Qualitative and Quantitative Qualities of Contemporary Capability Constructions

Kritika Iyer
Cheriton School of Computer Science
k4iyer@uwaterloo.ca

Sajin Sasy
Cheriton School of Computer Science
ssasy@uwaterloo.ca

Justin Tracey
Cheriton School of Computer Science
j3tracey@uwaterloo.ca

***Abstract*—Abstract text. Abstract text. Abstract text. Abstract text. Abstract text. Abstract text. Abstract text. Abstract text.**

I. Introduction

Capabilities are a compelling means of access control in digital systems. As such, they have seen much interest in computer science research, as well as usage in real systems ranging from distributed file systems to entire operating systems. However, as the design of existing and desirable systems changes, the design decisions with regards to access control must change as well. While research and implementations have to some extent addressed new problems and goals as they have arisen, these lines of work have primarily focused on their own utility, without much attention given to alternatives. In particular, there is a surprising lack of comparative study in the performance impact various design decisions have in existing access control solutions, and capabilities are no exception.

In order to decide how an access control system should be implemented, it would stand to reason that one must first have an understanding not only of the mechanisms that system would provide, but also the advantages and disadvantages of particular design decisions compared to other systems. Towards that end, we seek to examine the differences between various options of capability-based access control systems, both in practice, and in high-level design. To do so, we present in this paper an examination of some deployed capability systems, as well as a means of providing comparative performance results from a customizable workload.

In this section, we provide some background on access control, so as to provide a better understanding of the purpose and background of capability systems. Then, we give more detail on what capabilities specifically are, and what advantages they bring. In Section II, we give some specific examples and details on how capabilities have been used in kernels. Then, in Section III, we give examples of capabilities that are implemented in the user space level of systems. We then give a description of our experimentation on capability structures, followed by some concluding remarks.

A. Access Control

Sufficiently complicated systems must have some means of keeping track of resources. Who has access to what resource and under what conditions is a non-trivial problem that exists in a myriad of contexts, from management of physical locations to management of intelligence. Management of these resources, to some extent, can be used to analogously manage computer systems, but were generally insufficiently formalized to create the guarantees desired for systems like operating systems. To address this, precise models known as *access control systems* were created.

While details may vary, access control is primarily concerned with the interaction between two or three types of entities: an *object* (or *resource*), a *subject* (or *client*), and optionally, an *authority*. The object is the particular resource for which access is being determined by the access control system. Examples include a particular file in a file system, output from a sensor, and an address range in memory. The subject is the entity that is being granted or denied access. This may be anything from a human user, to a particular process or thread, or a physical device. The authority is the entity which is granting (or denying) the subject the ability to access the object. In practice, there typically exists some authority, such as the kernel or a user space program that manages the access control. However, strictly speaking, no authority is necessary, and often (particularly in theoretical constructions), any action an authority may implicitly be able to perform is instead represented as an object that another subject has access to (e.g. there may be a "grant access to object *o*" object).

The relationship between these entities may be represented as an *access matrix*.[3] In such a representation, the rows correspond to subjects, while the columns correspond to objects. No authority is explicitly represented in the matrix, but it would be responsible for populating or enforcing the rules of the matrix. Within each cell of the matrix are the actions a the respective subject may take on the respective object. (The original paper describes these cells as "capabilities," though this is not exactly how terminology used today.) An example of such a matrix may be found in table I. In it, the user Carol has read and execute permissions on File 2, and no permissions on File 3.

Because of the sparsity of the matrix, it is generally wasteful to store it in its entirety as a table. Instead, the matrix is typically stored as the columns of the table (e.g. in table I, there would exist an entry for File 1, which would store the fact that Alice has read and write permissions, and Carp; has read permissions), which is known as an *Access Control List*

	File 1	File 2	File 3
Alice	rw	rx	rwx
Bob		rx	
Carol	r	rx	

TABLE I
AN EXAMPLE ACCESS CONTROL TABLE.

(or ACL), or as the rows of the matrix (e.g. there exists an entry for Carol, which would store the fact that she has read permissions on File 1, and read and execute permissions on File 2), which is known as a capability-based system.

B. Capability-Based Access Control

Now that the purpose and general categorization of access control systems are understood, we will now describe some more details of the theoretical backing of capability-based systems, and what their advantages are over access control lists.

As previously stated, capability-based systems are a type of access control, in which the actions a subject may take with all objects that subject has been granted permission to interact with are stored. The structure in which this information is stored is referred to as a *capability*. In order for a subject to access an object, it must use a token that corresponds to the appropriate capability. What form this token takes, and how the capability itself is stored, varies depending on the implementation.

Strictly speaking, with an access control matrix that is fine-grained enough to represent all possible combinations of subject and object, and a set of explicitly defined rules to determine how the matrix is populated, there is no operation that can be performed in a system based on ACLs that could not be performed via capabilities, and vice versa. In practice, however, capabilities tend to be much more powerful and versatile. [5] To understand why this is, we must first understand how capability-based systems operate, as well as some of the limitations of ACLs.

Capability-based systems are typically based, at least loosely, on theoretical models that represent a set of capability rules. One of the most well known of these is the *take-grant* scheme, where authority over an object can be granted or denied according to specific rules.[4] These models are useful in that they can be used to prove that all potential capability distributions behave certain rules, in an efficient manner. For example, given that the state of all capabilities is N at this point in time, the system can prove in linear time that subject s will never be able to access data from object o, regardless of how the subjects use or grant their capabilities starting from N (so long as the rules of the system are followed, e.g. a subject can't grant a capability for an object to another subject when it wasn't supposed to). Using these theoretical frameworks makes reasoning about the security of systems and the guarantees they provide much simpler.

There are four general categories of capability systems in practice, differentiated by how capabilities and their tokens are represented: *tagged with tag bits*, *tagged with type system*, *segregated*, and *password* (or *sparse*). [6] As a baseline comparison, we also describe access control lists.

Access Control List (ACL): A list of all subjects with access to the object is stored. This typically takes the form of the object itself storing a list of subjects, and what actions that subject is and is not allowed to perform with the resource. This allows for operations like removing permissions for all subjects from the object to be relatively simple, while making other actions like removing all permissions for a particular subject to all objects very difficult.

An example of an ACL would be how typical Unix file permissions work. With each file, there is some metadata stored, containing which user owns a file, and which group is associated with a file. With each of these, the relevant permissions (read, write, or execute) are stored as well. This means it is relatively simple to change the permissions on a file to no longer allow anyone (or to allow everyone) to read it, but also means that changing the owner of all files of a particular user would require traversing the entire file system and identifying the files that user owns.

Tagged with tag bits: In these systems, the capability is stored within the actual token, while the object itself has no inherent knowledge of any capabilities. Every time a capability is used, the system must verify that the capability that was received with the token is valid (note that the verification could occur through the authority or the resource itself, though). If the system wished to revoke or change all capabilities for a particular object, it would require somehow finding every capability in the entire system, which is generally viewed as infeasible.

Tagged with type system: Similar to Tagged with tag bits, only the capability is represented as metadata structure that references the actual resource, as well as which components of the resource are additional capabilities. When a request is made for the resource, it instead requests the capability structure, which the system can then check and use to access the resource. This makes modification of capabilities referring to an object (via a capability metadata object) simpler. For example, if Alice, Bob, and Carol all have capabilities to the same object via the same capability metadata object, then modifying the capabilities of that one metadata object will modify the capabilities for all three (contrast this with the "Tagged with bits" method, where this would require modifying the capabilities of each user individually).

Segregated: All capabilities are stored in a protected region of memory, individual to each subject. Subjects can store pointers into these capabilities, but cannot access them directly, and must instead make system calls to modify them, which can enforce the access control policy. Capabilities can be shared, but only through the aforementioned system calls, which copy the capabilities to the respective region of memory for the target entity.

Password/sparse: Similar to segregated capabilities, but all subjects share one region of memory. While access and modification are still controlled by the access control policy implemented by the system, capabilities can be shared with

other subjects directly, by giving the address to where the capability is stored. To prevent malicious entities from brute-forcing all possible capabilities, a password field is commonly used as an additional requirement to gain access to the capability. Otherwise, the system would have no way of differentiating between capabilities that were legitimately shared, and those that were guessed.

While the majority of actual capability-based systems fall into one of these categories, it is conceivable that other structures may exist (at the very least, any of these categories could be modified to be made arbitrarily more complicated). What benefits and drawbacks these or other types of capability-based systems bring can and has been reasoned about to some extent, as described above, but the true limits and performance differences between them remain largely unmeasured.

II. Kernel Space Implementations

A. L4

L4 is a family of microkernels, originally based on the L3 microkernel. Much in the same way that Unix was originally a single OS but now often refers to a style or family of OSs, so is L4 both a single microkernel OS from 1994, and a wide family of kernels which are based on its design. [1]

Many of the advances in capability-based systems as applied to operating systems originate in the L4 family, originating from a shift around 2008 that created an emphasis in greater security and safety than alternative operating systems. Capability-based systems were added to the OKL4 kernel, and then to the Fiasco.OC kernel (both of which being in the L4 family) as their new primary access control mechanism, in attempt to provide greater security. However, due to a growing desire to create a formally verified OS, and the infeasibility of doing so without starting from a code base intended for it, an entirely new L4 kernel was written in 2013: seL4. With a kernel written in C ported from Haskell,[1] it is considered to be the first formally verified general-purpose OS. Integral to this formal verification is the capability system that it is built around.

Both due to the nature of being a microkernel, and to aid in the task of formal verification, seL4 makes every effort to minimize the services that are contained in the kernel. In the capability system that seL4 is built around, there are 6 fundamental categories of objects in the kernel: [2], [7]

1) **Capabilities:** Capabilities themselves are objects in the seL4 kernel. These objects are stored in what are called *CNodes*. When a CNode is created, it is given a fixed number of capability "slots" that store capabilities. As capabilities are objects, one capability a CNode may store in one of its slots is another CNode, forming a linked set of CNodes that is called the *CSpace*. For a thread (which, as described in more detail later, are the subjects of seL4's capability system) to have a certain capability on a particular object, that object must be stored somewhere in the thread's CSpace.
2) **Objects/memory:** All objects and memory start as what are called *Untyped Memory* capabilities, which can be thought of as a blank-slate for a potential capability. When a new object is created or memory is allocated, the Untyped Memory has the *retype* method invoked on it, which will create the desired object or allocate the desired memory (if granted). Conversely, *revoke* is used to remove previously granted capabilities.
3) **Virtual address space:** In a structure largely similar to that of CSpace, virtual memory addresses are managed in what is called a *VSpace*. How the VSpace is structured specifically depends on the processor architecture. In the case of IA32 and ARM architectures, the root of a VSpace is a *Page Directory* object, which points to *Page Table* objects. These Page Table objects then point to specific regions of physical memory, which are represented as *Frame* objects.
4) **Thread Control Block (TCB):** Threads are the subjects of the seL4 capability system. Each thread has a designated TCB object, which is used by the kernel for controlling the thread (e.g. scheduling). Most importantly for our purposes, the TCB is what designates the thread's CSpace and VSpace roots. When a thread wishes to use a capability, it presents the kernel the respective capability in its CSpace. As a result, seL4 uses a capability system that falls into the Segregated category, described in Section I.
5) **Endpoints (EP):** Inter-Process Communication (IPC) in seL4 is done using EP objects that facilitate message passing. These Endpoints can be either synchronous or asynchronous in nature, with the synchronous Endpoints being used for transferring data between threads, as well as capabilities if the sender has the *Grant* right on the EP. Asynchronous EPs are used primarily for simple notifications.
6) **Device I/O:** As a microkernel, device drivers run in user space and do not require a kernel object themselves. However, in order to provide access to lower-level hardware that the drivers may need, such as DMA address spaces, I/O ports, and interrupts, objects specific to these resources are available.

The structure of this capability system has been used not only to successfully engage in formal verification of the entire kernel, but it has also been used as the basis of at least one other OS: Barrelfish.

B. Barrelfish

Barrelfish is a research operating system built by ETH Zurich and Microsoft Research, Cambridge.[2] [3] The motivation for building this system from scratch is to provide an OS for multi-core, many-core systems. Their aim is to support the

[1] As an interesting aside, a Linux-compatible user space for seL4 is being built in Rust.

[2] http://www.barrelfish.org/

[3] http://research.microsoft.com/en-us/projects/barrelfish/

scalability and the heterogeneous architecture that is sought after in current systems.

A detailed review of the resource management for such a multi-core, heterogeneous system is provided in the works of Nevill et al. [6]. Nevill et al. analyse the capability management in such a system and specifically look at a single design for implementing it. If such systems have memory that is not shared or non-coherent memory, it requires that there be a separate kernel on each core in order to ensure that systems can manage and share resources. To do this there are various communication systems that it uses to synchronize the data between all the kernels and establish shared resources.

Barrelfish capability design is based on the design of L4, and it is an extension of L4 in a multi-kernel system. The capability system for L4 is in fact the design for Barrelfish on a single core. The capability types are hence an extended version on L4 as well. It uses a domain specific language to define these capability type, which can be found in the appendix in [6]. We now mention a few examples of how Barrelfish extends the L4 design.

1) **Main Memory:** The RAM type is the equivalent of L4's *Untyped Memory* and as in L4, it can be split into smaller parts.
2) **Page Tables:** Instead of having a separate page handler, in Barrelfish, the faulting applications themselves can manipulate the available memories through capabilities. This renders problems of external paging insignificant.
3) **Device Memory:** Barrelfish also has *DevFrame* and *PhsyAddr* capabilities, which ensure that there are capabilities that can be mapped to RAM capabilities without requiring that the memory be zeroed before it being read so as to prevent data leakage.
4) **Kernel Interface:** This feature enables Barrelfish to partition its kernel. The kernel is divided into privileged space and user space. As seen in 1 the user level can access the privileged operations by invoking Kernel capabilities via certain system calls.

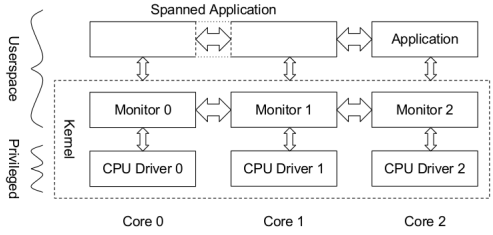

Fig. 1. Barrelfish Kernel Partitioning

To manage resource utilization by the kernel, Barrelfish maintains an individual copy of the kernel on each core, and then performs synchronization to keep the system consistent. This is done to reduce the effort spent in synchronization, since the individual copies have share-nothing policy by default. This design when extended to the userspace, creates a distributed system. This lets the system achieve the goal of running kernels on cores of different architectures in parallel. But it still faces the problems of low throughput and latency. The solution for this problem is to have a shared memory which does cache coherency, which requires certain communication channel between the cores to pass the capability information around.

In [6], the authors have presented basic operations that can be performed to when sharing of capability information is considered. The channels betweens cores maintain a send-once FIFO queue for sending messages containing these operations. Since sending and receiving these messages implies that there would be a need for synchronization between cores, there is a conflict which arises due to the fact that the initial goal was to maintain a synchronization-free system. The authors address this issue by considering object loyalty, which is that every capability object is assigned to an owning core. This ownership model is implemented along with certain defined invariants regarding the capability objects. [6] also gives preconditions and postconditions that must be met (as far as possible), to implement the Barrelfish capability operations.

We give a brief overview of the invariants, preconditions and postconditions below as mentioned in [6]-

1) **Invariants:**
   - **Ownership Property:** Every capability that is not Null has an *owner* property that refers to a running kernel instance in the system.
   - **Consistent Ownership:** Any two capabilities that are copies must have the same *owner* property.
   - **Owner Copies:** Every set of all copies of a capability must contain at least one capability where the owner matches the location. With other words, the owning core must always have a local copy.
2) **Preconditions:**
   - **Copy:** The source capability is not Null and the destination core is valid.
   - **Retype:** The source capability is not Null, may be retyped to the destination type, and has no descendants in the system.
   - **Delete:** The target capability is not Null.
   - **Revoke:** The target capability is not Null.
3) **Postcondition:**
   - **Copy:** The destination core has a copy of the capability and the invariants of the system hold.
   - **Retype:** The specified descendants exist in the target slots and the invariants of the system hold.
   - **Delete:** The source capability is Null and the invariants of the system hold.
   - **Revoke:** All copies and descendants of the source capability have been deleted and the invariants of the system hold.

These operations all refer to slots of memory. A slot is the storage required to store an individual capability, where a Null capability corresponds to n empty slot. As mentioned above, every slot has an owner as well as an immutable memory location. The slots are considered local if the owner and the memory location are the same, and foreign otherwise. The behaviour and interaction between these capability operations are described through global pseudocodes (for each operation), which we will not mention here due to lack of space.

Barrelfish design has successfully applied aspects from an object oriented approach to a distributed, multi-core, many-core system. Though the authors have set out various optimizations as future scope and there are currently issues in the design regarding undelivered/ unresponsive messages, it does lay out the basic groundwork for other capability based resource management systems to build on.

III. FILESYSTEMS

File systems are used to dictate how data is stored and retrieved, while traditionally file systems were designed for single system, although the advancements in distributed systems, gave rise to distributed file systems (often referred to as network file systems) which are now widely deployed. However one important concern, is the underlying access control mechanism for such a system, and capabilities are better suited[**?**] for this for a variety of reasons:

- ACLs (Access Control Lists) would be a bottleneck for a distributed system, since it requires explicit authentication whereas a user possessing a capability can access the resource listed in the capability with the rights specified in it.
- capabilities explicitly list privileges over a resource granted to the holder and hence naturally support the property of least privilege

While access control using capabilities sets the theme for access control in distributed file systems, their implementations vary vastly as seen by the implementation detail of two popular distributed file systems, OrangeFS and TahoeLAFS. While discussing these we limit ourselves to the capability aspect without addressing the other fundamental pillars of distributed file systems such as sharding and redundancy mechanisms.

A. *OrangeFS*

OrangeFS emerged as a development branch of Parallel Virtual File System (PVFS2) in 2007, and tries to address applications for the future, one of which is secure access control. PVSF2 was designed with focus on performance for parallel applications sharing data across many clients for which it uses the notion of an 'intelligent server' architecture. OrangeFS being branched out of PVFS2 is build entirely in C itself like the latter.

OrangeFS is an object based file system, where each file and/or directory has two (or more) associated objects to it, corresponding to the metadata and the actual file data (can be split into multiple objects). Storage nodes in OrangeFS usually provide two services

- Metadata Service: which sends and receives the metadata of directories and logical files (the actual data need not be stored on this node)
- Data Servce : which sends and receives data for objects stored on this node.

On a high level, OrangeFS uses a certificate based security consists of two main parts, Authentication and Access Control, and like all contemporary file systems provides access control by determining a user's access to an object based on the object's ownership and permissions. Traditionally, secure systems have used certificates to cryptographically assure verifiable identity information in storable blocks of data, these certificates are generally associated with a private and public key pair, of which the public is generally stored in the certificate along with other fields significant to the application in consideration. In case of OrangeFS the key fields are :

- Issuer Distinguished Name: to identify the entity that issued and signed this certificate
- Subject Distinguished Name(DN): identifies the user that the certificate authorizes
- Validity Period: the duration for which this issued certificate is valid.

Fig. 2. OrangeFS Architecture

Another entity involved in this system, is the Lightweight Directory Access Protocol(LDAP) directory which stores user objects (such as UID and GID to evaluate access rights). The Subject DN from the certificate is used to uniquely identify users in LDAP directory, and OrangeFS servers interact with LDAP servers to obtain the user information to enforce the desired access control.

In every system that makes use of certificates for security must have a root(or roots) of trust, which can be used to issue valid certificates for other entities in the system. In OrangeFS, the trusted root certificate is called a Certificate Authority(CA) certificate, and is securely stored across each OrangeFS servers to sign and validate user certificates. Every user must have a valid user certificate (and the corresponding private key to the public key signed in their certificate) which is signed by the CA certificate. OrangeFS uses the term credential for a signed user certificate, and capabilities are signed objects containing permissions for file system objects.

As seen in Fig. 2, capability based access control can be explained in a 4 step process:

1) To perform a request, client sends its credential along with request to the server.
2) Server verifies credentials signature and fetches corresponding user information from the LDAP server for the subject DN in the user certificate.
3) Server compares the user permissions with the object permissions and constructs a capability for the request.
4) Client can use the capability to act on the file system object

One aspect to notice is that all the secure components are created with an expiration period, since revocation in a system

based on capabilities can go in either two directions, by basing itself on revocation through time outs, or by using a central entity that performs revocations but there introducing a bottleneck again.

B. TahoeLAFS

One fundamental difference of TahoeLAFS(Least Authority File System) from OrangeFS is that it assures 'provider-independent security', implying that the server itself which provides the file system does not have the ability to read or modify the data, hence reducing quite a lot of latent security issues that arise from a compromised server. Moreover this bolsters their claim of true 'least-authority' semantics. This guarantee is integrated naturally into the implementation mechanism of Tahoe-LAFS and requires no additional step or management.

TahoeLAFS[**?**] considers two kinds of files : mutable and immutable, and provides its access control mechanism with this in consideration. Any file uploaded to the storage grid can be chosen as either kind, and any user who has read-write access to a mutable file or directory can give other users read-write access or even any diminishing subset of the capabilities owned by the user.

Irrespective of the type of file, all files in TahoeLAFS undergo erasure coding using Reed-Solomon codes, which allows a write to split a file into N share out which K shares need to be used to read (however Tahoe maintains small numbers for both K and N thus maintaining low computational costs for erasure coding)

Fig. 3. Immutable Files in TahoeLAFS

In TahoeLAFS, each immutable file has two associated capabilities, the read capability called as the *read-cap* and verify capability or *verify-cap*. To create an immutable file, the client chooses a symmetric encryption key to encrypt the file which it loads on the server. The verify-cap is derived from the ciphertext of the file using a secure hash (SHA256d[4]). However verification has two problems to be taken care of

- If integrity check fails, client needs to isolate the erasure code shares that were wrong, for which a Merkle tree is constructed over the erasure code shares, allowing client verification of each share involved.
- However just the Merkle Tree over erasure code shares is not sufficient as that does not link the shares to the original file content[5], and hence another Merkle Tree is constructed over the ciphertext itself.

Now, as seen in Fig.3 the roots of both the Merkle Trees are kept on each server alongside the shares, and is called as the 'Capability Extension Block' [6]. The verify-cap is now the secure hash of the Capability Extension Block, and the read-cap to an immutable file is the verify-cap along with the original symmetric encryption key. Thus allowing the clients with the read-cap to simply pass on the verify-cap component to other clients when it wants to give it a diminished form of it's capability, moreover since the file is encrypted with the symmetric key prior to storing on to the servers, the servers themselves cannot read the data, and any attempts to modify would break the integrity check done by the verify-cap.

Fig. 4. Mutable Files in TahoeLAFS

For mutable files the underlying process is significantly different and more convoluted to take care of the three desired capabilities *read-write-cap*, textitread-only-cap and *verify-cap*. Each mutable file is associated with an RSA key pair(with private key SK), a client first computes the write key, WK = SHA256d(SK) and corresponding read key, RK = SHA256d(WK). To write, a client generates a new 16 byte salt, and computes the encryption key with EK = H(RK,salt). A similar process for integrity verification is performed over the ciphertext, by computing a Merkle Tree over the erasure coded shares of the ciphertext, and the client signs the root hash of the Merkle Tree as well as the salt, with the private key SK. So the verify-cap for mutable files are the secure hash of verifying key : VC = SHA256d(VK), hence allowing clients with verify-cap to check that the verifying key which is stored on the server is the right one. The read-only cap to a mutable file include VC, and the read key RK(RC = VC,RK), while the read-write-cap to a mutable file includes VC and the write key WK (WC = VC,WK). However the owners of read-write-cap still need to be able to produce digital signatures on new versions for which the signing key SK is stored on the server encrypted with WK which is only available to the clients with read-write-cap.

A directory could simply be treated as a mutable file which contains a set of tuples of ¡child name, capability to child¿, this allows anyone given read access to a directory to be able to learn the child names and corresponding read capabilities, and similarly for read-write capabilities as well as for mixed capabilities for individual files of the directory. However, Tahoe implements transitive read-only access to directories, i.e. directories which have read-only access to the directory can get only a read-only-cap to a child. For this, directory includes two slots for the caps for each child, and read-write-caps for the children are encrypted separately before being stored in the mutable file.

C. Comparison

It becomes clear from the above two sections that Tahoe-LAFS and OrangeFS, while use the same abstraction of capability based access control mechanisms, they are entirely

[4]SHA256d(x) = SHA256(SHA256(x)) (to prevent length-extension attacks double SHA hashing is performed

[5]the creator of the file could generate erasure code shares from two different files and use the shares from both in the set of N shares covered by the Merkle Tree, resulting in a different file depending on the subset of shares used to reconstruct

[6]Logically should be part of the capability but maintained on the servers to minimize capability size

different in their implementation, and even their security properties and guarantees, the most significant being the notion of provider-independent security introduced in TahoeLAFS. There are many other subtle implementation details that make them highly incomparable against each other, such as the erasure coding in TahoeLAFS allows different subsections of files to be loaded from different servers, thus improving performance by improving parallelism. Moreover OrangeFS is a completely build from C, whereas TahoeLAFS is implemented in Python. Another aspect is the revocation of capabilities, while OrangeFS relies on time outs to perform revocation, in case of Tahoe since the key once given cannot be revoked, the only option is for a client to redistribute a new set of capabilities under new keys for an object, which is a significantly costly process. Hence in short, evaluating the performance of these two types of systems would not provide any insightful results.

IV. User Space implementations

A. Capsicum

Capsicum is one user space implementation for a capability based access control system developed at the University of Cambridge Computer Laboratory. [7] It is a lightweight framework that extends the POSIX API. It implements capabilities by modifying current kernel primitives as well as by performing sandboxing techniques in the userspace. It provides capabilities for logical applications by implementing *compartmentalization* of the application. The first implementation for Capsicum was on FreeBSD 8.x, and as a result of more research being conducted, it is now being released along with FreeBSD 9.0 as an experimental feature.

The implementation of capabilities by extending, rather than gives the applications the benefit of having least-privilege operations in the system. Separation or compartmentalization is a popular technique used by OSs today to help security-critical applications in dealing with vulnerabilities. It limits the impact of the system by exposing only a small part or compartment of the entire system to the vulnerability.The other critical aspect of implementing such a system is to make sure that this is achieved by minimum effort.

Capsicum achieves this by implementing certain security primitives like *capability mode* and *capabilities.* Some of the implementation modifies the kernel level primitives, while others are for userspace implementation. These primitives together support logical compartmentalization of the application. Once the application code is separated into individual logical parts, we can run independent sandboxes to form logical applications as shown in Figure 5.

Following is the list of all such primitives:

- **capabilities:** file descriptors with refined privileges.
- **capability mode:** implementing sandboxing so as to deny access to global namespaces.

[7]https://www.cl.cam.ac.uk/research/security/capsicum/

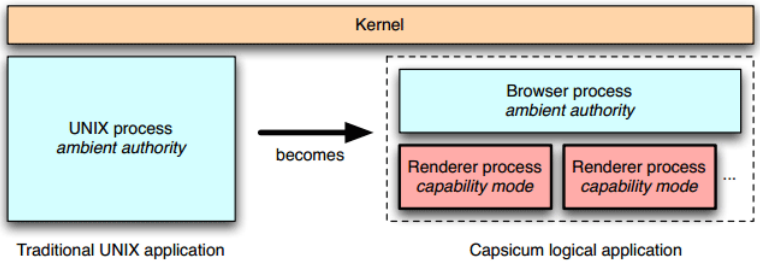

Fig. 5. Logical compartmentalization of applications

- **process descriptors:** replacing capability-centric process IDs.
- **anonymous shared memory objects:** an extension to the POSIX shared memory API.
- **rtld-elf-cap:** constructing sandboxed applications from run-time linkers by modifying ELF
- **libcapsicum:** library to create and use capabilities components
- **libuserangel:** library allowing sandboxed applications or components to interact with user angels, such as Power Boxes.
- **chromium-capsicum:** Google's Chromium web browser that provides effective sandboxing of high-risk web page rendering.

Capsicum has a host of capability implementation that can be used, but they leave the end user to use his/her discretion in order to maximize the security impact for their respective application. Implementing the *capability mode* for example, implies that a process flag is set by the system call. Once this flag is set, the system goes into the capability mode, and the process is then denied access to global namespaces. In addition to this, in the capability mode, the access to the certain system calls is also limited.

In Capsicum, capabilities can also be implemented using file descriptors. File descriptors have certain properties that are considered desirable in a capability system, like being a tokens of authority, being unforgeable as well as being able to pass the properties through inheritance. In Capsicum, the *cap_new* system call extends the file descriptor by creating a new capability and a set of rights. These rights get checked by *fget* which performs the conversion of file descriptor arguments to system calls to in-kernel references.

B. capBAC

V. CAPBAC (Capability Based Access Control)

CapBAC[**?**] is an access control framework designed for the Internet of Things, the primary idea is to accommodate seamless integration of devices in the internet by facilitating a distributed approach in which devices themselves can make authorization decisions. One major aspect that CapBAC focuses on is basing access control decisions based on contextual information related to the end-device itself (Various

IOT use-case situations relating to emergency response procedures).Almost all IOT access control architectures proposed can be classified into three types, centralized, centralized and contextual, and distributed. In a purely centralized approach, while access control logic is located in an entity without constraints of resources, it nonetheless becomes a bottleneck, and moreover the context of the end device cannot be taken into consideration. The centralized and contextual type is a solution which allows contextual information to be passed to the centralize policy decision process (PDP) and thereby allowing contextual information to participate in decision process. [8]. Alternatively in distributed architectures, the access control logic is embedded into the end devices, while the advantages of this mechanism is numerous , the key drawback is that it requires the end devices to be extended to support access control logic and the required cryptography support.

As shown in Fig.6, the architecture of CapBAC is intuitive and straightforward, it considers three entities in an interaction, an issuer, a subject and an end device to access. The issuer is in charge of granting a capability token for a subject for end device access requests the subject makes. The capability token itself is a JSON file, which contains :

- Capability Identifier(**ID**) - random identifier generated for a capability token
- Issuer(**IS**) - the entity that issued the token
- Subject(**SU**) - the public key of the subject to which the rights from the token are granted
- Device(**DE)** - a URI to identify the device to which token applies
- Other fields to account for the permissions to be granted, the duration or validity of the token and the context conditions of end device.
- Signature(**SI**) - the signature of the file generated by the issuer

Fig. 6. CapBAC Architecture

CapBAC uses ECDSA (Elliptic Curve Digital Signing Algorithm) to authenticate capability tokens and end device requests, for which all issuers and subjects are setup with an ECDSA key pair. A subject sends the request for access to a resource on an end device to an issuer, and an issuer validates and produces a capability token for this subject as a JSON file, and then signs it with its private key from it's ECDSA key pair, the subject now can sign the token with its own private key and forwards the token and it's signature to the end device. The end device first verifies the signature against the public key for the subject (**SU**) from the token, thus validating that the request did genuinely arrive from the source that is granted the capability, and then verifies that the token itself originates from a valid issuer by verifying the signature (**SI**) in the token, against the public key of the Issuer (**IS**).

[8]In time critical application domains, this has a slight difference between actual context and context used in decision making process

VI. IMPLEMENTATION AND RESULTS

¡k,j,s¿

VII. CONCLUSION

¡k,j,s¿

REFERENCES

[1] K. Elphinstone and G. Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 133–150. ACM, 2013.

[2] I. Kuz, G. Klein, C. Lewis, and A. Walker. capdl: A language for describing capability-based systems. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, APSys '10, pages 31–36, New York, NY, USA, 2010. ACM.

[3] B. W. Lampson. Protection. *Proc. Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Mar. 1971. reprinted in Operating Systems Review, 8,1, January 1974, pages 18–24.

[4] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM (JACM)*, 24(3):455–464, 1977.

[5] M. S. Miller, K.-P. Yee, J. Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals, 2003.

[6] M. Nevill and T. Roscoe. An evaluation of capabilities for a multikernel. Master's thesis, Systems Group, Department of Computer Science, ETH Zurich, 5 2012.

[7] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. *Interactive Theorem Proving: Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, chapter seL4 Enforces Integrity, pages 325–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.