2014-09

A survey of distributed capability file systems and their application to cloud environment

Jatho, Edgar W., III

Monterey, California: Naval Postgraduate School

http://hdl.handle.net/10945/43930

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A SURVEY OF DISTRIBUTED CAPABILITY FILE SYSTEMS AND THEIR APPLICATION TO CLOUD ENVIRONMENTS

by

Edgar W. Jatho, III

September 2014

Thesis Co-Advisors:	Peter Denning
	Mark Gondree

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704–0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE 09-26-2014	3. REPORT TYPE AND DATES COVERED Master's Thesis 09-01-2013 to 09-26-2014	
4. TITLE AND SUBTITLE A SURVEY OF DISTRIBUTED CAPABILITY FILE SYSTEMS AND THEIR APPLICATION TO CLOUD ENVIRONMENTS			5. FUNDING NUMBERS
6. AUTHOR(S) Edgar W. Jatho, III			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE
13. ABSTRACT *(maximum 200 words)* This thesis considers distributed capability systems as a potential solution to securing data in cloud environments. The U.S. Navy, Intelligence Community and Department of Defense have begun a significant investment to leverage scalable, distributed cloud-based solutions for information sharing. We believe capability systems suggest a promising direction for new platforms, a bold approach drawing directly from mature ideas first explored in the 60s and 70s. We survey the properties and limits of existing distributed capability file systems, as a step toward understanding how capability-based designs might serve cloud-scale systems. We highlight some lessons learned in our observations and find that, while no existing capability-based distributed file system demonstrates all of the desirable security traits observed of smaller-scale capability systems, it should be possible to define and create one that does, using capabilities carefully designed to obey a set of known properties.			
14. SUBJECT TERMS Capabilities, Distributed Capability System Survey, Navy Tactical Cloud, Cloud Security, capability systems, computer security, Distributed File System Security			15. NUMBER OF PAGES 93 16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

A SURVEY OF DISTRIBUTED CAPABILITY FILE SYSTEMS AND THEIR APPLICATION TO CLOUD ENVIRONMENTS

Edgar W. Jatho, III
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2014

Author:	Edgar W. Jatho, III
Approved by:	Peter Denning Thesis Co-Advisor
	Mark Gondree Thesis Co-Advisor
	Peter Denning Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis considers distributed capability systems as a potential solution to securing data in cloud environments. The U.S. Navy, Intelligence Community and Department of Defense have begun a significant investment to leverage scalable, distributed cloud-based solutions for information sharing. We believe capability systems suggest a promising direction for new platforms, a bold approach drawing directly from mature ideas first explored in the 60s and 70s. We survey the properties and limits of existing distributed capability file systems, as a step toward understanding how capability-based designs might serve cloud-scale systems. We highlight some lessons learned in our observations and find that, while no existing capability-based distributed file system demonstrates all of the desirable security traits observed of smaller-scale capability systems, it should be possible to define and create one that does, using capabilities carefully designed to obey a set of known properties.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ACL	access control list
API	application programming interfaces
C2ISR	command and control, intelligence, surveillance, and reconnaissance
CAK	capabilities as keys
CHERI	Capability Hardware Enhanced RISC Instructions
DARPA	Defense Advanced Research Projects Agency
DFS	distributed file system
DOD	Department of Defense
DOI	Digital Object Identifier
EROS	Extremely Reliable Operating System
GNOSIS	Great New Operating System In the Sky
IC	intelligence community
ISA	instruction set architecture
ISR	intelligence, surveillance, and reconnaissance
IT	information technology
ITE	information technology enterprise
JIE	joint information environment
MDS	meta-data server
MMU	memory management unit
NIST	National Institute of Standards and Technology

NPS	Naval Postgraduate School
OS	operating system
OSD	object storage device
RISC	reduced instruction set computing
SFS	Simple File System
SI	storage index
Tahoe-LAFS	Tahoe-Least Authority File System
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USG	United States Government

Acknowledgments

First and foremost, I would like to thank my amazing, always supportive, selfless, Proverbs 31 Wife, without whom, without a doubt, I could not have accomplished this task. Her willingness to take on everything else and put so much on hold, enabled me to give my best to this project and complete on time. She handled the lion's share of our adoption details for our son, Caleb, and the preparation for our soon to arrive second son.

I would like to thank both of my thesis advisors for the time, care and effort they poured into guiding my work. Without Professor Gondree, no doubt, *if* this thesis existed, it would have been far less interesting, accurate, or clear. He gave countless (truly countless) hours of his time to discuss nuances in interpretation, help me fully explore the thought-space, and clarify my work. I cannot thank him enough for his assistance and truly academically curious nature that makes him a joy to have as a Professor in class and especially as a Thesis Advisor and mentor.

Likewise, without Professor Denning, I would not have had the opportunity to research such an interesting vein of computer security research. His weekly guidance, brought me down the path to explore something that truly interested me, a luxury many do not get the chance to do. He made sure I acquired a breadth and depth of research knowledge with which to work, going back decades to the highly relevant work of the pioneers in Computer Science that would not have been possible with anyone else. Working with such an extraordinary person, accomplished scientist and author, and having him pour effort into your accomplishment is a once in a lifetime honor. I am extremely grateful for the privilege of working with him.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

This thesis considers distributed capability systems as a potential solution to securing data in cloud environments. The U.S. Navy, Intelligence Community and Department of Defense have begun a significant investment to leverage scalable, distributed cloud-based solutions for information sharing. We believe capability systems suggest a promising direction for new platforms, a bold approach drawing directly from mature ideas first explored in the 60s and 70s. We survey the properties and limits of existing distributed capability file systems, as a step toward understanding how capability-based designs might serve cloud-scale systems.

1.1 Navy Cloud Computing

The U.S. Navy is currently in the process of charting a course to a common, cloud-based architecture. The chief aim of this initiative is to bring all Unclassified, Secret, Top Secret, and SCI systems into a cloud architecture [1]. The motivation is to increase performance and simultaneously to "decrease the long term costs of end to end architecture" [1]. The Department of Defense (DOD), and the Intelligence Community (IC) have each announced plans to migrate their information systems to cloud-based environments, implementing the Joint Information Environment (JIE) and the Intelligence Community Information Technology Enterprise (ITE), respectively. While recognizing its own unique operational environment will require a different approach, the Navy hopes to leverage commonalities in cloud environments for relatively seamless interoperability with its sister services in the DoD and with the IC.

A recent 2014 RAND report describes an approaching "flood of data coming from the intelligence, surveillance, and reconnaissance (ISR) systems that Navy... commanders rely on for situational awareness" [2]. It reports that "as little as 5% of the data collected by ISR platforms actually reach the Navy analysts who need to see them" [2]. In a 2011 Maritime ISR Enterprise Acquisition Review, researchers concluded early in their analysis that "ongoing Navy ISR acquisition programs will generate far more new data than the existing C2ISR afloat and ashore infrastructure and associated acquisition programs..." could realistically accommodate, quickly reaching petabyte (10^{15} bytes) scale [3]. The report goes on to observe an issue widely acknowledged throughout the Navy: in the present system-centric information environment, frequently, required data can be spread or separated across multiple classification levels, or stored in databases and networks not accessible to the war-fighters, analysts, or tactical commanders that need it [3]. "Unlike a Google search, users have to know not only what they are looking for, but also where to look for it" [3]. Often this means that data from which a warfighter could have benefited is inaccessible when it is needed most. Moving to the cloud is seen as the best solution to deal with both the Navy's emerging Big Data problem and the increasing fiscal constraints under which it must operate and in which it must innovate [2].

1.2 Benefits of the Cloud

The Navy's current Information Technology (IT) environment is very much like the state of federal government systems, described in the 2011 Federal Cloud Computing Strategy: "characterized by low asset utilization, a fragmented demand for resources, duplicative systems, environments which are difficult to manage, and long procurement lead times" [4]. The potential benefits of moving to a cloud-based IT environment are attractive: more efficient use of computing resources (i.e., storage, memory, computational power), rapid provisioning of resources, elimination of data silos, scalable and on-demand services, and disaster recovery through multisite mirroring [5]. Cloud adoption promises a significant reduction in costs to the Navy, allowing it to reduce its population of data centers from 125, its current size, to roughly 20 [6]. This alone can result in substantial savings, considering both real estate reduction, and the savings in cooling, staffing, maintenance, and additional overhead incurred running those 105 eliminated data centers. Further, it has been suggested that cloud-based defenses can be more robust, scalable, responsive, and cost-effective [7].

1.3 A Growing and Continued Threat

While the Navy adopts new technologies in response to changing demand in its information environment, U.S. adversaries are growing more adept at finding and taking advantage of vulnerabilities in our information systems. The Wall Street Journal recently reported the discovery of a large cyber attack from Iran that infiltrated U.S. Navy unclassified networks [8]. In 2013, Mandiant published a report on what it called *Advanced Persistant*

Threat 1 (APT1), revealing a large-scale Chinese military operation to infiltrate and exploit the United States government, industry and infrastructure, employing diverse tactics (e.g., 40 different malware families, spear phishing, webserver exploits) [9]. Other attacks and programs have been attributed to Russian actors, targeting everything from defense, to the energy industry, to our banking industry [10], [11]. Further, a series of high-profile attacks by insiders have broad repercussions on how we partition trust across and within our networks, and severely undermine our reliance on boundary defenses in system security [12], [13]. Our adversaries are clearly not standing still. Increasingly, they employ the cyber domain as an arena in which they can stand, strike, and meet real objectives, arguably with both plausible deniability and impunity.

1.4 Security in the Cloud

In light of these threats, researchers have identified open questions regarding data security and integrity in the cloud, including:

- Confidentiality. How does an organization know that its data is protected when at rest and when in motion? How can we know that separation is assured between organizations, users with differing classification levels, or users in general?
- Integrity: When critical data is stored in the cloud, how can one know it will be present at a later time in an uncorrupted state?
- Provenance: When one possesses a piece of data, how can I track its pedigree, its source to give a commander an understanding of its level of certainty or trustworthiness? What is the origin of a particular piece of data [14]?
- Jurisdiction: Where does data reside (at rest) or pass through (in motion) physically under any given circumstances? What laws apply to it as a result?
- Secure Sharing: Allow sharing of data among authorized parties while obeying all the access control policies set by object creators.
- Program Encapsulation: Run untrusted software in a restricted environment that resists system hijacking and quarantines errors that the software might produce.
- Access Revocation: Enable revocation of access rights at any time, regardless of the number of parties that may be sharing access.

In moving to a common, cloud-based operating environment, the Navy will face bringing multiple distinct, formerly disconnected information systems together under a new operating and sharing paradigm. NIST's Big Data Public Working Group observes that "clouds and federations tend to introduce complications for application and the technology domains, and security mechanisms are often a previous generation of enterprise solutions that are being repurposed inappropriately for a radically different threat model" [15]. Clearly, finding methods to encapsulate, securely compose and protect these systems (internally and externally) will be as much a challenge as interoperability, enhancing performance, and developing new applications and analytics.

1.5 Capabilities for Cloud Systems

In the context of advanced perstent external threats, insider threats in privileged roles, and many open questions about security in the cloud, we are strongly motivated to investigate *capability systems* in the context of distributed environments, like the cloud. Capability systems represent a fundamental shift in approach regarding access, addressing, and control.

Capabilities can provide an elegant solution to some security problems that have continued to demand our attention over the years. In the 1960s, capability addressing was developed by Dennis and Van Horn [16]. The idea was to encapsulate all software in least-privilege protection domains, and to grant access to the objects belonging to domains by means of special protected pointers called capabilities. Capability architectures were successfully implemented on several commercial systems. Unfortunately, when RISC chips arrived in the early 1980s, the emphasis shifted from security to performance, and capability principles disappeared from many chips and operating systems.[1]

Today, security is once again a top priority, and capability architectures are making a comeback. In particular, the DOD has already begun to investigate clean-slate approaches to securing new systems, under a recent DARPA program. One such approach, CHERI, employs memory capability architecture. See Section 2.2 for more on CHERI and other contemporary capability projects. Among the intrinsic security features that make capability systems

[1]Though successful capability research continued throughout this period, (i.e., KeyKOS, EROS and others).

attractive as a solution, are the following:

- Enable fine-granularity, least privilege operation.
- Provide a mechanism for encapsulation and protection of process memory.
- Provide unforgeable references, supporting strong resource protection.
- Increase system reliability due to the highly compartmentalized nature of their resource and authority management.
- Use subject-controlled authorities, removing centralized management of permissions and admitting more scalable and efficient distributed designs.
- Enable controlled sharing, providing a secure way to delegate privilege and enable flexible methods of sharing data.
- Permit programmers to develop services less open to abuse (i.e., avoiding so-called *confused deputies*), reducing the threat of program hijacking, manipulation by malware, and memory safety violations.
- Reduce insider threat via compartmentalization of memory and resources.
- Allow systems to eliminate admin, super-user or root accounts [17].

Given the inherent security properties of capability systems, our principal question is: can capabilities be applied to similar effect in the Cloud?

1.6 Capability-based Distributed File Systems

As a first step toward exploring capabilities in a distributed cloud context, we survey capabilities in the context of *distributed file systems* (DFS). Several secure DFS have been proposed using capabilities for security policy enforcement. We survey six prominent, representative systems—Tahoe-LAFS, Maat, CapaFS, DisCFS, DOI, Neo—examining their designs and resultant properties. The rationale for this survey is in the sometimes ambiguous use of the term "capability" in systems engineering. Indeed, wide belief exists that capability systems are incapable of some desirable security properties (i.e., confinement, privilege revocation). Miller, Yee and Shapiro present a categorization of these properties and survey many capability-based operating systems, showing that not all capability systems are equal [18]. They find some systems are able to support these properties, while others do not. In particular, they describe one class of capability system (*object capability* systems), capable of supporting all their target properties.

We evaluate each of our distributed capability file systems using the criteria proposed by Miller *et al.* [18] for non-distributed capability operating systems, and using the security principles of Saltzer and Schroeder [19]. Our motivation is to extend prior analysis in this domain, with the following goals:

- Confirm (or correct) the claimed security properties for each system.
- Determine if capability-based distributed file systems exist satisfying all desirable properties described by Miller *et al.*
- Highlight tradeoffs made by designers in balancing the properties of capabilities and distributed systems.
- Confirm (or amend) the claimed relationship between the primary properties outlined by Miller *et al.* and a set of secondary, or emergent, properties (the later believed to be derived from the former).

We find that the distributed file systems we survey do not fall neatly into the capability models identified by Miller *et al.*, challenge some previously identified patterns among capability systems, and leave open the possibility of new DFS designs with desirable security properties.

1.7 Research Questions

We seek to interpret essential properties provided by capability systems in a distributed context. We adapt a metric from Miller *et al.* to enable our survey work, analyzing the security of distributed capability systems [18]. This metric may be successfully put to use for future proposed platforms employing capabilities in a cloud context. In particular, our work contributes to the following larger research questions:

- Are capabilities able to produce the security environment needed for the cloud?
- Do systems that meet this capability metric satisfy the cloud security requirements?
- Identify systems may be able to satisfy Navy cloud security requirements?

Before resolving any of these, we first consider which existing distributed file systems meet all the requirements outlined by our metric for distributed capability systems. The goal is to identify the key principles of capability systems in those distributed systems that may contribute in the same way to the security of cloud-scale clustered file systems.

1.8 Organization

Our work takes the following organization:

- In Chapter 2, we review capabilities and related work.
- In Chapter 3, we introduce the Saltzer and Schroeder security principles and the properties for evaluating capability systems defined by Miller *et al.*
- In Chapter 4, we review the capability-based file systems we survey.
- In Chapter 5, we survey the properties of our target systems.
- In Chapter 6, we discuss essential patterns and summary findings with respect to applying the metric of Miller *et al.* to distributed capability systems; we suggest refinements and interpretations.
- In Chapter 7, we conclude and outline future work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2: Background

In this chapter, we remind the reader of the general theory of how a capability system operates; we give a brief history of implementations, and we review recent work employing capabilities in related mobile, web and distributed settings, applicable to the cloud.

2.1 Capability Systems and History

There are more than a few interpretations of what constitutes a *capability*. Therefore, it is understandable as to why there are some differences in the understanding of their derivative properties. The idea is to encapsulate all software in least-privilege protection domains, and to grant access to the objects belonging to domains by means of special protected pointers called capabilities. Its useful before explaining capabilities further to define what we will mean by the terms object and subject. An object is any data entity that can be named and manipulated (e.g., files, images, data records, directories, and protection domains). A subject is a process that can access an object and must abide by the rights permissions granted it by other subjects.

2.1.1 What are Capabilities?

We choose the following description as it complies with most major system implementations. A capability is a protected bit pattern which simultaneously identifies an object (designates) and grants authority to access that object to the subject which holds it [19]. Capabilities should be unforgeable [20]. Unlike most systems we see today, in a capability system, resources do not reside in a global shared namespace. Instead they are created by a *subject* within its own subject-space [21, p. 6]. A subject's own space cannot be accessed without possessing a capability to either that space or a specific object in that space.

Depending on the system, a subject can be *fine-grained* (e.g., an individual process), or *coarse-grained* (e.g., a user). A subject can grant other subjects specific types of access to resources it creates or has capabilities to, by passing them capabilities to the resources. This is called *delegation* of authorities. The only form of access control in a pure capability system is showing possession of a capability to a resource. Without a capability, there is no way to access a resource; in some systems there is no way to even refer to a resource without a capability to it [18]. Unless a process is explicitly granted a capability, it cannot access the object named by the capability.

2.1.2 Why Capabilities?

Many researchers have expressed that capability systems have unique potential to implement secure systems. Saltzer and Schroeder state that "the capability system has as its chief virtues its inherent efficiency, simplicity, and flexibility" [20]. Most capability systems facilitate the principle of least privilege in a way that appears unmatched by other systems, that is, each *process* or instance of an *object,* is delegated only the authority it needs to execute its assigned tasks [21], [22], [23]. Miller argues that a capability architecture provides a far more sound foundation on which to provide precise, minimal, and meaningful delegation of authority [18]. According to Hardy, capabilities are the best way to prevent what he calls the *confused deputy problem* [19] (see Section 3.3.3). It is the presence of the privileged state in other systems that has allowed minor system flaws to yield dramatic escalation of privilege attacks. Denning observes that a by-product of capability systems is the elimination of the need for a *privileged state* (i.e., *root*) [17, p. 374].

2.1.3 A Brief History

Following the invention of capabilities by Dennis and Van Horn, the first practical implementation of a capability system was a modification of the PDP-1, called PDP-1 Supervisor, built in 1967 at MIT with Dennis's assistance [24]. It was a timesharing operating system that supported up to five simultaneous users [25]. The first architecture specification for a capability machine was made by Robert Fabry around 1967. He called it the MAGNUM machine (for magic number, because a capability was like a magic number that conferred access) [26, p. 39]. Maurice Wilkes extoled capability addressing in his 1968 book Time Shared Computer Systems [27]. The first commercially-produced capability system was the Plessey System 250. This system was produced primarily to be a reliable telecommunications switching computer, though significant military procurement suggests RADAR switching use as well [25], [28].

The Hydra system was implemented by Cohen and Jefferson at Carnegie Mellon as part of a DARPA project in 1975 [29]. Hydra was a fault tolerant multi-processor system that was the first to implement a fully object-based design [25]. In 1976, the Cambridge CAP computer was brought on line by Wilkes and Needham [21]. It served as an experimental machine, then later provided computing services in support of research and remained operational for a number of years [30]. In 1980, IBM delivered the System/38, a capability-based machine for commercial distribution [31]. The system met with some success; however, the follow-on IBM system AS400 dropped the capability addressing design [25], [32].

KeyKOS, a capability system designed to support "secure, reliable, 24-hour availability for applications on Tymnet hosts," grew out of the Great New Operating System In the Sky (GNOSIS) effort. The system began operating in 1983 and production continued for nearly ten years by the Key Logic team. Norm Hardy was the lead architect who developed it to be run on the IBM S/370 [33]. KeyKOS successfully ran VISA transaction processing and networking applications [34]. In 1999 and the early 2000s, the Extremely Reliable Operating System (EROS) took much of the lessons learned from KeyKOS, made it as fast as its contemporary systems, and formally proved the property of confinement for the system [35]. There are currently two projects which continue to extend this research, CapROS, (led by Landau, an original KeyKOS designer), and Coyotos (led by Shapiro, the creator of the EROS follow-on). Coyotos is reportedly being commercialized [36].

2.2 Capabilities in Cloud Systems

There are a number of capability systems and active areas of research that do not fall within the scope of our distributed file system survey work, but are worthy of mentioning.

2.2.1 Capsicum

An ongoing Cambridge project, *Capsicum* is a capability-based lightweight operating system built as an extension to FreeBSD 9. It extends the modified POSIX application programming interfaces (API), leveraging two new kernel primitives and a modified userspace API. It allows an array of options to implement hybrid capability programming designs within the OS [37]. This hybrid approach allows programmers to incrementally phase-in least-privilege handling of file operations in a traditional UNIX environment.

2.2.2 CHERI

Recently, an on-going Defense Advanced Research Projects Agency (DARPA) project, the *CRASH-worthy Trustworthy Systems Research and Development* program, led by Stanford, Cambridge, and Google researchers, seeks to build a clean-slate hardware and software capability design. Capability Hardware Enhanced RISC Instructions (CHERI) is a hybrid capability system and represents the hardware portion of the project. It combines "capability-based addressing with RISC ISA and MMU-based virtual memory" and demonstrates strong performance while enforcing memory protection [38]. The design employs a capability coprocessor (defining capability registers), and tagged memory (protecting capabilities stored in memory) [38]. This system is being designed to be the substrate for a future Clean-Slate Design of Resilient, Adaptive, Secure Hosts (CRASH) platform.

2.2.3 seL4

seL4 is a capability-based micro-kernel for ARM, with the goal of securing mobile platforms. This mobile operating system kernel is capability-based and has been formally verified to be *functionally correct* with respect to the following properties: the kernel

- will not crash.
- will not perform unsafe operations.
- is completely deterministic in every situation.

seL4 is the first comercial-grade general-purpose, microkernel; and the implementors decided that would be best accomplished using capabilities as a basis [39]. On-going research exists to formally prove seL4 implements a type of memory integrity and authority confinement [40]. Further, complimentary work seeks to implement a L4-based micro-hypervisor able to support general OS virtualization [41].

2.2.4 BCAP

Google's *Belay Cloud Access Protocol* (BCAP) project is a capability-based web API that enables code to "create and use capabilities as a means to securely enable access to resources" [42]. Modules are code and data that executes, like a process. The module can be passed or create a capability that gives it the ability to invoke some object or another module of code. Capabilities in this system are unforgable and resemble the capabilities in traditional systems. The system leverages Chrome to implement a secure browser protocol that enables secure sharing of modules or data [42].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3: Principles and Properties

Before we survey the various capability-like distributed file systems, we establish guiding principles and a metric to serve as a point of comparison to differentiate their relative security strengths and weaknesses. This section defines the principles that guide our notions of what is best in regard to system security properties. These security properties are the core metrics in the survey tool with which we evaluate the target distributed file systems.

3.1 Security Principles

In their seminal paper *The Protection of Information in Computer Systems*, Saltzer and Schroeder identify eight foundational design principles to guide the creation of security mechanisms [20]. We explain and interpret these in an access control policy setting (i.e., a policy governing access to objects by subjects), which is both classic and relevant to us.

- *Fail-Safe Defaults*. The default condition of any subject is a lack of access to any object. Subjects that require access to an object must therefore be explicitly granted access to that object.
- *Complete Mediation*. Every attempt to access any object within the system must be authorized before access to that object is granted.
- *Least Privilege*. Every subject within a system, no matter the granularity, whether a user or a specific program instance, should have only those privileges which it requires to complete its assigned tasks, and no more.
- *Economy of Mechanism*. Any security measure that is to be implemented should be kept as small and as simple as possible.
- *Least Common Mechanism*. The sharing of resources between subjects should be minimized or eliminated when possible.
- *Separation of Privilege*. A system should only grant permission based on when two or more conditions have been met.
- *Open Design*. The security of a mechanism should not depend on portions of its design or implementation being kept secret; secrets should be represented only by highly changeable keys, so that the whole of the mechanism is not compromised

when a secret is not kept.

- *Psychological Acceptability*. Any security mechanism should be seamless to the user, reflecting that its use is easier than any attempt to circumvent its use.

Saltzer and Schroeder also suggest two additional security design principles they contended "imperfectly translate" to computer systems, naming these "Work Factor" and "Compromise Recording." They are very useful nonetheless. We present them as broadened principles that may be applied to systems and design decisions:

- *Adequate Protection*. Objects should be protected to a level commensurate with their value [43].
- *Accountability*. Security mechanisms should show when they have been tampered with, if at all possible. This turns out to be at least very difficult, if not impossible to do, especially with a very knowledgeable and skilled attacker. It can be approximated by an audit system that records every access or use of a resource or security mechanism. This enables post-incident forensic analysis on the system to provide indications of attack and attack source.

Mechanisms for one of the principles above may complicate or even contradict mechanisms for other principles. Designers must determine which take priority in their system's requirements. The principles are guides for design, not binding rules [44].

Many of the Saltzer and Schroeder principles manifest naturally in capability systems. We discuss the relationship between typical capability systems and these principles next:

- In a capability system the default is that a subject has no access to any object. A subject gains access either by creating the object itself or by being granted the capability. The base case for a subject is an empty capability list (C-List). This follows the principle of fail-safe defaults.
- Because capabilities are the *only* method of authority invocation in capability systems, and every capability simultaneously names (designates) an object and grants specific rights to the object, it follows that it is impossible to access a resource without a capability. Complete mediation is a necessary consequence in such a system.
- Capabilities facilitate the implementation of the principle of least privilege. Since

transferring capabilities is the only means of transferring authority, a security-conscious programmer may easily ensure that only the authorities that the new process will need to execute its intended purpose are given to it, and no more. This is a marked contrast with a non-capability system that relies on root-associated privileges.

- With regard to economy of mechanism, Saltzer and Schroeder attribute efficiency and simplicity as the virtues of a capability system [20, §II-b-3].
- Capabilities support least common mechanism as well. This is especially true if resources and subjects are finely grained to the level of individual files or instances of processes, and subjects require capabilities to one another in order to communicate.
- Capabilities also naturally lend themselves to separation of privilege as demonstrated early in their history through the Hydra and CAL capability operating systems [20]. A simple implementation requires that in order to access a particular data object, two or more capabilities must be obtained by the would-be accessor.
- A capability is an unforgeable token that, when invoked by *any* subject, is sufficient to grant access to its designated resource. Because capability validations in their non-augmented state do not require identities, mapping an action back to a specific identity can be very difficult if not impossible, without additional mechanism [45]. As a result, capabilities do not axiomatically lend themselves to the principle of accountability.

Open design, psychological acceptability, and adequate protection are implementation dependent and orthogonal to the properties of capability systems. That said, all known capability systems to date have followed open designs. Additionally, there has been little or no investigation of the psychological acceptability of capability systems. This is an interesting avenue for further research and would require a human factors analysis study of the various implementations of capability systems.

3.2 Object Capability Properties

In the years following the invention of capabilities by Dennis and Van Horn in 1966, several successful capability machines were built [24], [28]. That led Wilkes and Needham to design a general purpose time sharing system in the 1970s called the Cambridge CAP Computer. They ran into problems of complexity with their design (1979) [21]. Their doubts took root and became accepted beliefs that capability systems would not work. The myths have propagated by people who did not understand the principles of capability systems.

In 2003, Miller, Yee, and Shapiro challenged the myths in the paper *Capability Myths Demolished.* They define a number of system properties, establishing a metric to distinguish between different types of capability systems [18]. They argue that many commonly understood deficiencies of capability systems were in fact limited to only certain classes of capability systems. Furthermore, they define a class of capability system—called *object capability systems*—whose properties avoid many of those limitations that had come to be widely accepted as the direct consequence of capabilities. Next, we describe the capability properties outlined by Miller *et al.*, and interpret them, explaining the method by which we determine if a capability file system satisfies each.

A. No Designation Without Authority
B. Dynamic Subject Creation
C. Subject Aggregated Authority Management
D. No Ambient Authority
E. Composability of Authority
F. Access Controlled Delegation Channels
G. Dynamic Resource Creation

3.2.1 A: No Designation without Authority

No designation without authority is the property requiring that there exists no mechanism other than a capability by which a resource may be designated (named). If there exists some method to reference or point to an object by a subject lacking the authority to access it, then the system does not have this property. According to Miller *et al.*, a capability simultaneously *designates* a resource (as in addressing) and encodes the *authority* to use that resource (as in access) [18, p. 3].

Miller *et al.* credit this property as enabling systems to avoid global, shared namespaces for resources [18]. Wilkes and Needham recognized the merits of this property much earlier, in the context of the Cambridge CAP system, observing that, much like the scoping rules of a high level programming language where "a programmer has no way of addressing a variable that is out of scope... similarly, the possession of a capability for an object gives the programmer the means of addressing that object; without the capability, he cannot validly even refer to it" [21, p. 6]. They saw this property as a natural form of memory protection stemming from a capability design.

The test for property A is the following question: For a given system, *is it possible for a subject to designate a resource without also possessing the corresponding authority?* If the answer is *no*, then this property is met by that system.

3.2.2 B: Dynamic Subject Creation

Dynamic subject creation is the property requiring that any process (subject) can dynamically create a new process *and* grant that child a strict subset of the parent's authority. In other words, parent processes cannot create a child with more authorities than it (the parent) has itself. This refers to creation only. A subject can gain authorities if some other subject with that authority grants it. This property is often used to guarantee that a subject can launch an untrusted process with the minimal set of capabilities that it needs to accomplish its purpose.

Miller *et al.* claim that capability systems enable a much finer granularity for authority management during subject creation, compared to systems using access control lists (ACLs). In ACL systems a domain corresponds to a user level account. In capability systems, a domain is defined by a C-list and can be associated with a subject as fine-grained as the designer wishes, even to the level of an individual process or software component.

The test for property B is the following question: *Does the system allow a subject to dynamically create a new subject, granting it only a subset of the parent's authority?* Given a system can do this, it has this property.

3.2.3 C: Subject Aggregated Authority Management

Subject aggregated authority management requires that a subject controls its own list of authorities. A subject can only downgrade its authorities and privileges. It can delete them, pass them as is or downgraded to other subjects, or create new subjects to whom it passes only a subset of its own authorities. A subject can gain an authority only when granted by another subject that has that authority.

As a result of this property, each subject (process or object) has no authorities granted by its environment. Its only authorities are those granted on creation or subsequently passed to it by subjects already having those authorities. Thus this property supports the next one, property D.

This property is conspicuously absent in ACL systems, where the access control list is necessarily associated with the object for which it provides access and protection. It is also absent in systems that allow supervisor states or supervisor privileges.

The test for property C is the following question: *Is the power to edit authorities aggregated by subject?* If the answer is yes, then the system has this property.

3.2.4 D: No Ambient Authority

No ambient authority means that no subject gets authorities implicitly from its environment. Miller *et al.* describe it as "...authority that is exercised, but not selected, by its user" [18, p. 8]. Ambient authority is analogous to a general power of attorney. Even though no specific rights are conveyed by the document, the general power of attorney allows the party who possesses it to do and exercise any authority its signatory would have had legal right to do his or her self. In the same way ambient authority, as in Linux or another ACL system, allows a process to act with the rights of the user or process that called it, even if some of those rights have nothing whatsoever to do with the purpose or actions it was called to perform. This behavior is a violation of Saltzer and Schroeder's *principle of least privilege*.

Thus, "no ambient authority" means that no general power of attorney exists. Instead, the signatory grants specific special powers of attorney to allow another party to accomplish a set of tasks on his or her behalf. Recipients must present the specific special power of attorney that grants them the authority to perform a particular task.

Most capability systems have this property. There is no ambient authority to convey to a subject(a process or object). A subject's authority comes solely from its C-list. When it wants to access a resource it must select which authority in its C-list to use.

The test for whether a system has property D is the following question: *Must subjects a priori select which authority to use when performing an access to an object?* If so, then the system has this property.

3.2.5 E: Composability of Authority

Composability implies that a system's subjects and objects can be freely combined and recombined in various networks of relationships to achieve different, yet reliable and predictable properties [44].

In a system with *composability of authority* subjects are functionally equivalent to resources, meaning "every subject is a resource, and every resource is conceptually a subject" [18, p. 9]. When the interfaces exposed by subjects and objects are equivalent (e.g., when requesting access to an object is functionally equivalent to sending a request to a subject), then networks of subject/resource relationships can be composed to any depth.

The test for property E is the question: *Are resources indistinguishable from subjects in this system?* If yes, then this property is present.

3.2.6 F: Access Controlled Delegation Channels

For a system to possess *access controlled delegation channels*, an access relationship must exist between any two subjects in order for a capability or authority to be passed from one to the other. Subjects are not allowed to pass their capabilities over unauthorized channels. The sender can only pass a capability if it possesses a capability to communicate with the receiver.

The test for property F is the question: *"Is an access relationship between two subjects X and Y required in order for X to pass an authority to Y?"* [18, p. 10]. If the answer is yes, the property is present in the system.

3.2.7 G: Dynamic Resource Creation

Dynamic Resource Creation has to do with a system allowing subjects to create resources and to express restrictions on those resources' accessibility. A system has this property if subjects can dynamically create new objects and set restrictions on their accessibility. This property complements the granularity issue expressed in property B. The ability to dynamically create new resources is a "dividing line between fine-grained and coarse-grained" [18, p. 10].

The tests for property G are the questions: *Can the system dynamically create new objects?*

Further, can a subject express access restrictions on objects as they are created dynamically? If the system manages a static set of resources with a static policy, then the answer is necessarily, no. Additionally, if interfaces for creating resources provide no option for assigning policy then the answer is also no.

3.3 Emergent Properties

Having defined, explained, and established a metric to determine whether a system has each of the Miller *et al.* properties, in this section we express the claimed consequences of these properties. We call these consequences secondary properties, or *emergent properties*. The following is a brief explanation of that reasoning.

3.3.1 Revocation

In the context of operating systems and file systems, *revocation* can refer to a number of related, yet subtly different system attributes. It is useful here to distinguish from among those a few that are germane to our survey (i.e., *immediate revocation, temporal revocation*, and *selective revocation*). Immediate revocation has to do with whether a system immediately removes access to an object previously made accessible, or if some period of time must pass before access is removed [29]. This can be an important aspect of revocation to consider in distributed systems. Temporal revocation has to do with whether it is possible to revoke access then later reinstate it [29].

Finally, *selective revocation* is the specific aspect of revocation we consider central to our discussion. It is the ability within a system to terminate a single specific subject's authority to access a resource when they have previously been given that authority without affecting other subject's access [46]. Consider a subject, Alice, with a capability to a customer database. She grants Bob access by giving him a copy of the capability. At some later time Alice determines that Bob is no longer trustworthy and decides to revoke Bob's access authority to the customer database. If the system provides a mechanism for Alice to accomplish this, then the system supports selective revocation.

Miller *et al.* observe that, historically, some types of capability systems have not supported this type of revocation. In particular, selective revocation is not possible in those systems where property B is present but property E is not.[2]

Miller *et al.* constructively show that for object capability systems, selective revocation can be achieved using forwarding proxy subjects; that is, creating new subjects, composing these with the target object, then using the new forwarding proxy subjects to extend the original object's access to the desired recipients. If the originating subject later changes its mind about the authorities it delegated, it need only destroy the intermediate proxy subjects to revoke the recipients' access to the target object.[3]

As a preemptive polemic, we also note that a solution that would rely on the cancellation of an entire capability to a resource does not meet the requirements of selective revocation, as this would be a costly and difficult practice. The system would have to redistribute the new capability to the revoked resource to all subjects that were still authorized its access. The nature of a capability system prevents tracking or having this knowledge, and any mechanism to track and keep it would significantly tax the benefits of that capability system. Finally, in that event, there would likely be a significant period of non-availability as this process was carried out.

3.3.2 Confinement

Confinement is the property of a system such that when an untrusted program is executed on it, the system does not permit that program to leak information to untrusted parties [47]. Miller *et al.* observe that, historically, some capability systems have been unable to control the bounds on the propagation of access rights (i.e., delegation) through the system and, thus, restrict the extent to which data is shared (i.e., confinement). They claim systems unable to achieve confinement are those where property B is present and property F is not.[4]

Originally, Lampson identifies seven ways in which an untrusted program could accomplish leaking sensitive information [47]. The property of confinement described by Miller *et al.* is a strict subset of the notion of confinement described by Lampson, in that it does not consider covert channels such as processor timings, however it does consider all the overt

[2]Miller *et al.* express this observation as the logical proposition $B \wedge \overline{E} \rightarrow \overline{Rev}$ [18, figure 15].

[3]As referenced by Miller *et al.* this idea is not new, it was first introduced by Redell with respect to his *Typical Capability System (TCS)* both of which he introduced in his 1974 MIT dissertation, *Naming and Protection in Extendible Operating Systems* [46].

[4]Miller *et al.* express this observation as the logical proposition $B \wedge \overline{F} \rightarrow \overline{Conf}$ [18, figure 15].

ways in which a program may misbehave and attempt to leak information.

3.3.3 Confused Deputy

A *confused deputy* is a program that is "fooled" by another program which abuses or misuses its authorities. The phrase was coined and the phenomenon described by Norm Hardy in his 1988 paper, *The Confused Deputy* [19]. Miller *et al.* describe it as a process "...that has been manipulated into wielding its authority inappropriately" [18, p. 11]. This may occur when an authorization given by one party for a certain purpose is used by a process to access a resource designated by a different party for a different purpose. This brings about an unintended transfer of authority to that separate purpose, enabling abuse. They further observe that when "designators and authorities take separate paths through a system, their recombination is likely to lead to confused deputies" [18, p. 12].

The *confused deputy* problem is the root of how many forms of malware are able to take advantage of systems [19]. Miller *et al.* admit that nothing can prevent the possibility of a confused deputy if a programmer is determined to write bad code, but offer that "...certain properties of a security model can have a profound effect on our likelihood of writing reliable programs" [18, p. 12]. They propose that a gradient of three preparedness levels exists with regard to a system's propensity for eliminating Confused Deputies. They are *danger* (missing property D, no ambient authority), *better* (having property D but missing property A, no designation without authority), and *best* (possessing both property D and property A).

As a note here, both *confinement* and *confused deputy* are related to one another, and related to the *principle of least common mechanism.* A system implementation that ensures confinement and minimizes the likelihood of a confused deputy is likely on the road to implementing this principle well.

3.3.4 Least Privilege

The final property that Miller *et al.* relate to properties A–G is *least privilege.* They state that least privilege operation requires that the minimum number of subjects are granted access to the minimum number of resources at the lowest permission level possible. Additionally, policies among subjects and resources must be expressed at a very fine granularity. Recall that property B enables subjects to be distinguished at a much finer granularity (i.e., per process), compared to systems without the property. Recall as well that property G likewise enables fine-grained access restrictions on objects. Thus, Miller *et al.* argue that the potential for least privilege operation can be maximized by the presence of both property B and property G. In particular, these properties are described as necessary, but not sufficient, prerequisites for least-privilege operation.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4: Capability-based File Systems

In this chapter, we introduce the target systems for our survey. We target a variety of distributed file systems (DFS) claiming to utilize capabilities for security. We include a simple, non distributed file system implemented using object capabilities, primarily as a point of comparison for our survey.

4.1 SFS

The Simple File System (SFS) is an object capability file system implemented for the Extremely Reliable Operating System (EROS) [34]. EROS has been formally verified to have the properties of confinement and revocation [34], [48]. Miller *et al.* [18] argue that EROS has the best possible properties of capability systems, including least privilege, and prevention of confused deputies.

In EROS, all resources are instantiated as processes and are held in-memory. Periodically, these active processes and all of the machine's operational state is flushed to the hard disk through what it calls a *checkpoint*. In this way, EROS implements persistence and fault recovery [49].

Each process in EROS is associated with a capability register, managed by the kernel, which holds the capabilities that the process can invoke directly [34]. The only method by which processes can interact and provide services to one another is through capability invocation. *Capability invocation* is essentially inter-process communication, where the capability both designates the process to which it is going and grants the authority to accomplish its purpose at that process [49].

SFS implements only two types of resources: file objects and directory objects. The file object process presents a simple interface, including the write call, read call, check alleged key type call (specifies whether a file or directory), and seek call (return or set a file's read or write offset). The directory object presents an interface that allows another process to read the contents of the directory, remove a file or subdirectory, create a new file or subdirectory, and check alleged key type. The file system consists simply of a tree of directory and file processes that are brought online as they are traversed or opened.

4.2 DOI

The Digital Object Identifier (DOI) system is a handle-based system for accessing resources on the Internet. It resembles, at least superficially, a capability-based distributed file system using unguessable identifiers in a global namespace. The DOI system that has experienced the widest adoption is the implementation described by the Corporation for National Research Initiatives (CNRI) [50]. This system is used by most publishers, movie studios, and even the Library of Congress [51]. It is made up of three constituent parts: digital object repository (dorepository), the handle system, and digital object registry (doregistry).

A dorepository provides storage and continuous access to the digital objects (DO) it contains. Dorepositories are interoperable, using a standard extensible interface protocol called Digital Object Protocol (DOP) [51]. DOP enables the DO's administrator to easily transfer the DO from one dorepository to another, while preserving all access control settings, provenance information, and other important metadata about the DO [52]. Additionally, and most importantly, the handle for the DO during such transfers remains entirely unaltered.

The handle system provides DOI resolution to the overall DOI system. It is a distributed system providing an extensible global name service for DOs. Each DO is associated with a *handle*, which is a persistent, globally unique name. The handle system accomplishes this by creating and storing a handle record for each handle-DO pair. In addition to the handle itself, the handle record stores other known identifiers and useful state information about the associated DO; for example, the IP and MAC address where it is stored, or information to verify the authenticity of the DO, etc. [52]. The handle system protocol enables the system to resolve these handles, and to respond to queries for them [53].

Lastly, within the DOI system, a doregistry is a final layer of abstraction that allows users to define and manage collections of DOs for searching and browsing, even when constituent DOs are stored across separate dorepositories. The doregistry facilitates protected access to these collections, from full private access, to group access, to complete public availability [51].

4.3 CapaFS

CapaFS is a global, decentralized file system allowing users to collaborate with each other, with no prior arrangements [54], [55]. The system uses *capability file names* (CFN) as sparse capabilities to name and grant access to files on remote servers. CapaFS allows remote users possessing a CFN to access the local file or directory it designates [55]. Though the files are stored remotely, they are interacted with as if they are stored locally. CapaFS is not a clustered file system, in fact, unlike other DFSs all files must be created on a local server; that is, file creation cannot be done on a server from a remote client.

CapaFS consists of two parts: a shared library replacing libc on client, and a user level file server that acts as a proxy for remote users to access the local filesystem. The shared library *wraps* file operations such as *open, close, read, write, lseek and fcntl*, extending these to handle capability file names. When a client calls *open* on a remote file via its capability file name, the library establishes a secure connection with the remote server whose IP and port number are encoded in the CFN [55]. Once established, the client passes the CFN to the server, which verifies its validity and issues a temporary sparse string key for use in all subsequent interactions during that session.

Each CFN encodes two parts: the client part and the server part. The *client part* consists of a distinguished CapaFS namespace designator, the IP and port for the proxy service used to access the resource, and optionally, its public key. The *server part* contains a cryptographically protected representation of the local resource's path and access rights [55]. Later the creators of a another capability based file DFS levy the criticism that CapaFS's CFNs were "long and meaningless" and as a result were difficult for users to remember or interact with, necessitating the creation of persistant symbolic links to overcome this [56].

4.4 Maat

Maat is a system that extends an existing high-performance cluster file system, such as Ceph [57], Panasas [58], or zFS [59]. Performance for high-throughput applications is an essential goal for Maat, which has reported handling peta-scale amounts of data with as little as a $6-7\%$ additional overhead [60]. This goal has motivated many of the design features of Maat, including how capabilities are handled. The details of Maat described in this survey pertain to the prototype developed for Ceph.

Maat consists of three main components: the client, a metadata server (MDS) cluster and an object storage device (OSD) cluster. Clients authenticate and log in with the MDS in a manner much like Kerberos [61]. The clients then are able to establish secure communications with the MDS and each OSD. Clients make *open* requests to the MDS for resources they need. The MDS issues cryptographically signed capabilities back to the client, enabling access to the requested resources. The client is then free to use the capabilities to perform the desired file I/O with the distributed OSDs. The OSDs are able to verify the capability once using the MDS's public key, then cache the result for future I/O calls with that capability [60].

Maat's capabilities have some noteworthy augmentations. They are designed to have short lifetimes, such as a five-minute expiration time. These short lifetimes enable *automatic revocation*; if a client's access is revoked, the capability is simply not renewed, allowing the capability to time out. If a client continues to need the file and its access has not been revoked, the MDS reissues a capablity to it automatically before the expiration occurs. Another feature of Maat is that authorized user identities are encoded into the capabilities. This ensures that an eavesdropper cannot steal and use the capability for unauthorized access to the files [60].

Finally, Maat also introduces the concept of *extended capabilities* to ameliorate the cost of generating and managing tokens between every client and resource, instead using a single token to embed policy between an arbitrary number of clients and resources [60].

4.5 Tahoe-LAFS

Tahoe-LAFS (Least Authority File System) is an open source distributed file system making extensive use of cryptography to reduce trust in its operation [62]. If some portion of Tahoe-LAFS nodes fail or are compromised by a malicious party, the filesystem can continue to provide confidentiality, integrity, and availability. This system has been used in operation commercially for several years: in an early incarnation as the service *allmydata.com* [63], and recently as the cloud-based *S4* service [64], making use of Amazon's S3 infrastructure. The designers openly encourage hackers to break Tahoe's security claims via an online challenge and reward [65].

Tahoe is designed around the idea of *provider-independent security*, meaning the "service provider never has the ability to read or modify your data" [66]. Tahoe claims to use "...capabilities for access control, cryptography for confidentiality and integrity, and erasure coding for fault-tolerance" [62, p. 1].

Tahoe distributes its client data and metadata across a number of servers on its *grid* [62]. A Tahoe grid is made up of servers or what we refer to as *storage nodes*, clients or *gateway nodes*, and one *introducer*. The introducer is a special type of server whose fixed address and port are hand-entered into every new node. Upon a storage or gateway node start-up, it first communicates with the introducer, which then notifies the new node of all other nodes on the grid so that it can establish communications with each [66].

A user can only access a resource on a tahoe grid if that user has a capability to that resource and presents that capability to a node. Tahoe has three basic types of capabilities: read-write capabilities (*read-write-cap*), read only capabilities (*read-only-cap*), and verify capabilities (*verify-cap*). From a writecap a user can derive a readcap. Likewise from a readcap, a user can derive a verifycap. A verify-cap enables its holder to verify the integrity of a file, but not learn the file's plaintext [62].

A gateway node connects a user or group of users on a mutually trusting network to the Tahoe storage grid, which is distributed over an untrusted network, such as the Internet. Data is encrypted inside a user's own trusted network on their gateway node. The gateway splits the encrypted file into n erasure coded shares. To retrieve a file, any $k <= n$ shares must be retrieved and recombined by the gateway. This logic is transparent to the client behind the gateway [66].

There are two types of data storage files on Tahoe, mutable and immutable. An *immutable file* cannot be changed after it has been created. A *mutable file* can be written and rewritten an unlimited number of times. A directory is an example of a mutable file [62].

4.6 DisCFS

The Distributed Credential FileSystem (DisCFS) is a distributed file system that enables users to access files on a remote server and collaborate with users from other domains [67]. It is not a clustered file system, but a distributed file system in the style of NFS or CapaFS.

DisCFS delegates access rights by allowing subjects to issue *credentials*, which are “a direct binding between a public key and a set of authorizations” [67, §4.1]. A credential is a type of capability, providing to the subject both designation and authority to access a resource. A credential has the added restriction that the subject must hold a corresponding private key to complete a signature chain associated with the credential before the server will allow access to the resource. This feature allows credentials to be passed arbitrarily and in the clear, without extending access to unauthorized third parties. Only the subject who is specifically issued, or intentionally delegated a credential through extending the signature chain, can use it to access a resource [67].

DisCFS is implemented over NFS, and thus supports a single remote server or a clustered server configuration. The client software runs on a user’s workstation and uses a valid credential to esablish secure communication to the server. DisCFS resources appear to a user as a mounted file system. Files for which valid credentials were supplied to the server will appear under the mount point like local files [67].

In DisCFS, minimally a single administrative setup action is required to initialize and make available for sharing existing resources: issuing some original user a credential binding his or her key to the resources (directories and files) they own. Thereafter, that user and subsequent generations of users, as allowed by policy, can delegate all or some subset of their authority to other users. Delegating the abilities to read, write, execute or even create new resources (files or directories) within existing directories [67, §5]. This is accomplished by issuing users a new credential, derived from an existing credential. Rights are appended to the existing credential, authorizing the new user to access the files. The delegator includes the new user’s public key, and signs the new chain of credentials. This chain and the new credential are a delegated capability that can be used to access the resource.

4.7 Neo

Neo is a distributed file system supporting flexible user-defined access control policies, accountable access, revocable authorities, and confinable access privileges [45]. The system is comprised of three types of components: clients, block servers to store file data, and a single, trusted metadata server to manage chits and store file metadata. All communication between components is over untrusted channels.

Policies on resources are mediated by a capability mechanism using an XML-based access token, called a *chit*. The chit differs from many capability systems in that it is not a secret token, but is an XML object paired with a *fingerprint* hash, to prevent tampering or abuse. There are three types of chits: the *master chit* associated with the user originally uploading the resource, and two derivative chits, called *authenticated chits* and *unauthenticated chits*. Authenticated chits use embedded public keys (similar to DisCFS's credential) to allow the metadata server to challenge the user holding the chit to verify it has a matching private key. The unauthenticated chit does not require authentication and must be passed via secure channels (much like CapaFS's capability file names), as any subject who learns the chit may employ it [45].

A chit can give read or write access to some set of files, or a tree of directories and their associated files. A subject holding a chit can add *tags* to a chit to edit the encoded policy before passing the chit to another subject. These tags can be used to label the chit (so that label will be logged during any transactions with the server), narrow the privileges the chit conveys, cause the chit to expire earlier than the existing chit, revoke the access of existing derivative chits, or even make the chit unable to be delegated further. These tags can be added, but they cannot be removed. Thus authority can only be decreased and never increased during the creation of derivative chits [45].

Though chits can be shared as allowed by the policies enacted by a chit's creator, there is no necessary limit to the number of users that can interact with the server via derivative chits. Only a single user with a conventional server-based user account is needed to create the original master chit for the shared resources [45].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5: Analysis and Findings

In the course of our survey of capability-based distributed file systems, we found no systems exhibiting possession of all of the Miller *et al.* properties. In particular, we found no examples of object capability *distributed file systems* (DFS). Four of the six DFSs—CapaFS, Tahoe-LAFS, DisCFS and Neo—have a very similar distribution of Miller *et al.* properties. These resemble a Capabilities as Keys (CAK) model, with Tahoe matching perfectly and CapaFS, DisCFS and Neo differing only by the absence of property D. The remaining two DFSs, Maat and DOI, depart significantly from any of the capability models described by Miller *et al.* and appear to have more in common with an ACL model. Table 5.1 summarizes our findings for each system, each of which are explained in depth in the sections that follow.

	Property	Obj Cap DFS	Caps as Keys	SFS (EROS)	DOI	CapaFS	Maat	Tahoe-LAFS	DisCFS	Neo
A:	No Desig w/o Authority	Y	N	Y	N	N	N	N	N	N
B:	Dynamic Subj Creation	Y	Y	Y	N	Y	N	Y	Y	Y
C:	Subj Agg. Authority Mngt	Y	Y	Y	N	Y	N	Y	Y	Y
D:	No Ambient Authority	Y	Y	Y	N	N	N	Y	N	N
E:	Composability of Authority	Y	N	Y	N	N	N	N	N	N
F:	Access Cntrl Deleg. Chnls	Y	N	Y	N	N	N	N	N	N
G:	Dyn. Resource Creation	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 5.1: File System Evaluation Results for Miller *et al.* Properties [18]

5.1 Generic System Categories

Miller *et al.* [18] describe four security models in their paper *Capability Myths Demolished*. We reproduce two in Tables 5.1 and 5.2: Capabilities as Keys (CAK) and Object Capabilities. These columns in Tables 5.1 and 5.2 are filled in directly from Miller *et al.*; we refer the reader to their paper for further explanation on those columns.

5.2 SFS

EROS's Simple File System (SFS) is an example of what Miller *et al.* call the Object Capability model, for which each of the seven properties is true. We highlight the underlying features in SFS that support each property.

5.2.1 A: No Designation without Authority

Recall from Section 4.1, every object in SFS is instantiated as a process. As in EROS, for a subect to reference or access an object in SFS, that subject must possess a capability or *key* allowing that type of access to that specified object; that is, the authorization is in the subject's *keyring*[5] [34]. A file or directory cannot be read, written to or referenced without a key granting that authority to the subject. In particular, there is no method to name or designate the resource in the absence of the capability; therefore, property A is true for SFS.

5.2.2 B: Dynamic Subject Creation

In SFS, the subjects are file and directory processes, which hold authorities in the form of keys in their keyrings. When a process creates a new process, it is able to pass a strict subset of its keys to a child process, thereby meeting the requirements for property B.

5.2.3 C: Subject Aggregated Authority Management

As explained in Section 5.2.2, each subject holds its authorities in a structure called a keyring. The keyring holds the keys (capabilities) possessed by the subject. Every subject may edit its own authorities, creating new resources with their associated keys, receiving messages from other subjects containing new keys, or destroying keys in its keyring that are no longer needed.

5.2.4 D: No Ambient Authority

In order for a subject to exercise an authority to access a resource, it must select the specific key to use ahead of time. It passes this key via a system message to the designated resource to which the key both points and grants access. No authority is derived by the process metadata or from the environment, only from the key that it selected. Therefore, SFS demonstrates property D.

[5]*keyring*: elsewhere described as a capability list, C-list, or capability register.

5.2.5 E: Composability of Authority

In SFS, the functional components are files and directories. When created, each file and directory is instantiated as a process with its own keyring. Every resource is also a subject, and every subject also a resource. In other words, requesting access to an object by invoking a capability is functionally equivalent to sending a request to a subject. Therefore, property E is true.

5.2.6 F: Access Controlled Delegation Channels

In SFS, in order for a process to pass a capability (key) to another process it must have a capability authorizing access to that subject. In simple terms, if process Alice wants to pass process Bob a capability to file process Diane, Alice must first have a capability to Bob enabling her to pass that authority to him. This means SFS does enforce access controlled delegation channels.

5.2.7 G: Dynamic Resource Creation

Shapiro *et al.* [34] subscribe to the same understanding and implementation of a capability as Dennis and Van Horn, namely: "a capabilty is an unforgeable pair made up of an object identifier and a set of authorized operations (an interface) on that object" [34, §2]. When a resource-object is created in EROS's SFS, that resource receives both a capability to itself and a capability list of its own in the form of a keyring. The parent process creating the resource puts the capability for the new resource (with read, write and execute permissions) into its keyring. Any process passed a capability to this resource will necessarily possess authorization to use it, as defined by the permissions specified in this capability.

5.3 DOI

Despite initial appearances, the DOI system turns out to be a very different kind of system for which it is inaccurate to describe as either a *distributed file system* or *capability system* [53, §6]. It is more akin to an access control list (ACL) model than a capability-based model.

5.3.1 A: No Designation without Authority

Recall from Section 3.2.1 that property A requires that a system have no form of designation separate from authority. The DOI system has many forms of designation separate from authority, such as the title of the object itself, its local identifier or any of its handle *values*. Most importantly, the DOI, or *handle*, does not convey any authority to access the object. If the handle's administrator chooses to make the object confidential, then the handle does not convey authority to access the resource. Instead, the system will authenticate the subject and perform an access control check [53, §5], [68, §6]. As a result the DOI system does not have property A.

5.3.2 B: Dynamic Subject Creation

As mentioned in Section 5.3.1, the handle does not convey authority to access an object, instead using an access control check based on the subject's identity and an ACL (or, the resource is public and no check is performed). As a result, a subject cannot grant a subset of its authority to a new subject. Instead, a subject relies on the ambient authority associated with its identity, to use any handles it may possess.

5.3.3 C: Subject Aggregated Authority Management

In DOI, the power to edit authorities is not aggregated by subject. Although a handle can be deleted, this has little to do with authority management, as access control lists are managed at the server by the handle's administrator [53, §5]. Access to the object is granted if it is not protected; otherwise, an access control check is performed based on the requestor's identity. This means that DOI implements *resource aggregated authority management*.

5.3.4 D: No Ambient Authority

As stated in Section 5.3.3, authorities in DOI are managed via an ACL associated with the resource. For non-public resources, the server issues a challenge to the requesting client, which must authenticate itself to prove it has authorization to access the object [53, §5]. Thus, subject identity carries ambient authority for all handles to which it has access. Additionally, the DOI system may support identity management so that, after presenting a single DOI and authenticating, access can be provided to a registry where multiple objects can be accessed without specifying any additional DOIs. Administrator privileges are required to edit each resource's ACL permissions. Administrator accounts by their nature require ambient authority to exist.

5.3.5 E: Composability of Authority

In DOI, the subject and the resource to be accessed are fundamentally different: requesting access using a handle is inherently different than sending a request to another subject. The handle is sent to the handle server where it is resolved and an action is taken by the server (e.g., challenge-response followed by returning a pointer to the object's location). The resource being accessed is not a subject that will reply to the request. While it is possible to "chain together" handles, this is not the same as *composing subjects and resources* into a "network of authority relationships to any depth" [18, p. 9].

5.3.6 F: Access Controlled Delegation Channels

No prior relationship or authorization is required in the DOI system to allow the exchange of handles between subjects. In fact, the system was designed to enable "universal information access," in the form of persistent global namespace handles [51]. The premise being it should be easier for everyone to acquire the information they seek. A handle can be obtained by reading it from an advertisement, receiving it in an email or seeing it referenced in a scientific journal. Clearly, no *access relationship* is required between subjects in order for a handle to be passed.

5.3.7 G: Dynamic Resource Creation

DOI allows for fine-grained administrative control of its digital objects. Handle creators can administer and express restrictions on the resource's accessibility, dynamically at creation and any time subsequent.

5.4 CapaFS

CapaFS is the earliest capability-based DFS that our survey evaluates. In many ways, it is a template upon which later capability-based DFSs, like DisCFS and Neo, improved. Its properties in Table 5.1 resemble the CAK model, but differ in that it lacks property D, *no ambient authority*.

5.4.1 A: No Designation without Authority

In CapaFS, files on a server have a path and file name separate from the capability itself. As a result, a resource can be designated without possessing any authority to access it.

5.4.2 B: Dynamic Subject Creation

Capability file names (CFN) are the representation of authorities in CapaFS. In this system, a new subject can be passed a subset of a parent process's authorities; however, at least two exceptions must be noted. First, any local subject with the same identity as its parent subject may have access to all the authorities of the parent on the local host. In light of this fact, the following precautions could be taken to preserve property B when new subjects are created on the local host.

Remark 5.1 *In this system, no assumptions are made about how subjects are managed on the* local system. *Consider a local operating system employing a separation policy between subjects, for example, type enforcement under SELinux. This would enable forked processes to transition to a new domain on creation, and be sandboxed with a fewer set of resources on the local system. The newly forked process can make only a subset of its authorities accessible in this new domain. Alternatively, the local host could employ an operating system itself supporting capabilities with property B. With either approach, its possible to fork a new local process holding only a subset of its parent's authorities. Although this requires additional effort, it demonstrates that it is possible to implement a system with property B for locally created subjects.*

Further, for a subject to pass a *subset* of its authorities, *each of those authorities must be instantiated as separate capabilities*. Miltchev *et al.* [56, §3.2.7] observe that CapaFS does not allow a remote user to diminish the file permissions of a capability (e.g., from a *read and write* authority to a *read-only* authority). This is due to the cryptographic protection of the server part of the capability and the inability to invoke the creation of a new CFN remotely. Not specifically pointed out in Miltchev's analysis, this cryptographic protection also prevents the diminishment of a directory capability to a specific file within the directory.

5.4.3 C: Subject Aggregated Authority Management

Either a subject in CapaFS owns a CFN to a resource, owns a CFN to a resource's parent directory or does not have access to it. There is no other source of authority by which a resource may be accessed. So, in this system, authority management is aggregated at the subject, who may create an authority, be passed the authority or delete the authority from its C-list.

5.4.4 D: No Ambient Authority

A pure CAK model would have this property; however, as pointed out in Section 5.4.2, the authority to access a directory under CapaFS (encoded as a single CFN) yields the authority to access all of the files and subdirectories it contains. More generally, we observe: as currently described, ambient authority could be avoided in CapaFS by requiring CFNs be implemented *only* at the granularity of individual files.

Remark 5.2 *Any system that allows a subject to encode the authority to access multiple separate resources into a single capability exhibits ambient authority. This is because, upon invocation of the capability, these separate authorities are simultaneously and implicitly available to the invoker. No individual authority is specifically selected prior to its use, rather the authority is ambiently available to the subject.*

Thus, CapaFS does not have property D.

5.4.5 E: Composability of Authority

CapaFS is very similar to the capability as keys (CAK) models, for which the following observation applies.

Remark 5.3 *Property E is not reflected in CAK-like systems such as this, where subjects are not equivalent to resources. In these systems, subjects* authorize *other subjects by passing them a capability for a resource. This is contrasted with how subjects* access *objects; that is, they submit a capability to a server and receive access to the resource in return. In these systems, it makes no sense for a resource itself to have a capability to a subject. Instead, the relationship between subject and resource is uni-directional and authority relationships cannot be composed to any depth, as required by property E [18].*

In particular, in CapaFS, subjects authorize subjects by passing a CFN that provides access to a resource, while subjects access resources by submitting a CFN to the server to receive responses holding resource data. This precludes composing these authority relationships to arbitrary depth; thus, CapaFS does not have property E.

5.4.6 F: Access Controlled Delegation Channels

CapaFS does not have access controlled delegation channels, due to the following observation, which is relevant to CAK systems generally.

Remark 5.4 *For any CAK-like system, the analogy of a key and lock may be employed. A* key *(capability) can be copied and passed to any subject and that subject can then use it to unlock the designated resource. It would not matter that the subject holding the key has no immediate connection to its original owner. Receiving a key does not require any special relationship with the resource owner or the original owner of the key. Access decisions are simply a matter of whether the key unlocks the door. Under these systems, a capability can be passed at will by any subject possessing it.*

In CapaFS, any subject who presents a valid CFN to the server has the authority specified by that capability. Additionally, a CFN can be freely communicated between subjects, over any medium. Thus, CapaFS does not have property F, *access controlled delegation channels*.

5.4.7 G: Dynamic Resource Creation

This property is exhibited in CapaFS, albeit in a diminished sense. New resources cannot be created on a server by remote clients. However, each client is free to start their own local server and create new resources, expressing access restrictions on the resource's accessibility, and distributing the associated CFN's to remote collaborators.

5.5 Maat

Maat extensively uses what it calls capabilities, but the manner in which they are aquired, held and employed is significantly different from traditional capability architecture. So much so, that most of the properties surveyed in Table 5.1 are not present in the system. While Maat is able to overcome some limitations of the CAK model, its implementation fails to attain several of the benefits of capabilities, as discussed later in Section 5.9.

5.5.1 A: No Designation without Authority

As stated in Section 4.4, a client must request a capability for a resource from the MDS. It does this by sending a message to the MDS, calling *open(path, mode)* for the resource [60, Fig. 3]. The fact that the resource has a path and file name separate from the mechanism conveying authority to the resource, establishes that Maat has designation without authority and, therefore, lacks property A.

5.5.2 B: Dynamic Subject Creation

In Maat, a subject can be created, but it cannot be passed a subset of the parent process's authorities: authorized subjects are encoded in the capability [60, §3.3.1]. As a result, a new subject would not be able to use any of its parent's authorities. Instead, it must request access from the MDS, requiring an ACL check and a new capability to be created. Further, *extended capabilities* in Maat encode access to multiple resources and cannot be split up to give another process[6] a strict subset of those authorities [60, §3.3]. Thus, Maat does not support the creation of new subjects with less authority dynamically, instead it requires the intervention of the MDS to re-issue capabilities to new subjects.

5.5.3 C: Subject Aggregated Authority Management

Superficially, on a short time-scale, this system appears to be subject aggregated. However, the power to edit authorities is explicitly held at the MDS and managed as an ACL. When a subject wants access to an object it must request a capability from the MDS, passing a global identifier for the resource and a desired mode. The MDS will then use its ACL to grant or deny the subject a temporary, short-lived capability to the resource [60]. Subjects cannot manage the authorities handed to them without help from the MDS; that is, they cannot split up extended capabilities and cannot assign capabilities to new users. Effectively, authority to manage permissions is aggregated at the MDS and not at the subject.

[6]Maat does provide a *secure delegation* ability, which encodes short-term non-renewable leases to resources, for outsourcing computation. This is enabled by the *delegated capability*, essentially an entirely parallel, second-class capability system. This amounts to a secondary short-term lease that must be periodically renewed by the sponsoring node (the actual capability holder). These delegated-capabilities can be granted to any subject with no ACL check or determination of whether it is appropriate for that client to have that resource [60, §3.5]. The existence of delegated capabilities, however, does not satisfy the requirements to meet property B.

5.5.4 D: No Ambient Authority

Maat allows for the authority for many files to be encoded into a single extended capability [60, §3]. Extended capabilities act as blanket authority to access a set of resources using a single capability, ignoring whatever individual capabilities may otherwise be associated with each of those resources. Per Remark 5.2, this mechanism provides ambient authority, as it means a resource's individual capability does *not* have to be selected before opening each specific resource.

5.5.5 E: Composability of Authority

In order for networks of authority relationships to be composable, access to a resource must be equivalent to making a request to a subject, each requiring the invocation of a capability to the respective entity. In Maat, this is not the case. Resources are not equivalent to subjects. Resources are striped across OSDs and are accessed via passing capabilities to the OSD. There are no capabilities to subjects in Maat and thus subjects and resources are not composable.

5.5.6 F: Access Controlled Delegation Channels

Maat does not have access controlled delegation channels between subjects as there is no logic to restrict communication between client nodes. For both system capabilities and extended capabilities, however, the system *does* effectively restrict delegation by encoding authorized users into the capability itself. As a result, non-authorized parties who obtain a capability are unable to access the associated resources, because the OSD checks prior to allowing access.

5.5.7 G: Dynamic Resource Creation

Access restrictions can be specified on objects as they are created dynamically in Maat. As an object is created, its ACL (specifying the clients and their permissions) is specified and recorded at the MDS.

5.6 Tahoe-LAFS

Our investigation of Tahoe-LAFS extended beyond its documentation to include experimentation with an operational *Tahoe grid* of storage and gateway nodes, as well as examination of the open-source code. Our analysis allows us to affirm Wilcox-O'Hearn and Warner's assertion that Tahoe-LAFS follows a general CAK model.

5.6.1 A: No Designation without Authority

Tahoe stores one or more erasure-coded shares on a user-designated number of storage nodes. Each share has a designation of its own, the *storage index*, or *SI*. The SI is used on each storage node as the directory name under the path */.tahoe/shares/<SI>/*. Each share under the path is named by a share number. This SI is used to retrieve the file shares when a capability is invoked at a gateway. The client iterates through each storage node requesting a share matching the SI derived from the capability. Shares are regularly designated separate from the authority to access the resource (i.e., reconstruct it from its shares); thus, strictly, Tahoe does not have property A.

5.6.2 B: Dynamic Subject Creation

Recall from Section 3.2.2 that property B asks: *does the system allow a subject to dynamically create a new subject, granting it only a subset of the parent subject's authority?* Tahoe subjects include processes running on the gateway, processes running on a trusted local host behind the gateway and processes running on a remote client. In each of these cases, it is possible to dynamically create new subjects and pass to these a subset of authorities. This property is especially supported when considering remote processes: subjects on local hosts can pass to remote subjects a subset of the capabilities in their C-list.

One exception must be noted with regard to the originating gateway for a resource, that is, the gateway which first uploaded the resource to the grid. The capabilities to these resources are stored in a plaintext file reachable by any subject started under the same identity as the Tahoe server daemon. In particular, a subject under the same identity can perform a "get" request on any known Tahoe path and file name to retrieve the plaintext file without invoking the corresponding capability. Furthermore, when the command *tahoe manifest* is invoked on a gateway the *tahoe* utility returns a list of all the capabilities for resources uploaded from that gateway. This poses a problem for passing only a subset of authorities to subjects created on the gateway.

To limit the authority of local subjects at or behind the originating gateway, Remark 5.1 can be used to ameliorate the above exception and enable property B to be met. Without such precautions, however, property B would not necessarily be preserved for newly created local subjects.

5.6.3 C: Subject Aggregated Authority Management

In Tahoe, a subject can manage its authority either by adding a capability to their list upon creation of a new resource, adding a capability to their list upon receiving it from a remote subject, or deleting an existing capability from their list. Thus, property C is present in Tahoe.

5.6.4 D: No Ambient Authority

In the context of the remarks from Section 5.6.2 for controlling local subjects, property D is provided by Tahoe. For a subject to access a resource, it must specifically select and invoke the associated capability from the list of capabilities it possesses. This can be executed from the command line of a gateway node with the *get <capability>* request.

Organizationally, a Tahoe directory holds the capabilities to the directories and files it contains. This, however, does not constitute the type of implicit authority described in Remark 5.2. Rather, each file's capability must be individually invoked to gain access to the files contained in the directory.

5.6.5 E: Composability of Authority

In Tahoe, as in CAK models and following Remark 5.3, subjects and objects cannot be composed to any depth. Specifically the invocation of a capability to a resource is not similar to making a request via a capability to another subject. In particular, there are no capabilities to subjects, only capabilities for access to resources like files. The invocation of a capability is always the same: a *get* request for the capability string is processed by the gateway node. The gateway retrieves the associated shares from the grid, recombines the shares, to recover the plaintext file, returning it to the requestor.

5.6.6 F: Access Controlled Delegation Channels

As for any CAK system and as described in Remark 5.4, there exists no mechanism in Tahoe for controlling how capabilities are shared. No capability is required to communicate to another subject. No method is provided to limit sharing a capability.

5.6.7 G: Dynamic Resource Creation

Subjects in Tahoe are free to create either a mutable file, which can be either read or written, or an immutable file, which can only be read. For mutable files, subjects are free to distribute read-only-caps, read-write-caps or verify-caps to other subjects. Thus, property G is present in this system.

5.7 DisCFS

DisCFS makes some significant improvements over CapaFS, especially in the flexibility of its capability implementation, the *credential*. Despite this, its properties from Table 5.1 exactly parallel those of CapaFS, and closely approximate CAK, the only difference being the absence of property D.

5.7.1 A: No Designation without Authority

Resources specified in credentials are referenced by *handles*. These handles designate the resource to be fetched on the server, but convey no authority themselves. Additionally, the file name on the server is preserved in the comment section of the credential [67].

5.7.2 B: Dynamic Subject Creation

With respect to subjects dynamically created on remote clients, DisCFS has property B. A subject can create a new subject and pass to it a subset of its authorities. When subjects share the same client machine, the observations from Remark 5.1 apply, to limit the authority shared with new local subjects.

Overall, DisCFS demonstrates improvements over previous DFS systems we survey exhibiting property B. For instance, unlike CapaFS, DisCFS allows a subject to diminish existing capabilities. This can be done in two ways. First, a subject may reduce permissions *rwx* associated with a credential. Second, a subject may split a credential encoding authority to multiple resources into individual authorities. As every file on the server has its own handle, it is possible in DisCFS to reduce a credential for a directory to one for a specific file within that directory. This flexibility makes it possible for a subject whose C-list holds a single directory credential to pass to other subjects some diminished authorities for a subset of those resources.

5.7.3 C: Subject Aggregated Authority Management

In DisCFS, the subject is able to discard, create or even narrow the authorities it owns via modifying its credentials. Subjects can pass authorities to and receive them from other subjects, according to the policy set within those credentials. Thus, authorities are aggregated at and managed by the subject.

5.7.4 D: No Ambient Authority

Despite the ability to narrow credentials to individual files and diminish the permissions associated with credentials, discussed earlier in Section 5.7.2, DisCFS does not support property D. Remark 5.2 applies here, resulting in the absence of this property. Since a single credential (capability) can encode authority to access many separate resources, the bundled authorities constitute ambient authority when invoked.

5.7.5 E: Composability of Authority

As with other CAK-like systems, the argument from Remark 5.3 applies equally to DisCFS: the relationship between subject and resource is uni-directional and does not allow composability of authorities to arbitrary depth, as described by Miller *et al.* [18].

5.7.6 F: Access Controlled Delegation Channels

In DisCFS, no prior access relationship needs to exist between subjects for one to pass an authority to another. The rationale follows from Remark 5.4: a subject may delegate capabilities to any other subject. It is worth noting that the process to delegate a credential in DisCFS requires more actions than in other systems. A subject must copy the credential, (possibly) reduce its authorities, then add the recipient's public key to the credential and sign it, effectively delegating these authorities to the new subject. Thus, compared to other CAK systems, merely observing a capability does not grant the subject authority to use its associated resource.

5.7.7 G: Dynamic Resource Creation

DisCFS enables dynamic resource creation. It allows new resources to be created freely on the server as authorized by the policy in the credential that is being exercised. Access restrictions can be expressed as needed in the resultant credential to the resource.

5.8 Neo

The final system we survey is also the most recently developed. Neo incorporates many features expressed in CapaFS and DisCFS, providing more policy functionality and flexibility than either. Its properties from Table 5.1 resemble those of a CAK model, with the exception of lacking property D.

5.8.1 A: No Designation without Authority

In Neo, the *chit* conveys authority to use a resource. Each chit must specify within it the path for the server to follow to retrieve the referenced resource [45]. The existence of the path constitutes a method to designate a resource without authority, so Neo does not exhibit property A.

5.8.2 B: Dynamic Subject Creation

Neo exhibits property B. Neo also features the options supported by DisCFS (described in Section 5.7.2). In particular, a capability to a directory may be reduced to allow access to only a single file, or may be diminished from read-write-execute to read-only. As with the other distributed systems, Neo suffers from the same issues related to controlling new local subjects, described in Remark 5.1.

5.8.3 C: Subject Aggregated Authority Management

In Neo, a subject is able to discard, create, narrow or receive new authorities from other subjects through chits (capabilities). Chits are held and managed by subjects, so the system exhibits property C.

5.8.4 D: No Ambient Authority

Neo does not exhibit this property in general, for the same reasons documented in Remark 5.2. In particular, Neo allows authorities to many resources to be encoded into a single chit that, when invoked, implicitly associates all these authorities to the subject—even when the full set of authorities is not required by that subject (i.e., ambient authority).[7]

[7] As with our comments for DisCFS and CapaFS, Neo *could* exhibit this property by limiting each chit to a single file object.

5.8.5 E: Composability of Authority

Just as in other CAK-like systems, the reasoning in Remark 5.3 applies equally to Neo and property E is not present. In particular, resources and subjects are not equivalent, and accessing a resource is not functionally equivalent to requesting a service from another subject.

5.8.6 F: Access Controlled Delegation Channels

In Neo, a subject can delegate a chit to any other subject. No capability is required between subjects to enable this. The prior observations in Remark 5.4 apply equally to Neo and, thus, the system does not satisfy property F.

The above notwithstanding, Neo allows subjects to introduce policies into delegated chits that disable delegation further. This policy is sufficiently expressive to also limit the number of generations that can be further delegated, such as limiting delegation to no more than two degrees of separation from the original resource creator.[8]

5.8.7 G: Dynamic Resource Creation

Neo supports dynamic resource creation. Authority to each resource is expressed through chits authorizing subjects to use a resource.

5.9 Emergent Property Trends

In this section, we investigate properties Miller *et al.* attribute to consequences of the primary properties surveyed earlier. We call these emergent properties: revocation, confinement, confused deputy, and least privilege. In the sections that follow, we discuss the (degree of) presence or absence of each emergent property. The results are summarized in Table 5.2.

5.9.1 Revocation

DFSs in general appear to run into some issues with regard to revocation, for example:

- How does one revoke copies of data cached at a remote client?
- How does one revoke all replicas stored across the DFS?

[8]A *kind* of access controlled delegation would be enforced if Neo's design required (a) only authenticated chits may be passed, and (b) public keys are limited to a predetermined whitelist of authorized subjects.

	Obj Cap DFS	Caps as Keys	SFS (EROS)	DOI	CapaFS	Maat	Tahoe-LAFS	DisCFS	Neo
Irrevocability Myth	F	T	F	F	T	F	T	T	F
Confinement Myth	F	T	F	F	T	F	T	T	T
Confused Deputy	Best	Better	Best	Danger	Danger	Danger	Better	Danger	Danger
Least Privilege	Better	Better	Better	N	Unlikely	N	Better	Better	Better

Table 5.2: Capability-based File System Emergent Properties

- How does one revoke data that has been copied into new objects in the DFS?

None of the distributed systems we observe fully address these problems. Recall from Section 3.3.1 that the type of revocation our survey considers is *selective revocation*. Having granted an authority to a subject, selective revocation allows the granting subject to revoke that authority without affecting other user's access to that object. The focus is on revocation of authority to a specific instance of data, and not revocation of all copies of that data. For some systems, separate authority may allow access to the same data by the same user, even after one authority held by that user is revoked; this is allowable and can obey selective revocation. The systems we observe fall into one of the following groups with regard to their support for revocation:

1. Revocation through resource deletion: CapaFS, Tahoe-LAFS, DisCFS
2. Selective revocation through expiring capabilities and Blacklists: Neo
3. Selective revocation through revocable forwarders: SFS
4. Selective revocation through access control list management: DOI
5. Selective revocation through a combination of these methods: Maat

5.9.1.1 Capability Deletion and Redistribution

This group of systems correspond to those offering only a trivial form, that is, one provided any system: deletion of the capability, followed by redistribution[9] of a new capability for the resource. This sense of *revocation* is specifically *not* the kind considered by Miller *et al.* and we believe it to be particularly inflexible and impractical in large distributed systems.

CapaFS proffers to support only this kind of revocation, that is, complete revocation for

[9]Redistribution in these cases is not a trivial task since, in many capability systems, the knowledge of which subjects hold a capability is not visible globally.

all users. Regan and Jensen suggest implementing revocation in CapaFS by keeping a list of revoked Capability File Names on the server and checking the list prior to responding to a CFN request [55, §3.1.5]. This would revoke access to the resource for *all* users, invalidating the CFN. The method could also slow every CFN invocation, as the list is searched for a match prior to fulfilling a request. As the blacklist increases, this per-request overhead would increase as well. This suggestion would also encounter those challenges associated with redistributing a new capability to subjects who should retain access to the resource.

Tahoe-LAFS makes no claims to support revocation. Subjects can effectively accomplish the same type of revocation accomplished by deleting the resource from the server entirely, making a small change[10] to it, then re-uploading the resource. The resource will necessarily have a new capability which, at that point, can be redistributed to all subjects requiring access.

DisCFS states that it supports revocation through invalidating the original access credential by changing the resource's handle. This removes access to the resource for all users with credentials referencing that handle. Miltchev *et al.* suggest blacklisting user keys as another option for revocation [67, §4.4]. This would instead remove a user's access to all of his files, not just a particular resource. Disallowing a user's key would have the additional unintended consequence of removing access for any users who have credentials derived from that user's credentials, even for other files. In summary, none of the existing suggestions to support revocation under DisCFS provide selective revocation.

5.9.1.2 Blacklists and Expiration

This grouping of systems achieves selective revocation by blacklisting capabilities. This is made practical through the use of regular capability expiration. In Neo, chits have defined expiration times. Subjects can revoke any chit descended from a chit they hold by issuing a *revocation certificate* to the server. This adds the *revokes* tag into the XML of the parent chit (or higher), followed by the label of the chit targeted for revocation. The revocation

[10]This change is necessary as identical immutable files uploaded from the same gateway will converge to the same key. This is because the process is deterministic and designed to prevent storage of the same file twice. Thus, deleting the file then re-uploading it would result in the same capability string that was in use before. Changing the file, even as small a change as an added space, would prevent this convergence.

certificate is sent to the server, which verifies its authenticity and stores it [45, §5.5]. In the future, when the revoked chit is inspected for validity at the server, the delegation chain will not appear valid due to the revocation certificate, and the access will be denied. When the revoked chit expires, the revocation certificate expires as well and is garbage-collected. This prevents extensive use of server resources to maintain blacklists of revoked chits.[11] This mechanism provides fine-grained revocation of authority in Neo. Relatedly, it should be noted that DisCFS *could* achieve selective revocation through adopting the same method employed by Neo.

5.9.1.3 Revocable Forwarders

This group of systems implements selective revocation through the mechanism known as *revocable forwarders*. Revocable forwarders are a conceptual mechanism conceived by Redell in the context of capabilities in his 1974 dissertation. In it, he called the mechanism a *caretaker* [46, §2.3]. Graham and Denning actually presented the same idea in a slightly more general context two years earlier in their well-known paper, *Protection: Principles and Practice* [69]. The capability mechanism requires that the system interface between a subject and a resource be the same as the interface between a subject and another subject; that is, accessing a resource is functionally equivalent to sending a request to another subject. This enables a subject, say process Alice, to pass an authority to access a resource, say a database, to another subject, process Bob, yet retain the ability to revoke that authority in the future. Alice, instead of issuing a direct capability to the database, creates a new subject process, Carol, to which is given the direct capability to the database. Alice then issues a capability to access process Carol to Bob. This way when Bob invokes his capability to Carol, Carol provides Bob access to the database. At a later time Alice can revoke Bob's access by retiring process Carol [18].

EROS's SFS can accomplish selective revocation in the same manner, inserting directory processes in place of Alice, Bob, and Carol, and a file process in place of the database. This mechanism provides fine-grained *immediate revocation* in whatever system it is utilized.

[11]What is not clear in Neo, is the mechanism for renewal of chits before their expiration. One relatively easy solution is for the server to implement a daemon that looks through revocation certificates and if none apply to a certain chit approaching expiration, reissue that chit to its owner with the same paths, and new expiration date. This is similar to a mechanism implemented by Maat for capability renewal [60].

5.9.1.4 Access Control Lists

These systems can revoke access for a subject by altering the ACL associated with the resource. DOI operates in this way, requiring remote subjects authenticate themselves and be checked against the ACL. Access to the actual resource held in the object repository can be protected in a similar manner, requiring authentication using PKI [52].

5.9.1.5 Combination

Maat's implementation of revocation essentially comes down to an access control list managed at the MDS. The MDS makes all determinations as to whether it replies to an open() call with a capability or not. This is the long-term strategy for revocation on Maat.

When a capability has already been issued for a resource to a subject that is no longer trusted, revocation is performed by updating the ACL on the MDS. The MDS will then do one of several things depending on the value of the data. In Maat, capability expirations are very short, a maximum of five minutes. *Automatic revocation* occurs when the server does not renew the capability and it is allowed to expire. The designers comment that this seems sufficient for protecting some low-value resources.

For data that is of higher value, Maat allows subjects an alternative method: *immediate revocation*. OSDs are able to keep blacklists of capabilities that are revoked. In the event a capability is revoked, the MDS sends out a revocation message to all of the OSDs on the network. While called immediate, this process takes time to propagate and occurs at the speed of the network [60]. As in Neo (see Section 5.9.1.2), the blacklist is kept small in Maat since capabilities regularly expire.

5.9.2 Confinement

Recall from Section 3.3.2 that *confinement* is the property of a system such that when an untrusted program is executed on it, the system does not permit that program to leak information to untrusted parties. We found confinement related issues that no existing distributed systems address, such as the following:

- Copying content to new resources and delegating those further than the original resource is allowed.
- Sending resources to subjects outside the system.

- Leakage of secret key material allows new subjects to access related resources when the system is dependent on keys.

In our survey, we observe five approaches used by systems for confinement:

1. Confinement via access controlled delegation channels: SFS
2. No confinement but limited delegation: Neo, DisCFS
3. Trivial confinement via tokens that convey no authority: DOI
4. Trivial confinement via tokens that are non transferable: Maat
5. No confinement support, that is, capabilities can be delegated arbitrarily: CapaFS, Tahoe-LAFS, Neo (unauthenticated)

5.9.2.1 Access Controlled Delegation Channels

SFS is the only system surveyed that belongs to this group. SFS limits the authorities that a subject can pass to its child to a strict subset of its own authorities, and it constrains passing authorities to those subjects for which it holds a capability. Thus, it follows that a new subject process cannot communicate outside of the channels explicitly given to it on creation. "Confinement of authorities within a set of objects can be determined... by observing that the subgraph containing the set of objects is not connected to the rest of the object graph" [18, p. 5]. In other words, if capabilities to outside objects are not explicitly given to a child process, there is no way for it to communicate with those outside processes. This is the method by which confinement is achieved in SFS.

We see this group and type of system as embodying the strongest implementation of confinement, as no action on the part of an untrusted subject can result in undermining its confinement. Though other systems that we survey do claim to enable confinement of some form, we believe them to implement strictly weaker notions of confinement.

5.9.2.2 Limited Delegation

DisCFS and Neo allow a subject to disable delegation for capabilities that they pass to other subjects. When capabilities in these systems are delegated, their authority can be *narrowed* such that the receiving subject cannot delegate them any further. We describe the details of how both systems implement this approach.

DisCFS implements a mechanism to limit delegation with certain policy settings in the credential. Recall from Section 4.6 that DisCFS enables subjects to delegate authority to access resources to other subjects by appending the recipient's public key to a credential chain and signing this chain. DisCFS checks these delegation chains on access requests to grant access to the new subjects. This allows the issuer of credentials in DisCFS the freedom to implement a number of different delegation policies. The issuer can disable delegation entirely, or restrict the number of credential (capability) delegations that will be honored. Both policies limit which subjects have authority to access the file, implementing some form of delegation control.

Neo supports *authenticated chits* and *unauthenticated chits*. Unauthenticated chits can be derived from the master chit, or other unauthenticated chits. These are very similar to the type of capabilities issued in CapaFS and Tahoe-LAFS, in that mere knowledge of them constitutes delegation of their associated authority. Therefore, unauthenticated chits must be passed over protected channels only. Thus, it follows that unmitigated delegation of unauthenticated chits cannot necessarily be prevented [45]. Neo's authenticated chits, however, support a mechanism called *nodelegate*. Keleher *et al.* state that Neo supports confinement by using the no-delegate option in authenticated chits [45, §3]. Signed certificate chains allow the system to either disable or limit delegation to other subjects. Instead of signing the chain as in DisCFS, a subject delegates a Neo chit by signing both the subject's public key and the chit *fingerprint*, that is, a hash chain that allows the server to verify that no tampering has taken place [45, S5.4].

We observe that the limited delegation mechanism in DisCFS and the nodelegate mechanism in Neo can each be undermined by an malicious subject in the same way. In DisCFS, an untrusted subject could undermine limited delegation by sharing its credential and *leaking* its private key to other subjects. In much the same way, in Neo, a subject can undermine the nodelegate mechanism by leaking its key as it passes the chit to other subjects. This undermines Neo's confinement implementation.

5.9.2.3 Tokens Convey No Authority

In DOI, handles neither carry authority nor independently provide access. Thus, their propagation does not need to be *confined*. Instead, authority to access a resource is managed by the ACLs for the DOI held by the MDS, and by the systems holding the corresponding object.

5.9.2.4 Tokens are Nontransferable

In Maat, both capabilities and extended capabilities encode the identity of authorized users directly into the capabilities, using a Merkle tree. As a result, no unauthorized user is able to access the resource designated by the capability. Thus, propagation of authorities in Maat is not possible beyond those users defined in the ACL that is encoded in the capability.

5.9.2.5 No Confinement

In this group of systems, capabilities are implemented much like private keys: they can be shared arbitrarily, and used by anyone without limit. Tahoe's capabilities, CapaFS's capability file names (CFNs) and Neo's unauthenticated chits fall into this category.

Tahoe's capabilities may be freely communicated and delegated. They can be invoked by any subject capable of communicating with the associated Tahoe grid, which is designed to be implemented over an untrusted network such as the Internet. Thus, Tahoe does not enable any mechanism to achieve delegation control or confinement.

CapaFS's CFNs can be propagated to any subject over any communication channel and subsequently used to access the designated resource. It provides no method to prevent uncontrolled delegation or enforce confinement [54]. Regan and Jensen, however, propose extending CapaFS by adding client authentication through the use of public keys, and certificate chaining for delegation [55, §3.4]. They suggest this extension can restrict delegation according to a variety of policies. It would enable the server to limit users to a predetermined whitelist, akin to an ACL, to limit the right to delegate, or to prevent delegation past the intended recipient altogether. This approach is similar to the one described in Section 5.9.2.2, employed by DisCFS.

5.9.3 Confused Deputy

Recall from Section 3.3.3 that a *confused deputy* refers to a phenomenon in systems where a program is "tricked" into performing a transfer of authority which may lead to misuse of that authority. In our survey, we observe three groups of systems with regard to the extent to which a confused deputy may be implemented:

1. Easily avoided confused deputies: Tahoe-LAFS, SFS

2. Confused deputies from capabilities encoding authority to multiple resources: CapaFS, DisCFS, Neo, Maat
3. Confused deputies from the ambient authority associated with user IDs and ACLs: DOI, Maat

5.9.3.1 Easily Avoided Confused Deputies

This group of systems facilitate avoiding confused deputies by requiring subjects to select a single authority before an access can take place. Every request for access arrives with the associated authority necessary to grant that access.

In SFS, as in any object capability system, authority and designation are inseparable. Thus, programs cannot be confused by designation in the absence of the authority to access a resource. The (non-ambient) authority to perform some service always arrives in-context with the request to do it. For these systems, Miller *et al.* argue it is easier to avoid building confused deputies [18]; following this logic we categorize SFS with respect to confused deputy avoidance as *best*.

In Tahoe-LAFS, a capability to a resource must be explicitly selected in order to invoke the authority it wields. While a method to designate a resource share exists separate from the authority to invoke the resource, we find this designator adds little avenue for "confusion." These designators may allow shares to be retrieved, but the underlying resource must be recovered through the use of the capability, which acts as a decryption key after shares are combined. The net effect is similar to that of SFS: it is easier to avoid building a confused deputy in this system. Following the logic of Miller *et al.* for CAK models, we have categorized Tahoe as *better*.

5.9.3.2 Capabilities Encoding Multiple Resources

This group of systems support capabilities encoding the authority to access multiple resources, where a secondary designation must be provided to differentiate an access. This accommodation provides a significant risk of a confused deputy. CapaFS, DisCFS, and Neo each provide mechanisms that open this possibility because they allow multiple authorities to be encoded into a single capability, for example, access authority to all the files and subdirectories of a directory. This fact, coupled with the existence of authority-less forms of designation, mean that authority and designation travel different paths in arriving to a subject. If subjects are not careful to associate each capability they receive with the context of the request in which it was provided, a confused deputy can result.[12] As described, these systems meet what Miller *et al.* characterize as *danger*, with respect to confused deputies.

In Maat, the first problem we see with respect to confused deputies is its *extended capabilities*. They muddy the waters in a similar way as the capabilities encoding multiple resources in CapaFS, DisCFS, and Neo. Extended capabilities grant heuristically-determined multiple authorities to a large set of subjects, where no context is required to perform an access to the associated resources. The motivation for extended capabilities is to reduce the number of requests for capabilities to the MDS; this loose, non-context-associated handling of capabilities is intentional for performance reasons, but dangerous in the context of confused deputies.

5.9.3.3 Authority from User ID and ACLs

In this group of systems, a resource's designation is fully separated from authority to access the resource, which is managed through an access control list. All authority is tied to and derived from the user identity. A program's authority to perform some action, and the designation of the resource on which to perform that action, arrive separately. This makes the implementation of a confused deputy more likely, following the logic of Miller *et al.*.

The second way in which Maat is susceptible to confused deputies has to do with the way it associates authorities with a user ID. A request to open a file is compared to the ACL on the MDS and, depending on the subject's user identity, access is granted or denied. Likewise, in DOI, authority is tied to the identity of the subject presenting the handle to the system. The object's ACL determines whether a request to use a resource is granted or denied. Thus, following the logic of Miller *et al.* we have categorized DOI and Maat as *danger*.

5.9.4 Least Privilege

Recall from Section 3.3.4 that in order for a system to operate with *least privilege*, the minimum number of subjects must be granted the minimum number of authorities at the

[12]With each of these systems it should be noted that if a policy of ensuring only a single resource was encoded to any of their capabilities, the situation with respect to confused deputies could be significantly improved.

lowest permission level possible. In our survey, we found the following natural groupings of systems in regard to their support for least privilege operation:

1. Fine-grained authorities, exclusively: SFS, Tahoe-LAFS
2. Aggregate and separable authorities: DisCFS, Neo
3. Aggregate and inseparable authorities: CapaFS
4. Authority associated with a user: DOI, Maat

5.9.4.1 Exclusively Fine-grained Capabilities

Systems in this group mandate that only one resource be referenced by a capability at a time. Separate capabilities encode the type of access authority for the resource (i.e., read-write or read-only). These traits allow systems like SFS and Tahoe to truly operate in least privilege fashion. Every subject receives only the authority it needs to complete its required tasks, no more. We categorize these systems as *better*.

5.9.4.2 Coarse-grained Capabilities that Support Separation

As mentioned in Section 5.9.3.2, some systems allow authorities for multiple resources to be encoded in a single capability. This would seem to conflict with least privilege when a subject only needs a subset of those authorities. DisCFS and Neo, however, allow subjects to separate capabilities from those aggregate forms. These systems, then, are conducive with least privilege operation, as long as subjects separate unneeded authorities before passing to them other subjects. We categorize these systems as equal to those in the previous group, *better*.

5.9.4.3 Coarse-grained Capabilities that do not Support Separation

CapaFS encodes multiple resources into a single capability file name (CFN) in a way that cannot be separated or diminished by subjects. This conflicts with least privilege as it would necessitate a parent subject pass more authority to a child than may be necessary to operate. Additionally, CapaFS does not allow a parent to diminish those authorities associated with a CFN. This too prevents least privilege operation. Thus, we categorize CapaFS as less desirable (or *unlikely*) with respect to least privilege operation when compared to DisCFS, Neo, Tahoe and SFS.

5.9.4.4 Those with User Associated Authority

Subjects in DOI and Maat operate with the authority provided by the identity under which they are operating. DOI does not follow least privilege because subjects execute with the ambient authority granted by their identity (environment), not the smallest subset of privileges that would enable them to execute their assigned tasks. The result is that subjects may run with more authority than they need to perform required tasks. In Maat, a subject's rights and privileges are also determined by the identity under which a subject is operating. If that identity is listed as an authorized user on the MDS's ACL for a resource, it will get access to that resource; therefore, a subject's authorities depend on the environment of the user executing it. As a result, there is no way to enforce fine-grained authorities and limit the authority extended to an instance of a process. Following the logic of Miller *et al.*, we find that both DOI and Maat are in the category *infeasible* for least-privilege operation.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Discussion

In this chapter, we discuss the implications of our findings, compare some system approaches to the emergent properties, and address the value and predictive nature of the metric provided by Miller *et al.* with respect to the emergent properties. Additionally, we identify differences between various system designer's interpretations of revocation and confinement. Finally, we make some suggestions for topics that should garner more attention in future distributed file system implementations.

6.1 Revocation

According to Miller *et al.*, the presence or absence of property B and property E have consequences on whether one can expect revocation to be possible in a system. Specifically, they imply: $B \wedge \overline{E} \rightarrow \overline{Rev}$. Therefore, we would expect any system that exhibited property B, dynamic resource creation, but lacked property E, composability of authority, to be unable to provide for selective revocation of authorities. By cross-referencing our findings in Table 5.1 and Table 5.2 one can observe that, in fact, one system, Neo, demonstrates that it is capable of supporting selective revocation despite having this distribution of the Miller *et al.* properties. Our findings suggest that the Miller *et al.* logic only applies to one kind of selective revocation implementation, revocable forwarders, not selective revocation as a whole. Therefore, we find that the correct logical proposition would be: $B \wedge \overline{E} \rightarrow \overline{Rev{-}Fwd}$.

Our observations suggest there are many ways to implement revocation in distributed systems, with widely varying secondary effects. We describe some of the varying implementations thereof.

1. *Immediate selective revocation*: Upon action taken, the resource is not available to a specific subject via this system [29].
    - Revocable forwarders where one process acts as the per-file caretaker, is one possible implementation.
    - Neo's revocation certificate, when the remote storage server is a single node, is

another example producing immediate selective revocation.

2. *Lazy selective revocation*: Capabilities expire if not renewed, as seen in Neo and Maat.
3. *Eventual selective revocation*: Knowledge of revocation will eventually propagate throughout the system at network speed to all storage nodes. This is the case for Neo when the storage server is spread over many nodes. What Maat calls immediate revocation is also an example of this.

6.2 Confinement

According to Miller *et al.*, the presence or absence of property B and property F have consequences with regard to whether a system exhibits confinement. Specifically, they imply: $B \wedge \overline{F} \rightarrow \overline{Conf}$. We would expect any system that exhibited property B, dynamic resource creation, but lacked property F, access controlled delegation channels, to be unable to provide confinement. Our survey findings support this logical proposition.

One system, Neo, had property B and lacked property F, yet claimed to support confinement [45]. However, our interpretation of confinement is not merely concerned with whether a rule-abiding program can propagate its authorities or information outside its granted sphere of operation. Rather, we believe confinement must restrict untrusted subjects like trojan horse programs. These programs may take resources to which they have access, and try to leak authorities to that data to other subjects [70]. It is clear that a rule-abiding subject in Neo could not delegate authorities to other subjects using the method of delegation defined by Neo. However, it is also clear that nothing prevents a subject from communicating its chit to another subject along with its own secret keys, thereby circumventing the nodelegate mechanism and allowing the new subject to use the original chit. We believe that Neo's claim to support confinement is not a contradiction of the logical proposition regarding confinement that Miller *et al.* describe for capability systems. Instead, Neo meets a strictly weaker notion of confinement, due to the fact that its confinement mechanism relies on trusting that subjects will not misbehave.

6.3 Least Privilege

Our observations with respect to least privilege agree with Miller *et al.*; however, we find that the categories of *better* and *infeasible* are not descriptive enough for some systems.

In particular, CapaFS supports both dynamic subject creation and dynamic resource creation, but *can* encode multiple inseparable authorities in its capabilities. This introduces the possibility of coarse granularity in authority passing, and thus can result in a subject not operating with least privilege. This point highlights a tension that will arise in large capability-based distributed file systems between the efficiency afforded by aggregate capabilities (Maat calls them extended capabilities) and the significant cost of losing least privilege operation.

6.4 Confused Deputy

Our findings also concur with Miller *et al.* with regard to each system's propensity for confused deputies. We found one distributed system, Tahoe, provides favorable properties helping prevent the implementation of confused deputies. We note that, of the distributed systems we surveyed, five of six discuss revocation in their documentation, four of six discuss confinement in some form (e.g., delegation control), and four of six discuss least privilege; however, *none* discuss or mention their system's propensity for or defense against confused deputies. This seems to be a large oversight in the security concerns of their creators given the importance of the problem.

The confused deputy problem captures the heart of what allows server processes and other programs to get hijacked [19], [71]. If there were no ambient authority in systems; that is, each program had access only to the authorities it needed to operate, and furthermore, every authority a process had was tied to the context with which it was to be used, then malware would be unable to hijack systems, escalate their privileges, and perform their malicious intent. At most, they could hijack only those small number of authorities for which the target program had a need.

6.5 Other Observations

It is interesting to note that Miller, Shapiro, and Yee's work is not about distributed systems. Our survey highlights that in a *distributed* capability based system, additional factors can play into its effectiveness with respect to various tasks, such as: storage node distribution, the presence of a reference monitor, the existence of centralized metadata store, and even the mechanism by which capabilities are delegated between subjects.

Additionally, in a distributed system, *accountability* as discussed in Section 3.1, becomes of much greater concern, as the number of subjects and variety of data increases. We believe it is natural to desire an audit trail for post-event forensic analysis and attribution. Neo has two separate mechanisms to enable logging, for either authenticated or unauthenticated chits. While authenticated chits provide the more reliable and desireable logging, unauthenticated chit logging provides value all the same.

In unauthenticated chits, every generation of key can be created with a tamper-proof tag that differentiates it from previous generations of the chit. This tag can be logged and associated with its rightful user, so that its user is flagged when it is used [45, §5.7]. This does not prevent others from using the chit, but it can provide a starting place for a leaked chit.

In Neo's authenticated chits, every capability provides the identities (public keys) of the subjects in the delegation chain as well as the invoking subject. This information is logged along with the referenced resource and other metadata about the access [45, §5.2]. This same type of mechanism could be easily implemented in DisCFS by recording the delegation chain used in each credential as it is invoked by a subject. No such mechanism is available in CapaFS, though a solution using public keys in a similar way to above is suggested by Regan and Jensen [55, §3.3]. Tahoe has a number of daemons it calls *gatherers*. Tahoe's gatherers are limited, however, in the kind of information they can record to at most a storage node, gateway node or share information. This falls short of the information that we would like for a proper audit trail, but it may be extensible to act as such [72].

While some designs seem incompatible, small modifications may enable accountability. Thus, it is an open question whether accountability is orthogonal or related to the properties of the capability systems as defined by Miller *et al.* [18].

CHAPTER 7: Future Work and Conclusion

The Navy is moving the entirety of its information systems and Big Data processing to the Cloud. There are still open questions regarding the security of the Cloud. Capability-based operating systems have many inherent security properties that we find compelling:

- Fine-grained, least privilege operation
- A mechanism for encapsulation and protection of process memory
- Unforgeable references, supporting strong resource protection
- High system reliability
- Subject-controlled authorities, no centralized management of permissions
- Controlled sharing/flexible secure sharing of data
- Resistant to program hijacking, manipulation by malware, and memory safety violations (confused deputy avoidance)
- Reduced insider threat
- No super-user (no root)

The properties previously observed in capability operating systems motivate us to critically review the properties of distributed file systems using capabilities, to understand the advantages and limits of capabilities in distributed designs. One of the aims of this thesis is to determine if a capability architecture could support the security objectives of a tactical cloud. We believe that a (yet to be described) design may, indeed, serve as such.

A primary challenge to this goal has been that capability systems vary widely in implementation; as a result, they vary in the security properties that they exhibit. Miller *et al.* argue this by categorizing capability systems into archetypes. Some capability systems exhibit selective revocation, others do not; the same is true for confinement, confused deputies (anti-hijacking), and least privilege. *Object capability systems* are a class of systems they identify that possess *all* of the desired emergent properties.

Our survey investigates six capability-based distributed file systems—CapaFS, DOI, Maat, DisCFS, Tahoe-LAFS, and Neo—and one local object-capability file system—EROS's

SFS—using the seven properties defined by Miller *et al.* Our survey extends their analysis into the domain of distributed file systems (DFS).

We find that none of the *distributed* systems surveyed exhibit all of the Miller *et al.* properties; to wit, we found no object-capability distributed file system. Additionally, we find that no distributed systems exhibit all desirable emergent properties: selective revocation, confinement, confused deputy resistance (anti-hijacking), and least privilege. We re-evaluated the relationships Miller *et al.* claim exist between their capability properties and the emergent properties, affirming some and refuting others. Our observations will be useful to the Navy because they highlight many tensions that must be considered when making decisions about future Big Data systems and access control. Capabilities must be further examined as an avenue toward the robust security enforcement the Navy requires for its future platforms; however, as we consider cloud-scale distributed capability systems, we will have to weigh these tradeoffs and prioritize those security properties motivating the use of capabilities in the first place.

Given the potential benefits of a DFS with the properties of an object capability system, and the fact that no distributed system exhibiting these properties yet exists, we believe it to be valuable future work to pursue the following:

- Develop a full reference model for an object capability distributed file system.
- Create a prototype aiming to achieve all of the emergent properties by merging two of the existing systems that each covered part of them and accountability (i.e., Tahoe-LAFS and Neo).
- Prototype and evaluate a simple object capability distributed file system using a distributed object capability framework such as *E* [73].

In summary, capability systems hold significant promise for solving many of our complex security problems in the Cloud. More research must be done to harness their benefits and create the highly-secure, scalable, capability-based distributed systems on which future cloud systems might be based.

List of References

[1] Deputy Chief of Naval Operations, Information Dominance, "Task force cloud charter," Jan. 2014, Department of the Navy memorandum N2N6/4U119014.

[2] I. R. Porche III, B. Wilson, E.-E. Johnson, S. Tierney, and E. Saltzman, "Data flood: Helping the Navy address the rising tide of sensor information," White Paper, RAND Corporation, 2014. [Online]. Available: http://www.rand.org/pubs/research_reports/RR315.html

[3] "Maritime ISR enterprise acquisition (MIEA) review," White Paper, Programs, Management, Analytics and Technologies Inc., Jan. 2011. [Online]. Available: http://goo.gl/H2cwsb

[4] V. Kundra, "Federal cloud computing strategy," White Paper, US Whitehouse, Feb. 2011. [Online]. Available: http://www.whitehouse.gov/sites/default/files/omb/assets/egov_docs/federal-cloud-computing-strategy.pdf

[5] K. L. Jackson and B. Gourley, "Cloud computing: Risks, benefits, and mission enhancement for the intelligence community," White Paper, Intelligence and National Security Alliance, Mar. 2012. [Online]. Available: http://www.insaonline.org/i/d/a/Resources/Cloud_Computing.aspx

[6] A. Corrin, "Navy struggles to streamline IT," *Defense News*, Apr. 10, 2014. [Online]. Available: http://goo.gl/XUUI1y

[7] "Cloud computing: benefits, risks and recommendations for information security," White Paper, European Network and Information Security Agency, 2009. [Online]. Available: http://www.enisa.europa.eu/activities/risk-management/files/deliverables/cloud-computing-risk-assessment/

[8] J. E. Barnes and S. Gorman, "U.S. says Iran hacked Navy computers," *The Wall Street Journal*, 27, 2013. [Online]. Available: http://online.wsj.com/news/articles/SB10001424052702304526204579101602356751772

[9] "APT1: Exposing one of China's cyber espionage units," Mandiant, Feb. 2013. [Online]. Available: http://www.mandiant.com/apt1

[10] "Global threat report: 2013 year in review," White Paper, CrowdStrike, Jan. 2014. [Online]. Available: http://www.crowdstrike.com/sites/all/themes/crowdstrike2/css/imgs/platform/CrowdStrike_Global_Threat_Report_2013.pdf

[11] J. Robertson and M. Riley, “JPMorgan, four other banks hit by hackers: U.S. official,” *Bloomberg.com*, Aug. 27, 2014. [Online]. Available: http://www.bloomberg.com/news/2014-08-27/customer-data-said-at-risk-for-jpmorgan-and-4-more-banks.html

[12] K. Zettr, “Network admin allegedly hacked Navy—while on an aircraft carrier,” *Wired*, May 7, 2014. [Online]. Available: http://www.wired.com/2014/05/navy-sysadmin-hacking/

[13] J. Vijayan, “Snowden serves up another lesson on insider threats,” *ComputerWorld*, Nov. 8, 2013. [Online]. Available: http://www.computerworld.com/article/2485759/cyberwarfare/snowden-serves-up-another-lesson-on-insider-threats.html

[14] P. Buneman and S. B. Davidson, “Data provenance—the foundation of data quality,” White Paper, 2010. [Online]. Available: http://www.sei.cmu.edu/measurement/research/upload/Davidson.pdf

[15] “NIST big data interoperability framework: Volume 4, security and privacy requirements,” NIST Big Data Public Working Group, Apr. 23, 2014, draft, version 1.

[16] J. B. Dennis and E. C. Van Horn, “Programming semantics for multiprogrammed computations,” *Commun. ACM*, vol. 26, no. 1, pp. 29–35, Jan. 1983. [Online]. Available: http://doi.acm.org/10.1145/357980.357993

[17] P. J. Denning, “Fault tolerant operating systems,” *ACM Computing Surveys*, vol. 8, no. 4, pp. 359–389, Dec. 1976.

[18] M. S. Miller, K.-P. Yee, and J. Shapiro, “Capability myths demolished,” Johns Hopkins University Systems Research Laboratory, Tech. Rep. SRL2003-02, 2003. [Online]. Available: http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf

[19] N. Hardy, “The confused deputy: (or why capabilities might have been invented),” *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, Oct. 1988.

[20] J. Saltzer and M. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[21] M. Wilkes and R. Needham, *The Cambridge CAP computer and its operating system*, ser. Operating and programming systems series, P. Denning, Ed. New York, NY: Elsevier North Holland, 1979.

[22] M. S. Miller, B. Tulloh, and J. S. Shapiro, “The structure of authority: Why security is not a separable concern,” in *Proceedings of the Second International Conference on Multiparadigm Programming in Mozart/Oz*, Charleroi, Belgium, 2005, pp. 2–20. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31845-3_2

[23] J. Shapiro and N. Hardy, “EROS: a principle-driven operating system from the ground up,” *IEEE Software*, vol. 19, no. 1, pp. 26–33, Jan. 2002.

[24] W. B. Ackerman and W. W. Plummer, “An implementation of a multiprocessing computer system,” in *Proceedings of the first ACM symposium on Operating System Principles*. ACM, 1967, pp. 5–1.

[25] H. M. Levy, *Capability-Based Computer Systems*. Newton, MA: Butterworth-Heinemann, 1984. [Online]. Available: http://homes.cs.washington.edu/~levy/capabook/

[26] “An interview with Peter J. Denning, oral history 423, conducted by Jeffrey R. Yost on 10 April 2013,” April 2013. [Online]. Available: http://hdl.handle.net/10945/37925

[27] M. Wilkes, *Time-Sharing Computer Systems*, ser. MacDonald computer monographs. American Elsevier Publishing Company, 1968.

[28] D. England, “Capability concept mechanisms and structure in System 250,” in *Proceedings of the International Workshop on Protection in Operating Systems*, 1974, pp. 63–82. [Online]. Available: http://www.cs.ucf.edu/courses/cop6614/fall2005/englandplessey250.pdf

[29] E. Cohen and D. Jefferson, “Protection in the Hydra operating system,” *ACM SIGOPS Operating Systems Review*, vol. 9, no. 5, pp. 141–160, Nov. 1975.

[30] A. Herbert, “The Cambridge CAP computer,” *Resurrection: The Bulletin of the Computer Conservation Society*, no. 63, pp. 12–24, Autumn 2013. [Online]. Available: http://www.cs.man.ac.uk/CCS/res/pdfs/res63.pdf

[31] M. E. Houdek, F. G. Soltis, and R. L. Hoffman, “IBM System/38 support for capability-based addressing,” in *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA’81)*, Minneapolis, MN, 1981, pp. 341–348.

[32] P. G. Neumann and R. J. Feiertag, “PSOS revisited,” in *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, Las Vegas, NV, Dec. 2003, pp. 208–216. [Online]. Available: http://dx.doi.org/10.1109/CSAC.2003.1254326

[33] A. C. Bomberger, N. Hardy, A. Peri, F. Charles, R. Landau, W. S. Frantz, J. S. Shapiro, and A. C. Hardy, “The KeyKOS nanokernel architecture,” in *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992.

[34] J. S. Shapiro, J. M. Smith, and D. J. Farber, “EROS: A fast capability system,” *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, pp. 170–185, Dec. 1999.

[35] J. S. Shapiro, "EROS: A capability system," Ph.D. dissertation, University of Pennsylvania, 1999.

[36] J. S. Shapiro, "Jonathan S. Shapiro," February 2009. [Online]. Available: http://www.coyotos.org/~shap/

[37] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "A taste of Capsicum: Practical capabilities for UNIX," *Communications of the ACM*, vol. 55, no. 3, pp. 97–104, Mar. 2012.

[38] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*, Minneapolis, MN, 2014, pp. 457–468.

[39] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an operating-system kernel," *Communications of the ACM*, vol. 53, no. 6, pp. 107–115, June 2010.

[40] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, D. Cock, and K. Gerwin, "seL4 enforces integrity," in *Proceedings of the 2nd International Conference on Interactive Theorem Proving*, Nijmegen, The Netherlands, August 2011, pp. 325–340. [Online]. Available: http://www.ssrg.nicta.com.au/publications/papers/Sewell_WGMAK.pdf

[41] G. Heiser and B. Leslie, "The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors," in *Proceedings of the First ACM Asia-pacific Workshop on Systems*, ser. APSys '10. New York, NY, USA: ACM, 2010, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/1851276.1851282

[42] M. Lentczner, "Belay cloud access protocol & api," May 2011. [Online]. Available: https://sites.google.com/site/belayresearchproject/bcap

[43] C. P. Pfleeger and S. L. Pfleeger, *Security in Computing*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 2006.

[44] P. G. Neumann, "Principled assuredly trustworthy composable architectures," SRI International, Menlo Park, CA, Tech. Rep. CDRL A001, Dec. 28, 2004. [Online]. Available: http://www.csl.sri.com/users/neumann/chats4.pdf

[45] P. Keleher, B. Bhattacharjee, N. Spring, V. Lekakis, and Y. Basagalar, “Capabilities and identity in a wide-area file system,” White Paper, 2014. [Online]. Available: http://414.kelehers.me/papers/chitfs.pdf

[46] D. D. Redell, “Naming and protection in extendible operating systems,” Massachusetts Institute of Technology, Project MAC, Cambridge, MA, Tech. Rep. MAC TR-140, 1974. [Online]. Available: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-140.pdf

[47] B. W. Lampson, “A note on the confinement problem,” *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, Oct. 1973.

[48] J. S. Shapiro and S. Weber, “Verifying the EROS confinement mechanism,” in *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000, pp. 166–176.

[49] J. S. Shapiro, D. J. Farber, and J. M. Smith, “State caching in the EROS kernel,” in *Proceedings of the 7th International Workshop on Persistent Object Systems*, Sep. 1996, pp. 88–100.

[50] R. Kahn and R. Wilensky, “A framework for distributed digital object services,” *International Journal on Digital Libraries*, vol. 6, no. 2, pp. 115–123, Apr. 2006. [Online]. Available: http://dx.doi.org/10.1007/s00799-005-0128-x

[51] P. J. Denning and R. E. Kahn, “The long quest for universal information access,” *Communications of the ACM*, vol. 53, no. 12, pp. 34–36, Dec. 2010.

[52] “Overview of the digital object architecture,” White Paper, Corporation for National Research Initiatives, July 28, 2012. [Online]. Available: http://www.cnri.reston.va.us/papers/OverviewDigitalObjectArchitecture.pdf

[53] S. Sun, L. Lannom, and B. Boesch, “Handle system overview,” RFC 3650, Nov. 2003. [Online]. Available: http://www.handle.net/rfc/rfc3650.html

[54] J. Regan, “CapaFS: A globally accessible file system,” master’s thesis, 1999. [Online]. Available: http://www.tara.tcd.ie/handle/2262/785

[55] J. T. Regan and C. D. Jensen, “Capability file names: Separating authorisation from user management in an internet file system,” in *Proceedings of the 10th Conference on USENIX Security Symposium (SSYM’01)*, Washington, D.C., 2001.

[56] S. Miltchev, J. M. Smith, V. Prevelakis, A. Keromytis, and S. Ioannidis, “Decentralized access control in distributed file systems,” *ACM Computing Surveys*, vol. 40, no. 3, Aug. 2008.

[57] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating Systems Design and Implementation*. USENIX, 2006, pp. 307–320.

[58] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004, pp. 53–62.

[59] O. Rodeh and A. Teperman, "zFS — a scalable distributed file system using object disks," in *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2003)*, Apr. 2003, pp. 207–218.

[60] A. Leung, E. Miller, and S. Jones, "Scalable security for petascale parallel file systems," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC'07)*, Nov. 2007, pp. 1–12.

[61] J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *Proceedings of the USENIX Winter Conference*, Feb. 1988, pp. 191–202.

[62] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: The least-authority filesystem," in *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability (StorageSS '08)*, Alexandria, VA, 2008, pp. 21–26. [Online]. Available: http://doi.acm.org/10.1145/1456469.1456474

[63] R. Paul, "P2P-like Tahoe filesystem offers secure storage in the cloud," *Ars Technica*, Aug. 4, 2009. [Online]. Available: http://arstechnica.com/information-technology/2009/08/p2p-like-tahoe-filesystem-offers-secure-storage-in-the-cloud/

[64] B. Byfield, "Hide cloud data from the cloud vendor: Secure storage with Tahoe-LAFS," *Linux Pro Magazine*, May 2014. [Online]. Available: http://www.linuxpromagazine.com/Online/Features/Hide-Cloud-Data-from-the-Cloud-Vendor

[65] Z. Wilcox-O'Hearn, "Hack Tahoe-LAFS," Aug. 2014. [Online]. Available: http://www.tahoe-lafs.org/hacktahoelafs/

[66] Z. Wilcox-O'Hearn, "Tahoe-LAFS," Aug. 2014. [Online]. Available: http://www.tahoe-lafs.org/

[67] S. Miltchev, V. Prevelakis, S. Ioannidis, J. Ioannidis, A. D. Keromytis, and J. M. Smith, "Secure and flexible global file sharing." in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2003, pp. 165–178.

[68] S. Sun, S. Reilly, and L. Lannom, “Handle system namespace and service definition,” RFC 3651, 2003. [Online]. Available: http://www.handle.net/rfc/rfc3651.html

[69] G. S. Graham and P. J. Denning, “Protection: Principles and practice,” in *Proceedings of the AFIPS ’72 Spring Joint Computer Conference*, Atlantic City, NJ, 1972, pp. 417–429. [Online]. Available: http://doi.acm.org/10.1145/1478873.1478928

[70] S. B. Lipner, “A comment on the confinement problem,” in *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP ’75)*, Austin, TX, 1975, pp. 192–196. [Online]. Available: http://doi.acm.org/10.1145/800213.806537

[71] C. Fournet and A. D. Gordon, “Stack inspection: Theory and variants,” *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 3, pp. 360–399, May 2003.

[72] Z. Wilcox-O’Hearn, “Tahoe-LAFS: source: trunk/docs/logging.rst,” Nov. 2013. [Online]. Available: https://tahoe-lafs.org/trac/tahoe-lafs/browser/docs/logging.rst#overview

[73] M. S. Miller, “Robust composition: Towards a unified approach to access control and concurrency control,” Ph.D. dissertation, Johns Hopkins University, Baltimore, MD, May 2006. [Online]. Available: http://www.erights.org/talks/thesis/

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California