy privilege-separated software.

For example, the suEXEC feature of Apache HTTP Server allows it to run CGI and SSI programs under UIDs different from the UID of the calling web server, by using `setuid` binaries. However, creating UIDs for CGI/SSI programs and setting up the `setuid` binaries requires administrator privilege. Not only can use of administrative privilege require human involvement, it also adds opportunities for configuration mistakes that can actually *harm* security. The suEXEC documentation[1] warns the user, *"if suEXEC is improperly configured, it can cause any number of problems and possibly create new holes in your computer's security. If you aren't familiar with managing setuid root programs and the security issues they present, we highly recommend that you not consider using suEXEC."* By contrast, DCAC allows forms of privilege separation, like delegating user privileges to sub-principals, that even in the case of a configuration mistake, limit the effect of a compromise to the privileges of the original user.

Decentralized policy specification. OS-level access control typically defines its principals and policies in a centralized, secure location, such as the `/etc/group` file, the `policy.conf` file in SELinux, or a central policy server (e.g., a Lightweight Directory Access Protocol (LDAP) server). DCAC decentralizes policy specification: policies are stored in files and file metadata at arbitrary locations. DCAC generalizes the `setuid` mechanism of Unix, allowing processes to use the file system

* Work completed while at the University of Texas at Austin.

[1] `http://httpd.apache.org/docs/2.4/suexec.html`

to gain fine-grained, user-defined privileges (i.e., not just root). With DCAC, applications control their privileges with a mechanism implemented and enforced by the operating system, but without central coordination.

DCAC is particularly practical for distributed environments, e.g., where machines share a file system via NFS. In such an environment, applications simply use the file system to express access control policy, and any host that mounts the file system will enforce identical access control rules. DCAC does not add its own synchronization requirements, such as entries in a central database. Applications make all access control decisions with access only to their own files. In contrast, a centralized policy server might become a bottleneck when policy queries and updates are frequent, as in many server applications.

Application-defined principals: attributes. Attributes make applications simpler and more secure by allowing them to use access control implemented by the operating system rather than reimplementing their own. Traditional OS principals, such as users, are heavy-weight abstractions that cannot be directly used by applications e.g., a web application that manages its own users.

DCAC *attributes* are hierarchically named strings. Strings are separated into components by the "." character. The string $.u.alice$ can represent the user Alice, but applications are free to define their own encodings and even their own principals. For example, the string $.p.387.1357771171$ might be a principal referring to a process with identifier 387 started about 1.4 billion seconds after January 1, 1970; $.app.browser.password$ might be a component of a browser that is responsible for storing and retrieving the user's passwords. A process may carry multiple attributes simultaneously.

Practicality. DCAC combines ideas from mandatory access control (MAC) systems [7, 16], sandboxing [6], and decentralized information flow control (DIFC) [9, 15, 17, 21, 29] into a practical access control system that is fully backward compatible with current Linux. While MAC and DIFC systems can provide stronger guarantees than DCAC, they require far more effort to use and often struggle with backward compatibility. We believe that application developers will incrementally improve the security of their applications if presented with a simple security programming model that introduces a minimum of new concepts and that can implement security idioms common in modern web-connected applications.

The next section (§2) provides motivating scenarios for DCAC, followed by an extended discussion of design (§3) and relationship to Linux DAC (§4). We describe our DCAC prototype (§5), with a discussion of several applications we modified to use DCAC (§6). We evaluate DCAC (§7), discuss related work (§8) and conclude (§9).

2. Modern access control idioms

We discuss three access control idioms common in today's web-connected applications that are difficult to achieve in modern systems: sandboxing, ad hoc sharing, and managing users. Because these idioms are not well served by current system security abstractions, applications constantly reimplement these idioms, and implement them poorly. Section 3.2 shows how DCAC supports them more naturally than current systems.

Privilege separation/Sandboxing. Suppose Alice wishes to run a photo management program. By default, her program will run with the same privileges as her user account. However, routines for interpreting file formats are often subject to exploitable bugs (such as in the zlib library used to decompress `.png` files [1]). If Alice receives photos from untrusted sources, or even if Alice makes a mistake, all of her potentially sensitive files are endangered, rather than just those that should be managed by her photo management application. Running untrusted or partially-trusted applications has become commonplace, as users frequently download applications from less than trusted sources, or run applications that are often exploitable, such as pdf viewers.

In order to separate her application into a separate privilege domain, Alice must contact an administrator to create a new user (e.g. *alice-photos*), potentially create a new group containing both her and *alice-photos* so that she may easily share files, and install support for running her desired application as the new user, such as via a setuid binary. While it is possible to enforce privilege restriction without superuser support, solutions that do so require complex application-level support (such as in the Chromium web browser [6]) that are easy to get wrong, with disastrous results (e.g., the frequent exploits enabled by bugs in the Java VM's sandboxing mechanism [2, 3]).

This scenario is an example of privilege separation, a well-known technique for building secure applications [20], where each component has only the minimal set of privileges necessary to operate. Privilege separation with OS support is coarse-grained, such as user-controlled namespaces [13] (e.g., namespaces can control access to mount points, but not files or directories).

Ad hoc sharing. Existing systems provide limited facilities for sharing between users. Suppose Alice wishes to share a file or directory with Bob. She can directly send the relevant data to Bob, but if both users wish to be able to update the data they must manually communicate each change. She might change file or directory permissions to allow read/write access to a group that contains both Alice and Bob. Unless this specific group already exists, Alice must rely on a system administrator to create a new group containing both users. If she wishes to add or remove users, she must also rely on a superuser.

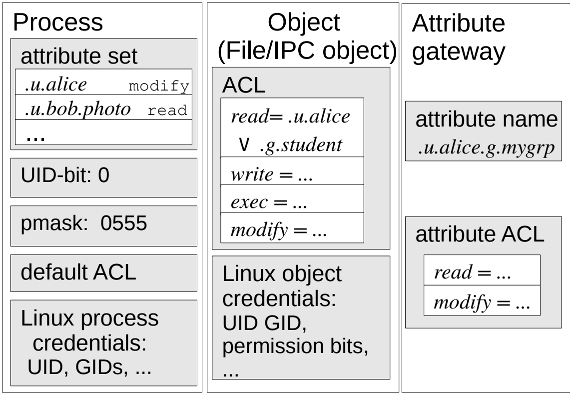

Figure 1: Overview of processes, objects and attribute gateways in DCAC.

If the need to share files expires, an administrator must clean up stale groups.

Facilities such as POSIX ACLs [4] allow users to create more expressive access control lists, by specifying multiple users and groups who may read and write a file. If Alice wishes to share with many users, however, she must either add each user to an ACL for each file to be shared, or again use a group whose membership may only be modified by the superuser. Users should be able to define new groups of users without administrator support, and use those groups in access control on their files.

Managing users. Server applications often have access control requirements similar to those of operating systems, with multiple users having different privileges. However, such applications usually implement access control with manual checks in their own code. Custom security code is a common source of security bugs: a single missed access check can expose sensitive data to unprivileged users. However, most non-trivial applications leveraging OS access control for application security must execute as different OS-level principals and hence require superuser privilege. Such an application must reserve user or group identifiers for its use during installation and possibly during maintenance.

3. Design

In DCAC, the OS enforces simple rules about hierarchical strings (called *attributes*) that are stored in OS-managed process and file metadata. Applications define conventions for what attributes mean to them, allowing them to create access control abstractions that are enforced by the OS.

Hierarchical strings are a self-describing mechanism to express decentralized privilege, where any extension of an existing string represents a subset of the parent's privileges. DCAC attributes have components separated by a "." character: $.u.alice$ is a parent attribute of $.u.alice.photo$. The hierarchy allows regular users to manage principals and policies without requiring a system administrator. For example, Alice may define a new principal for running her photo application (e.g., $.u.alice.photo$) because she owns and controls $.u.alice$.

Each process carries an *attribute set*, which is inherited across process control events such as `fork()` and `exec()`. A process also maintains a *default ACL*. Similar to Linux's `umask`, files created by the process have their access control lists set to the process' default ACL.

Each object (such as a file or shared memory segment) has an *access control list* containing rules that allow processes to access the object based on attributes in the processes' attribute sets. An access control list specifies four *access modes*: read, write, execute, and modify. Each mode has an *attribute expression*, and a process may access a file in a given mode if the process' attribute set satisfies the attribute expression for that access mode. For example, the read access mode might specify $.u.alice \vee .g.student$ which provides read access to user Alice and members of group student (see the read access mode of the object in Figure 1).

Read, write and execute access modes represent the `rwx` permission bits in UNIX-like file systems. The modify mode specifies the permission to modify the file's access control list. In UNIX-like systems, the file's owner implicitly has the right to modify a file's access control.

In DCAC, each access mode has an attribute expression in disjunctive normal form (DNF) without negations. For example, a `jpg` file may have the following access modes:

$$\begin{aligned} read &= (.u.alice.photo) \vee (.u.bob.photo) \\ write &= (.u.alice.photo \wedge .u.alice.edit) \\ exec &= \varnothing \\ modify &= (.u.alice) \end{aligned}$$

If a process has attribute set $\{u.bob.photo\}$, it can read this `jpg` file, but cannot write to it or modify its ACL.

Broadly, DCAC defines rules similar to existing discretionary access control (DAC) in Linux and other systems. A process' attribute set specifies the acting principal, similar to a process' UID and GID. A file's access control list specifies which principals may access the file, similar to permission bits, which in conjunction with a file's owner and group specify access rights for a UID and GID. However, DCAC is significantly more flexible and decentralized. A process can have many attributes in its attribute set without differentiating between users and groups, files can specify formulae for allowing processes access, and there is no central mapping between string identifiers understandable by the user and integers understandable by the system.

Note that although we use $.u.\langle user\rangle$ and $.g.\langle group\rangle$ throughout the paper to couch our discussion in terms of users and groups, DCAC does not enforce any attribute

naming scheme. Users and groups in DCAC exist only by convention.

3.1 Adding and dropping attributes

DCAC allows a process to change its access control state by modifying its attribute set. Attribute set modification allows users and applications to run different processes with different privileges. A process p can always drop an attribute from its attribute set, but to add an attribute, it must satisfy one of the following rules:

- p has the parent of the requested attribute (e.g. $.u.alice$ is the parent of $.u.alice.photo$).
- p has permission to use an *attribute gateway* (discussed later in this section).

A process can add attributes to its attribute set in one of two modes, *read* or *modify*, depending on its attribute set and the configuration of attribute gateways. Read mode enables a process to use an attribute. Modify mode adds control over granting the attribute to other processes. A process with an attribute in modify mode can always downgrade that attribute to read mode. A process cannot upgrade an attribute to modify mode without a gateway or a parent attribute in modify mode. When extending an attribute by adding a new component, the extended attribute has the same mode as the parent.

Allowing a process to create and add attributes based on hierarchy permits regular users to create new principals without requiring administrator privileges. To enable flexible, application-defined resource sharing, processes use attribute gateways to acquire attributes based on user-defined rules.

Decentralized policy via attribute gateways. A primary design goal for DCAC is decentralization. Instead of centralized credentials (e.g., `/etc/group`) or centralized access policy (e.g., SELinux's `policy.conf`), DCAC distributes credentials and access policy using *attribute gateways*, which are a new type of file (in our Linux implementation, they are empty regular files with specific extended attributes). An attribute gateway allows a process to add new attributes to its attribute set based on its current attribute set. An attribute gateway is a rights amplification mechanism, like a `setuid` binary, but more flexible.

An attribute gateway has only two access modes, read and modify (execute and write are not used). If a process' attributes fulfill an access mode for a gateway, then the process may add the attribute to its set in that mode. For example, Alice might tell her colleagues that she is starting a group with attribute $.u.alice.g.atc$ for documents related to a submission to USENIX ATC. She sends email to the members of the group explaining that there is a gateway for the group in a file called `~alice/groups/atc.gate`. Her collaborators modify their login scripts to open that attribute gateway as part of their login process. The gateway's read access mode consists of a disjunction listing the user attributes for the group's members (e.g., $read = (.u.alice) \vee (.u.bob) \ldots$).

Gateways decentralize credentials and access policy. Multiple gateways (or no gateways) may exist for any attribute, with the location of the gateways and their access controlled by users and convention. DCAC does not force use of a central repository of credentials or access policy, though users may choose to create and use centralized repositories.

Having an attribute in modify mode allows a process to decide which other principals can obtain the attribute, just as having modify access to a file allows a process to control which principals can access the file. Specifically, having an attribute in modify mode allows a process to change the access control list for any gateway for the attribute, or to create new gateways for the attribute.

Access policies should incur performance penalties only for features they actually use [26]. With a hierarchical attribute namespace and decentralized attribute gateways, different users and applications can perform administration in their own domains without contention; in contrast, any administrative operations on UIDs and `/etc/passwd` (for example) require central coordination across the system.

Gateway management. Gateway management is up to users and their applications. Since DCAC is a generic kernel-level mechanism, it does not impose requirements for gateway locations; however, specific system deployments or applications can follow conventions such as a central repository of gateways. (Similarly, the Linux kernel does not enforce the use of the `/etc/passwd` file, making it general enough to support Android's application-based access control.) While users can harm themselves with incorrectly set gateway permissions, gateways do not add problems beyond those of current systems as there are already opportunities for self-harm with standard file permissions. For example, a user can make his or her `ssh` private key file world readable.

DCAC allows gateways to be in any location, which could result in a less regulated environment than in centralized systems. With current file permissions it is relatively easy (modulo perhaps setuid binaries) to determine exactly which users and groups may (transitively) access a file. Attribute gateways require an exhaustive search of the filesystem to find all attributes that might allow a given process access to a file. On the other hand, because it is easier to run processes with reduced privilege, a user could restrict her programs to only create gateways in specific, relevant directories.

Summary. We believe DCAC achieves a new balance of expressivity and simplicity due to the features:

- The attribute abstraction is generic. An attribute can represent a user, a group, a capability, an application

or a category of files, depending on the user or the application's need.

- Attributes are hierarchically named, making privilege delegation possible by extending existing attributes.
- Decentralized attribute gateways allow processes to acquire attributes that they would otherwise not be able to acquire strictly from attribute hierarchy. Creation of and policy for gateways is controlled by users and applications.
- Attributes are self-explaining strings. There is no need to map attributes to other OS-level identifiers like UIDs. Identical strings from different machines refer to the same attribute, making attributes directly sharable. This enables DCAC to support NFS (§6.3) with minimal development effort.
- There is no rigid distinction between "trusted" and "untrusted" processes. Instead, process access boundaries can be flexibly defined and flexibly delegated.

3.2 DCAC supports modern access control idioms

Here, we describe how DCAC supports the modern access control idioms described in §2.

Privilege separation/Sandboxing. Suppose Alice wishes to run her photo manager in a separate, restricted environment. Alice invokes her photo manager with a simple wrapper that does the following:

1. Adds a $.u.alice.photo$ attribute (allowed because Alice's process runs with the $.u.alice$ attribute).
2. Drops the $.u.alice$ attribute.
3. Executes her photo application.

She may similarly run her PDF reader in a separate, restricted environment by following the same steps with a $.u.alice.pdf$ attribute. Alice then sets up ACLs to allow processes running with $.u.alice.photo$ to access her photo manager's files, and $.u.alice.pdf$ to access her pdf reader's files. Each application may now access only its own set of files. In §6.2, we show how DCAC helps to sandbox an application (Evince) with vulnerabilities.

A DCAC-aware application may also enable finer privilege separation between different components. Suppose Alice's photo application wishes to isolate its file decoding routines. It may run those routines in a separate process which carries the $.u.alice.photo.reader$ attribute, and drops the $.u.alice.photo$ attribute. The program can grant $.u.alice.photo.reader$ read-only access to the photo files but nothing else, to prevent an exploit from reading other files or writing any file.

Ad hoc sharing. Hierarchical attributes combined with policies for adding attributes to a process' attribute set allow regular users to customize how they share files. Suppose that Alice (whose processes carry the $.u.alice$ attribute) wishes to share a file or set of files with a group of users including Bob (see Figure 2). Rather than updating each file every time she wants to share with a new user, like Bob, Alice instead does the following:

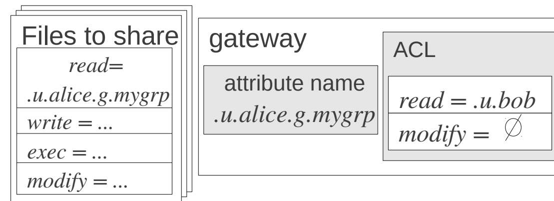

Figure 2: Ad hoc sharing with DCAC. Alice shares files by allowing $.u.alice.g.mygrp$ read access. She then creates an attribute gateway allowing $.u.bob$ to add the attribute.

1. She creates a new attribute $.u.alice.g.mygrp$.
2. Alice creates a new attribute gateway for $.u.alice.g.$-$mygrp$ (e.g., `~alice/groups/mygrp.gate`) and sets its attribute formula for the read access mode to $(.u.bob)$. Bob must learn the location of this file.
3. Processes running as Bob can use Alice's new group by locating the attribute gateway, and using it to add the attribute $.u.alice.g.mygrp$ to their attribute set. These processes will be allowed the relevant access to any file with an ACL that matches $.u.alice.g.mygrp$.

Alice can change membership of her group via ACLs on the attribute gateway. Note that the ad hoc group is a set of attributes; besides OS users, they may also represent any application-defined principals. Ad hoc sharing is used in our modified DokuWiki server in §6.1.

Managing users. Consider a server application that runs processes for multiple users, and stores user data in files. The server can use DCAC to implement access control by assigning different attributes to different users' processes, allowing application-level security requirements to be easily expressed as string attributes. For example, Linux user Alice runs `myserver`. A user on the server, `webuser`, is assigned an attribute set that has only $.u.alice.myserver.u.webuser$. A server process serving `webuser`'s requests can only access files which allow access to $.u.alice.myserver.u.webuser$. If the processes need to share some common files, the server can add a common attribute $.u.alice.myserver.common$ to their attribute sets, and add the attribute to the files' ACLs. Our modified DokuWiki server (§6.1) manages user and group permissions this way.

If a server application executes on multiple machines, it only needs to synchronize on relevant policies under its management, instead of updating them for the whole system. Server applications do not need to synchronize on a map between application-level users and OS-level access control state.

We apply DCAC to NFS (see §6.3). While different machines still share a set of OS users and groups that are already under centralized management, application-defined principals and policies do not require centralization.

Sub-principals delegating to sub-principals. DCAC allows flexible delegation to sub-principles, addressing one of the most vexing problems created by many application-specific user management systems (such as `sshd` and `apache`).

Consider the following: Professor X and Professor Y wish to collaborate. Y sends a credential (like a public key) to X and X uses it to add Y as a sub-principal. Y can now access resources shared by X, such as a subversion repository holding a joint publication. However, Professor Y recruits graduate student Z to actually do the work. Most sub-principal systems are not flexible enough to allow Y to delegate to Z without giving Z his credential. Therefore, Y must talk to X and make him aware of Z. X's list is a centrally administered bottleneck.

DCAC allows principals in a service to define and manage their sub-principals, without registering them with the service in a centralized way. Using DCAC, Professor Y can provide Z a login without involving Professor X and without revealing his credential to Z.

We let a local user, X, have his own program, `sub-auth`, to authenticate his sub-users. The program is located at a known per-user location in the system, such as `~X/sub-users/sub-auth`. X's `sub-auth` program can define where the credentials of his sub-users are stored, and how to authenticate sub-sub-users. For example, it may use a file to store sub-users' credentials, and delegate authentication to the sub-user. Thus, Y, a sub-user of X, would have his own `sub-auth` program to authenticate Y's sub-users. §6.4 shows a detailed example of delegation for sub-users using `sshd`.

DCAC does not require `sub-auth` programs, nor does it require particular naming conventions for them. A user may authenticate sub-users with a chain of certificates provided during login. A sub-user would require a certificate chain of length one: a local user vouching for the sub-user. A sub-sub-user must provide a chain of length two, and so on. Such a scheme does not require local storage for credentials and each user must only vouch for (and know about) their direct sub-users.

4. Harmonious coexistence of DCAC and DAC

DCAC is designed to work harmoniously with Linux's existing discretionary access control system (DAC), but it has mandatory access control (MAC) mechanisms to express policies more nuanced than what can be expressed by DAC alone. DCAC can augment a process' rights (e.g., allow Alice's calendar process to read a file Bob's process wrote and shared with Alice), but it can also restrict rights (e.g., limit Alice's photo reader from reading her email).

Augmenting Linux DAC. A DCAC system is permissive by default, allowing access if DAC or DCAC checks succeed. By default, files have empty ACLs, so access checks reduce to DAC checks. As a result, all valid Linux disk images are valid DCAC disk images. By preserving DAC permissions, DCAC can be deployed incrementally.

Allowing access if DAC or DCAC checks succeed makes sharing easy. For example, Alice may share a file with Bob by simply adding $.u.bob$ to the file's ACL. If instead we required DAC *and* DCAC access checks to succeed, Alice would have to adjust permissions on many of her files to start using DCAC (e.g., she would have to make her `authorized_keys` readable by sshd).

Restricting Linux DAC. Using DCAC for restricting Linux DAC permissions requires two additional pieces of state, a **pmask** (permissions mask), and a **UID-bit**. Each process has a `pmask` and a `UID-bit` that are inherited by child processes after a `fork()`, and are maintained across `exec()`. A process can only change its `pmask` by making it more restrictive (i.e., by clearing bits), and it can only clear the `UID-bit`.

The permissions mask is intended to prevent a process from reading or writing resources that Linux's DAC permissions allow (e.g., because the UID of a process and file owner match, and the file owner has read access). The permissions mask is ANDed with standard DAC permissions bits before each permissions check. So a DCAC system will check (`perms` & `pmask`) instead of `perms`. Therefore, `pmask` = 0777 does not restrict Linux DAC, while `pmask` = 0755 restricts write access to other users' files, and `pmask` = 0 causes all DAC permission checks to fail.

Each process has a `UID-bit` which is intended to limit the ambient authority granted to a process when its UID matches the UID of a resource. For example, currently in Linux, a process may change the permissions on a file or directory with matching UID even if it has no permissions on the file or its containing directory. If the `UID-bit` is clear, the process is restricted in the following ways:

1. It can only change the DAC permissions and DCAC ACLs of files with matching UID if it satisfies the modify access mode.
2. `kill` and `ptrace` are restricted to child processes.
3. It cannot remove an IPC object, change permissions on it, or lock/unlock a shared memory segment, unless the DCAC check for the object's modify access mode succeeds.

A process with the `UID-bit` set may modify DAC permissions and ACLs for files and directories with matching UIDs. An unrestricted process (e.g., running as root) with the `UID-bit` set can change DAC permissions and DCAC ACLs on any file or directory. It can also can send signals to any process.

5. Prototype implementation

We implement a DCAC prototype by modifying Linux 3.5.4.

5.1 Programming interface

In DCAC, attributes are managed through file descriptors. When a process successfully adds an attribute, it receives a file descriptor for that attribute. The corresponding kernel file object is a wrapper for the attribute. We use system calls `open`, `openat` and `close` to add and drop attributes.

- `int openat(int fd, char *suffix, int flags)`:
  If `fd` represents an attribute *attr* in the attribute set, this call adds $attr.\langle$**`suffix`**$\rangle$ and returns the associated file descriptor. `flags` can be `O_RDONLY`, or `O_RDMOD`, representing that the requested access mode is read or modify (which subsumes read). If the parent attribute has only read mode, requests for modify access mode on the new attribute will be denied.
- `int open(char *pathname, flags)`:
  If the file at `pathname` is an attribute gateway, DCAC will evaluate the access modes according to `flags` (`O_RDONLY` or `O_RDMOD`). On success, the attribute is added and the corresponding file descriptor is returned. Note that this operation does not open the actual gateway file.
- `int close(int fd)`:
  If `fd` represents an attribute, that attribute is dropped.

A potential issue with using file descriptors is compatibility with applications which are not aware of DCAC. Sometimes applications close all open file descriptors as a clean-up step (since file descriptors persist on fork and exec), which results in unintended dropping of attributes. To address this problem, we add a `lock` flag to each process' DCAC state. When the flag is set, the process' DCAC state cannot be changed, until the flag is cleared. One can set this flag in a wrapper and then invoke the application (see §6.2). The lock flag is intended solely to ease backward compatibility.

A related complication of using file descriptors to represent attributes is that programs may set the `close_on_exec` flag on their open file descriptors. For instance, if `bash` is started as a login shell (where it needs to load the user's custom settings), it sets the `close_on_exec` flag on all file descriptors except the standard I/O streams. This can causes attributes to be dropped unintentionally. DCAC ignores `close_on_exec` for attributes.

Besides `open`, `openat`, and `close`, all other operations are encoded into 4 new system calls. Table 1 shows the DCAC API.

We additionally wrote a 274-line[2] SWIG[3] wrapper to make DCAC functionality available in PHP.

5.2 Processes and Objects

The core functionality of DCAC is implemented as a Linux security module (LSM [28]).

Processes. LSMs use a `security` field in the Linux per kernel thread `task_struct` structure to store the security context of a process. The DCAC state for a kernel thread includes the attribute set, default ACL, `pmask` and the `UID-bit`, all of which are stored in a structure pointed to by the `security` field (see §4).

In a multi-threaded application, each thread can have its own DCAC state. Threads can thus run on behalf of different principals, and access control decisions are based on their individual DCAC state, which would be useful for a trusted, uncompromised server application. However, running untrusted code in a thread can lead to loss of isolation between principals in case of a compromise as threads share the same address space.

Files. For persistent file systems, DCAC requires support for extended attributes. File permission (read, write, execute, and modify) ACLs are stored in files' extended attributes, with the entire ACL encoded in a single extended attribute. For attribute gateways, the attribute controlled by the gateway and its ACLs (read and modify) are encoded in a single extended attribute.

Permission checks happen only when files are opened and not on subsequent reads/writes. The permission check occurs in the `inode_permission` LSM hook. The `inode_permission` hook is a *restrictive* hook, which means it cannot grant access if a request is already denied by the Linux DAC. However, DCAC allows access if either DAC or DCAC is satisfied (§4). Therefore, in addition to LSM, we modify 4 lines of code in `fs/namei.c` to achieve DCAC's semantics.

ACL cache. DCAC keeps a generic, in-memory ACL cache for each file in the VFS layer. There are two motivations for such a cache. First, it reduces performance overhead for remote file systems (e.g. NFS). Second, it makes DCAC usable for non-persistent file systems (e.g. sysfs). The in-memory `inode` structure contains an `i_security` field, where DCAC stores the ACL cache. The cache is initialized (from the file's extended attributes) when the ACL is first needed: when a DAC permission check fails. The cache also records each file's change time (ctime) when the ACL is fetched. The cache provides a mechanism for file systems to invalidate ACL entries to enforce filesystem coherence semantics. The ctime value in each cache entry can be used to deter-

[2] All line counts: `http://www.dwheeler.com/sloccount/`

[3] `http://www.swig.org`

Sys call	API	Functionality
dcac_add	`int dcac_add_any_attr (const char *attr, int flags)`	Add the the attribute `attr`, for root user only.
dcac_acl	`int dcac_set_def_acl (const char *dnf, int mode)`	Set an access mode, specified by `mode`, in the process' default ACL to `dnf`.
	`int dcac_set_file_acl(const char *file, const char *dnf, int mode)`	Set one access mode in the file's ACL to `dnf`.
	`int dcac_set_attr_acl(int afd, int ffd, const char *read, const char *mod)`	Create/change a gateway. `afd` and `ffd` are file descriptors of the attribute and the gateway file.
dcac_info	`int dcac_get_attr_fd(const char *attr)`	Get the file descriptor of the attribute `attr`.
	`int dcac_get_attr_name (int fd, char *buf, int bufsize)`	Get the string representation of the attribute associated with `fd`.
	`int dcac_get_attr_list (int *buf, int bufsize)`	Store the file descriptors of all the attributes of the process to `buf`.
dcac_mask	`int dcac_set_pmask(short mask)`	Set `pmask` to (`pmask` & `mask`).
	`void dcac_clear_uid_enable(void)`	Clear the `UID-bit`.
	`void dcac_lock(void)`	Lock the process' DCAC states.
	`void dcac_unlock(void)`	Unlock the process' DCAC states.

Table 1: DCAC API.

mine whether the entry needs to be invalidated, because changing the extended attribute causes a ctime change.

IPC objects. The Linux kernel's IPC object data structures share a common credential structure, `kern_ipc_perm` [28], where the DCAC ACL is stored. DCAC checks permissions to access these objects in LSM multiple hooks. We changed 6 lines in `ipc/utils.c` to allow access if DAC or DCAC allows it.

5.3 Attribute management

Using the rules described in §3.1, a process can only add new attributes based on existing attributes in its attribute set. To initialize attribute state, our Linux DCAC implementation allows processes running as root to add any attribute to their attribute sets with arbitrary modes (read or modify). We then modify system binaries, such as `login`, to initialize the attribute state for user processes. `login` is already responsible for initializing a process' UID and GID state by reading the `/etc/passwd` and `/etc/group` files and invoking system calls such as `setuid` and `setgroups`. We extend this responsibility to include attributes.

Our examples use $.u.alice$ to represent an attribute corresponding to a specific Linux user. We encode this convention in our prototype by changing `login` to add the $.u.\langle username \rangle$ attribute, with both read and modify access modes, to user login shells. Similarly, $.g.\langle grpname \rangle$ attributes represent Linux groups, and they are added with only read mode, since only root has administrative control of them. Modifying `login` required a 28 line change to the shadow 4.1.5.1 package.

We also modify `sshd` and the LightDM desktop manager[4] to set up the attribute state when the user logs in remotely or via a graphical user interface. We changed 37 lines of code in OpenSSH 5.9 and 20 lines of code in LightDM 1.2.1.

6. DCAC application implementation

We demonstrate several use cases of DCAC in real applications.

6.1 Application-defined permissions in DokuWiki

DokuWiki[5] is a wiki written in PHP that stores individual pages as separate files in the filesystem. As a result, OS file-level permissions suffice for wiki access control. In fact, access control only requires setting attributes and default file ACLs on login. Then the OS ensures that all ACL checks occur properly, without need for application-level logic.

We add a 246-line DCAC module to DokuWiki's collection of authentication modules. DokuWiki with DCAC executes in a webserver initialized with the $.apps.dokuwiki$ attribute. Upon user login, the webserver process acquires a user-specific attribute $.apps.dokuwiki.u.\langle username \rangle$ and drops $.apps.dokuwiki$. Default ACLs on all created files are set to the user-specific attribute. The server permits anonymous page creation and access through a common attribute $.apps.dokuwiki.common$. DokuWiki runs as a CGI script to ensure a new process with the $.apps.dokuwiki$ attribute handles each request, restricting the impact of a compromise during a request to the logged in user. Otherwise, a reused compromised process could affect other users when it acquires their attributes during a new request.

DokuWiki has a built-in ACL system that is only controllable by superusers. We modify DokuWiki to support user-created groups. To do this, we create a

[4] `http://wiki.gentoo.org/wiki/LightDM`

[5] `http://www.dokuwiki.org`

new directory to hold gateway files with a directory per wiki user. When a user creates a group, she also creates a gateway file to attribute $.apps.dokuwiki\text{-}.u.\langle username \rangle.g.\langle groupname \rangle$ with the name of the group in a per-user location defined by a DokuWiki naming convention. The gateway file has read permission for the members of the group. Each user's group directory is traversable (has execute permission) by all users. When a user is informed (out of band) that she has been added to a new group, she records the group name and gateway path name in her groups file. We have a user modify her own groups file to ensure that her groups are not disclosed by arbitrary access to a common file. During login, the DCAC authentication module reads the user's groups file and attempts to access gateways and add group attributes.

We add calls to DokuWiki's XML RPC interface to use this group functionality: `setPerms(pageName, perms)` allows the DCAC permissions of wiki files to be adjusted. `setGroup(groupName, members)` modifies the membership of the group named `groupName`. Finally, `getMyGroups` and `setMyGroups` allow a user to activate the additional privileges given to her by groups by modifying her groups file.

6.2 Sandboxing Evince

We implement an application wrapper that sets up DCAC state for an application such as the default ACL, `UID-bit`, `pmask` and attribute set. The wrapper can perform user-specified work (via scripts) before application execution (e.g., granting a sandboxed application permissions to a specific file) and after application termination. With a wrapper, DCAC can sandbox unmodified applications because DCAC state is inherited across `fork` and `exec`.

We port a simple stack-based buffer overflow targeting an old version of Evince (evince-0.6.1) to test the sandboxing ability of DCAC. Since document viewers generally do not need write permission, we can sandbox Evince by clearing its `UID-bit` and setting its `pmask` = 0115, by using the application wrapper. We allow execute permission for directory traversal, and allow read for world-readable files so Evince can load its shared libraries. Additionally, to allow evince to open the target `pdf` file, the wrapper keeps $.u.alice.evince$ in the attribute set, and adds it to the `pdf` file's ACL. The wrapper can reset the file's ACL after Evince terminates. Upon triggering the exploit, the attacker only has access to world readable files, and even a shell opened via an exploit remains confined.

6.3 NFS

In a normal NFS environment, machines are within the same trust domain, and share a common set of OS users and groups, which usually requires centralized management. By adopting an attribute naming convention for users and groups (such as $.u.\langle username \rangle$ and $.g.\langle grpname \rangle$), DCAC eliminates centralized management. A user can define her own sub-principals and manage them in her own way, on all machines.

The Linux NFS implementation does not support extended attributes, but we ported a patch for NFSv3 [18] to add extended attribute support. In addition to the patch, we also modified 326 lines of code for NFS in the Linux kernel source.

The NFSv3 specification [24] does not define how a server should check permissions. In the NFS implementation in Linux, a client OS checks permissions when a file is opened, but the server does not check permission for subsequent `read` calls, or for `write` calls on the files that belong to the process' user. It only checks permission for `write` calls on files that belong to a different user. We remove this extra check in DCAC. We believe that this change is sensible, since under this change NFS files still obey the standard UNIX convention where permissions are checked only on open.

Clients must make DCAC access control decisions, as they have access to a process' attributes as well as the resource's (e.g., file's) attributes. However, creating or removing files and directories requires write access to the parent directory and in Linux NFS the client simply forwards these operations to the server without checking permissions. In DCAC, if a client uses attributes to determine that a `create` or `remove` operation is legal, it appends a hash of the cached ACL for the parent directory in its RPC to the server. The server checks the hash against the ACL, and if they match, it knows the client has an up-to-date copy and can trust the client's decision to allow the operation. While NFS servers trust their clients, this check is to ensure that clients do not make wrong decisions based on stale permissions. In addition, DCAC appends the process' default ACL to `create` and `mkdir` calls, to initialize ACLs on newly created files and directories.

For regular files, DCAC also uses the ACL cache to determine permission when they are opened. To guarantee *close-to-open* consistency [24], the cache is invalidated when a ctime change is observed.

6.4 Managing sub-users in SSHD

We modify 81 lines of `sshd` to support the access control model described in Section 3.2.

Modern versions of `sshd` support a *forced command* option, which allows unprivileged users to authenticate sub-principals with public keys via the `svnserve` program. Arguments to `svnserve` control details like the user name for sub-principals (because sub-principals do not have user names in `/etc/passwd`). However, `svnserve` does not allow the kind of flexible delegation described in the next example.

Authentication example. When `sshd` receives a login request for X.Y, it invokes X's `sub-auth` program with only X's privilege, and passes to it the sub-user name "Y" and the credentials the request provides. If the `sub-auth` program returns successfully, `sshd` approves this request and restricts X.Y's privilege by properly setting the attribute set, `pmask` and `UID-bit`.

For a more concrete example, consider a hierarchy of users: X.Y.Z, described here and illustrated in Figure 3.

1. When X.Y.Z tries to login using `ssh`, he provides username "X.Y.Z" and some credential.
2. `sshd` invokes X's `sub-auth` program, passing sub-username "Y.Z" and the credential to it, with UID set to X's and only $.u.X$ in the attribute set.
   (a) X's `sub-auth` finds that it is one-level down, it keeps only $.u.X.Y$ in the attribute set, and restricts `pmask` and `UID-bit`.
   (b) It `exec`'s Y's `sub-auth`, passing sub-username "Z" and the credential to it. Y's `sub-auth` verifies the credential, and returns successfully.
3. Now `sshd` knows the request is authenticated. Before `exec`'ing the shell, `sshd` keeps only $.u.X.Y.Z$ in the attribute set, and restricts `pmask` and `UID-bit`.

Note that another OS user, A, can have a completely different `sub-auth` program. His `sub-auth` program may be based on certificate chains, and does not need further lower-layer `sub-auth` programs. For example, A can sign a certificate for B's public key as A.B, and B can sign another certificate for C's public key as A.B.C. When A.B.C logs in, he needs to provide the two certificates to be verified by A's `sub-auth`, as well as a proof that he has C's private key, such as a signature.

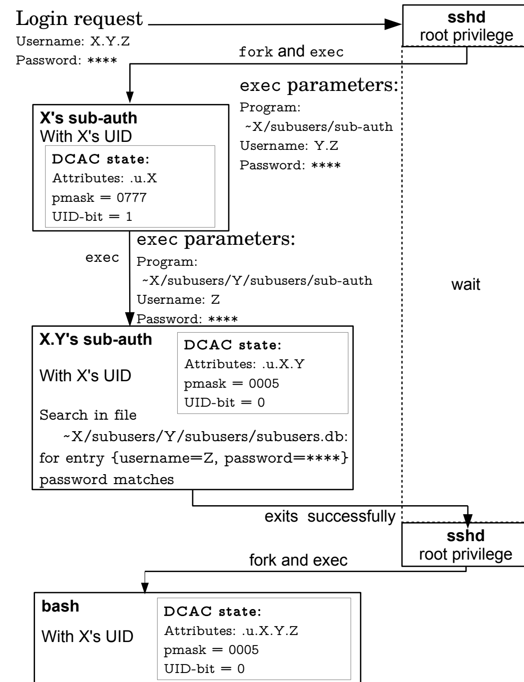

Figure 3: Authentication of sub-users in our modified `sshd`: support for arbitrary nesting of sub-principals.

7. Evaluation

We measure the performance overhead of DCAC through both targeted benchmark programs and representative applications. Our benchmarking systems had quad-core Intel Core2 2.66 GHz CPUs, 8 GB of RAM, and a 160 GB, 7200 RPM disk. All servers and clients were connected by gigabit Ethernet.

7.1 Microbenchmarks

Filesystem. We run the Reimplemented Andrew Benchmark (RAB) [19], a version of the Andrew benchmark scaled for modern systems, on both a local ext4 filesystem and NFSv3.

RAB initially creates 100 files of 1 KB each and measures the time for the following operations: (1) creation of a number of directories, (2) copying each of the 100 initial files to some of these directories, (3) executing the `du` command to calculate disk usage of the files and directories, and (4) using `grep` to search all file copies for a short string and checksumming all the files. The exact number of operations varied depending on category (ext4 or NFS) and is described with the corresponding figures. Results are shown in Figure 4.

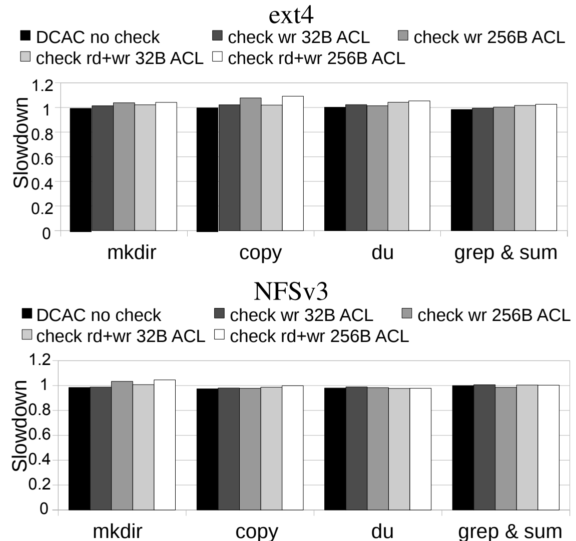

Figure 4: RAB results on local ext4 and NFSv3. 20,000 directories are created in the mkdir phase, and 100 files of 1 KB each are copied to 500 directories in the copy phase. The slowdown is relative to unmodified Linux.

We compare the results for the following cases:

- The baseline, which uses an unmodified Linux kernel.

FS	Time (μs)		
	chmod	Changing DCAC file ACL	
		32B ACL	256B ACL
ext4	1.24	2.19	4.27
NFSv3	224	228	238

Table 2: Time to change the ACL of a file, compared to chmod.

- DCAC kernel, where the default ACL is empty, and ACLs on files are not checked. DCAC adds at most 1% overhead on local ext4 and NFS.
- DCAC kernel, where the default ACL only contains the write access mode; DCAC ACL checks occur for every write access. On local ext4, the overhead is 0% to 4% for a 32B ACL, and 0% to 8% for a 256B ACL. On NFS, the overhead is below 4%.
- DCAC kernel, like the prior case, but DCAC ACL checks occur for both read and write accesses. On local ext4, the overhead is 1% to 4% for a 32B ACL, and 2% to 9% for a 256B ACL. On NFS, the overhead is below 5%.

On ext4, the kernel stores extended attributes within inodes if they are below a certain size threshold. With 256-byte inodes, 32 bytes is below this threshold; 256 bytes is not, so extra disk blocks must be allocated, resulting in larger overhead in the mkdir and copy phases, where new files are created. For the du phase and the grep and sum phase, observed differences correspond to whether read ACLs are being checked and differ little for ACL sizes. This is likely because DCAC reads ACLs from ACL caches for most of the time.

On NFS, the number of round trips per operation dominates the performance; overhead for disk storage for extended attributes on the server is negligible in comparison. Most of the time, DCAC reads ACLs from the ACL caches, which does not incur network communication. On the creation of a file or directory, DCAC appends the initial ACL to the `create` or `mkdir` RPC, instead of using a separate `setxattr` RPC. As a result, DCAC adds very small overhead on NFS.

ACL manipulation. We compare the time it takes to change the ACL of a file, to `chmod` in Linux. Table 2 shows that, on a local ext4, changing ACLs can be 1.7× to 3.4× slower than `chmod`, depending on the size of the ACL; on NFSv3, ACL size has a small impact on performance, and the time spent for changing ACLs is comparable to `chmod` (under 6.5%).

IPC. DAC and DCAC check permission for shared memory segments and named pipes only when they are attached to the process' address space or opened; however, every up or down operation on a System V semaphore requires a permission check. We measure the overhead induced on semaphore operations by DCAC ACL checks. The baseline comes from the DCAC kernel where DAC permission checks always succeed, so no ACL checks ever occur. The DCAC measurement comes from removing a process's DAC permissions by setting its `pmask` to 0 and giving it an attribute. The semaphore is accessible to processes with this attribute. Thus, we ensured that a DCAC ACL check is performed on every semaphore operation. We measure the average time per semaphore operation over a long sequence that alternated between up and down (measuring the average overhead of both), and set the initial semaphore value so that no semaphore opertion blocks. Table 3 shows 27% slowdown for a semaphore operation requiring a DCAC permission check.

Setup	Baseline	DCAC
Time (ns)	279	355 (1.27×)

Table 3: Overhead for a semaphore operation.

FS	Time (s)		
	baseline	check wr	check rd&wr
ext4	197.3	198.3 (1.01×)	201.2 (1.02×)
NFSv3	322.1	325.4 (1.01×)	325.7 (1.01×)

Table 4: Kernel compile time, averaged over 5 trials. "check wr" means the DCAC write access modes on output files and directories are checked; "check rd" means the DCAC read access modes on the source files are checked. "baseline" means using unmodified kernel. The size of each ACL is 256 bytes.

7.2 Macrobenchmarks

Kernel compile. We measure the time to compile the Linux kernel (version 3.5.4, without modules). Table 4 shows the overhead is negligible for both an ext4 filesystem and NFS. DCAC only performs additional permission checks on file open, creation, and deletion. The amount of time spent on these operations is small enough compared to the computation involved in a kernel compile to cause low overhead.

DokuWiki. We benchmark DokuWiki by playing back a set of modifications made to the DokuWiki website (which is itself run using DokuWiki). This is a set of 6,430 revisions of 765 pages. We made a set of requests to a wiki with a 90% read workload. Each write operation replaces a page of the wiki with the next version in the set of revisions that we have. We measured the total wall clock time for 16 clients to perform 100 requests apiece against the wiki. The baseline is the wiki running on the same machine with the same kernel but no attributes applied to any files and using standard authentication. Results are in Table 5. DCAC authentication and plain authentication results were within margin of error of each other. This is expected, as DCAC merely adds a few system calls to operations that otherwise have a lot of computation and file I/O through running PHP scripts.

Setup	Baseline	DCAC
Time (s)	45.5 ± 0.7	45.3 ± 0.7

Table 5: Wall clock times for 16 clients to complete 100 requests apiece to DokuWiki. Standard deviations are determined from 10 trials.

Systems	Relation to DCAC
Sandboxes	Focus on isolation, DCAC also accommodates fine-grained sharing
Flexible policy specification	Focus on completeness rather than usability, DCAC strives for balance
Application- and user- defined access control	DCAC provides fine-grained control, supports network filesystems
New security models	DCAC concepts easier to understand: closer to traditional users and groups

Table 6: Comparing DCAC with related systems.

8. Related work

DCAC is most directly inspired by two systems, Capsicum [27] and UserFS [14]. Capsicum shows that programmers want and will use system abstractions that make writing secure code easier. Capsicum implements a fairly standard capability model for security; its innovation is in casting file descriptors, an abstraction familiar to Unix programmers, as a capability. DCAC applies this insight by representing attributes as file descriptors. Capsicum's capability mode is similar to DCAC's `pmask` and DCAC's `UID-bit`, in that they are both used to deprivilege a process and restrain its ambient authority granted by legacy access control systems; however, DCAC's `pmask` and `UID-bit` are more fine-grained – they can selectively restrict a process' ability to perform different operations on different files.

UserFS [14] leverages existing OS protection mechanisms to increase application security by explicitly maintaining a hierarchy of UIDs to represent principals. Unfortunately, system-wide UIDs are awkward for dynamic principals. For example, independent server applications would contend for UIDs even though their principals are in logically separated domains. Moreover, in a distributed setting, groups of machines would need to synchronize on what UIDs are currently in use. DCAC is designed to work well where UserFS struggles—highly dynamic, distributed deployments within a single administrative domain.

There are too many access control systems and proposals to analyze them all, so we describe the novel combination of features in DCAC by contrasting with entire classes of access control systems, with modern exemplars. Table 6 summarizes our analysis.

Sandboxing. Many projects try to isolate ("sandbox") potentially malicious code from the rest of the system. Android repurposes UIDs to isolate mutually distrusting applications from one another. The Mac OS X Seatbelt sandbox system can constrain processes according to user-defined policies, which is used by Chromium [8].

User-level sandboxes are often specific to an application, and are hard to get right because applications regularly change the files and directories they access. As a result they suffer problems with usability and security vulnerabilities [2, 3].

DCAC provides a single mechanism for all applications, usable by ordinary (non-administrator) users, that can meet the varying data access requirements of applications. DCAC also meets access control requirements that go beyond sandboxing, like user-controlled, fine-grained sharing. A key contribution of DCAC is that it combines the models used by users (file access control and sharing) and administrators (creating sandboxes).

Capsicum has a daemon, Casper, which provides services to sandboxed processes; in DarpaBrowser [25], confined code can access resources in the system via Powerboxes, which is controlled by user interface interactions. These techniques are also applicable to DCAC, providing confined processes an alternative path to reach privileged resources without escalation.

Flexible policy specification. SELinux [16] and AppArmor [7] aim to provide comprehensive policies for the resources that applications can access. This comprehensiveness can lead to usability problems: SELinux is notoriously difficult to use [22]. Both of them are only manageable by administrators and have difficulty accommodating situations where policies for one application vary per user. DCAC is configurable on a per-user basis.

eXtensible Access Control Markup Language (XACML) [5] is an XML-based format for defining access control. While flexible, it relies on XML manipulations (e.g., XPath queries) that are unsuited for use in frequent latency-sensitive operations within an OS.

POSIX ACLs [4] allow for users to define expressive access control lists for permissions. However, these have important limitations, like the inability to support user-defined groups without explicitly putting all group members in the ACL of every file that has group permission (which in turn makes group membership updates difficult). DCAC can accommodate user-defined groups.

The Andrew File System [12] (AFS) supports flexible, per-directory ACLs, and allows users to create groups under their own administration. In comparison, DCAC is not restricted to a specific file system or IPC mechanism, and supports more general usage due to its attribute-based model.

Application- and user-defined access control. Several prior systems allow program-controlled subdivision of users into further users for finer-grained protection [10, 14, 23]. These systems still label processes by one user, and as a result are less flexible than DCAC. Ad-

ditionally, these systems do not store information about the user hierarchy in a way that easily allows shared use in a network filesystem: UserFS [14] requires synchronization of unrelated applications on a global database of all users. By contrast, DCAC attributes are strings that self-describe where they belong in the attribute hierarchy.

DCAC is inspired by attribute-based access control, proposed as part of InkTag [11]. DCAC generalizes the approach to a trusted OS and makes it coexist with existing access control.

New security models. Decentralized Information Flow Control (DIFC) [9, 15, 17, 21, 29] systems modify access privileges based on the information that applications have accessed. DIFC-enforcing systems may provide stronger security guarantees than DCAC.

Systems that use radically different security models require that developers adapt the logic of their code to work in these models. While DCAC may require code changes to applications, we expect they will be less significant because DCAC's core concept, the attribute, is implemented as a file descriptor, and is easily mapped onto users and groups, concepts that are familiar to developers and likely reflected in their code. Enforcing new security models can require extensive global OS modification, whereas DCAC's changes fit mostly within the existing LSM framework.

9. Conclusion

OS-level support for application-defined principals makes DCAC usable and flexible enough to solve modern access control problems. DCAC decentralizes privilege and policy specification, improves application security, and supports distributed operation.

References

[1] GNU Zlib : List of security vulnerabilities. http://www.cvedetails.com/vulnerability-list/vendor_id-72/product_id-1820/GNU-Zlib.html.

[2] National Vulnerability Database: CVE-2012-4681. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4681.

[3] National Vulnerability Database: CVE-2013-0422. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0422.

[4] POSIX 1003.1e Draft (Security APIs). http://users.suse.com/~agruen/acl/posix/Posix_1003.1e-990310.pdf.

[5] eXtensible Access Control Markup Language (XACML) Version 3.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs02-en.html, August 2012. OASIS Committee Specification 02.

[6] BARTH, A., JACKSON, C., REIS, C., AND GOOGLE CHROME TEAM. The security architecture of the chromium browser. Tech. rep., Google Inc., 2008.

[7] BAUER, M. Paranoid penguin: an introduction to novell AppArmor. *Linux Journal* (2006).

[8] The Chromium Project: Design Documents: OS X Sandboxing Design. http://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design.

[9] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIERES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the Asbestos operating system. In *SOSP* (2005).

[10] FRIBERG, C., AND HELD, A. Support for discretionary role based access control in ACL-oriented operating systems. In *Proceedings of the second ACM workshop on Role-based access control* (1997).

[11] HOFMANN, O. S., DUNN, A. M., KIM, S., LEE, M. Z., AND WITCHEL, E. InkTag: Secure applications on an untrusted operating system. In *ASPLOS* (2013).

[12] HOWARD, J. H., ET AL. *An overview of the Andrew File System*. Carnegie Mellon University, Information Technology Center, 1988.

[13] KERRISK, M. Namespaces in operation, part 1: namespaces overview. https://lwn.net/Articles/531114/, Jan. 2013.

[14] KIM, T., AND ZELDOVICH, N. Making Linux protection mechanisms egalitarian with UserFS. In *USENIX Security* (2010).

[15] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. In *SOSP* (2007).

[16] LOSCOCCO, P., AND SMALLEY, S. D. Meeting critical security objectives with security-enhanced linux. In *Proceedings of the Ottawa Linux Symposium* (2001), pp. 115–134.

[17] MYERS, A. C., AND LISKOV, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM) 9*, 4 (2000), 410–442.

[18] XATTR protocol patch for NFSv3. http://namei.org/nfsv3xattr.

[19] PORTER, D. E., HOFMANN, O. S., ROSSBACH, C. J., BENN, A., AND WITCHEL, E. Operating system transactions. In *SOSP* (2009).

[20] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *USENIX Security* (2003).

[21] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: Practical fine-grained decentralized information flow control. In *PLDI* (2009).

[22] SCHREUDERS, Z. C., MCGILL, T., AND PAYNE, C. Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM. *TISSEC* (September 2011).

[23] SNOWBERGER, P., AND THAIN, D. Sub-identities: Towards operating system support for distributed system security. Tech. rep., University of Notre Dame, 2005.

[24] SUN MICROSYSTEMS, INC. RFC 1813 - NFS: Network File System Version 3 Protocol Specification. IETF Network Working Group, 1995.

[25] WAGNER, D., AND TRIBBLE, D. A Security Analysis of the Combex DarpaBrowser Architecture. *Online at: http://www. combex. com/papers/darpa-review* (2002).

[26] WATSON, R. N. A decade of os access-control extensibility. *Communications of the ACM* (2013).

[27] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: Practical Capabilities for UNIX. In *USENIX Security* (2010).

[28] WRIGHT, C., COWAN, C., MORRIS, J., SMALLEY, S., AND KROAH-HARTMAN, G. Linux security modules: General security support for the Linux kernel. In *USENIX Security* (2002).

[29] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *OSDI* (2006).