

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SoK: Capability Operating Systems: Is the Future Finally Here?

Noah Mauthe, Eric Ackermann, and Sven Bugiel,
CISPA Helmholtz Center for Information Security

https://www.usenix.org/conference/usenixsecurity26/presentation/mauthe

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

SoK: Capability Operating Systems: Is the Future Finally Here?

Noah Mauthe, Eric Ackermann, Sven Bugiel
CISPA Helmholtz Center for Information Security

Abstract

Capability operating systems have existed for six decades, yet we do not have a systematic way to compare them and their different design aspects. We even lack a consensus on the purpose of capabilities in these systems. Given the recent resurgence of interest in capability systems in different domains, we provide a definition of such systems and a set of characteristics to describe and distinguish them. Applying our characteristics to seminal capability systems, we then provide a taxonomy of fundamental design approaches that emerged over the last sixty years. Our results lead us to discuss further research in this field and highlight open challenges in creating *pure* object-capability systems.

1 Introduction

Capability-based access control can offer many benefits. By supporting fine-grained subjects and objects, flexible creation of new protection domains for compartments with a clear purpose (i.e., least-privilege) they have the potential to eliminate entire attack classes that current non-capability systems do not offer. These include memory safety [97], confused deputies [35], and supply chain attacks [79].

Despite these conceivable benefits, capabilities have never seen widespread adoption. In fact, there is persistent doubt they will ever be viable on a broad scale:

> [Butler Lampson [73]] Capability systems are the way of the future, *and always will be.*

However, interest in capability systems never wavered over the last six decades. In fact, the last two decades have seen a steady stream of new capability works. Many systems build on CHERI [97] and are primarily concerned with memory safety [23, 25, 34, 77, 90, 94, 95]. Some reuse existing commodity hardware [22] or propose their own modifications [3, 5, 101]. Others focus on capabilities as a way of securing inter process communication (IPC) [62, 93] or remote procedure calls [92]. These advances are paired with renewed industry interest in building capability languages [92], operating systems [62], or even hardware [29, 34, 83].

While this multiplicity clearly shows a growing interest in capabilities for a diverse range of applications, it complicates the understanding of the term *capability*. Originally, capabilities were intended to isolate user tasks from each other [20]. Over the years, the purpose was adjusted to serve as means of distributed addressing [72], a method of encapsulation [98], to ensure memory safety [97] or the reliability of systems [84].

Naturally, these changes have resulted in varying notions of *capability* for different purposes. We propose a framework that helps characterize and map the design space. Applying the framework reveals that the majority of capability systems is concentrated in a small fraction of the design space. The rest of the design space is underexplored, opening up new avenues of research. We identify challenges that we believe can move the field forward and highlight new problems that can benefit from a capability based approach.

To systematically analyze capability systems and extract new insights, we answer the following research questions:

RQ1. How do we define capability systems, and what are their basic characteristics? Despite their differences, capability systems share common characteristics. Articulating these will not only help in analyzing the systems, but also in differentiating them from other, non-capability systems. The need for such a differentiation has been touched upon about 20 years ago by Miller et al. [71]. However, recently, the CHERI family of works [25, 90, 94] and industrial products like ARM Morello [34] or Google Fuchsia [62] caused a spike in interest in capability systems. Their differences show how "capability" has become a broad term in access control, denoting capability concepts from fat-pointeresque constructs [94] to IPC or file handles [62, 93] and beyond [1, 72]. This underlines the need for an up-to-date and broadly applicable baseline that characterizes capability systems.

RQ2. How can we systematically compare and contrast capability systems? The second challenge is to find a common frame of reference and language to compare the potentially significantly different solutions from almost six decades

at a technical level. To our knowledge, the last work that compared capability systems was again Miller et al. [71] in 2003. Based on seven security properties, the authors examined three capability models to clarify common misconceptions of capability systems and advocate for the security benefits of pure object-capability systems over access control list systems. In our work, we extend, refine, and rework the properties defined by Miller et al. From the existing body of literature, we identify eleven additional characteristics that can be used to compare capability systems (§4). We showcase the application of our criteria at the example of 31 capability systems that represent different eras of system design—from the 1960s to the 2020s—and provide a taxonomy of fundamental design approaches (§5). We find that a) a new model for IPC capabilities emerged in recent years, introduced by Capsicum [93], but practically seen in appified systems like Google Fuchsia [62] and Android [18], and b) the design differences between different capability systems are more nuanced, entailing different security benefits that need to be carefully considered in the basic system design.

RQ3. What are the open questions and possible future directions in capability research? Based on our systematization of capability systems, we discuss our observations that could be beneficial for further research in the field and highlight open challenges (§6). We find a vast design space for capability systems, but large fractions remain unexplored. Especially synergies between object capabilities and modern system stacks with extensive middlewares are under researched. We challenge the long-held belief that capability systems require explicit revocation, as other mechanisms with the potential to make it obsolete have been proposed. Moreover, revocation has been "misappropriated" for purposes other than the historically prevalent "retroactive removal of authority", necessitating further discussion. Finally, we highlight a lack of research into policy design for capability systems and emphasize the need for operating system design that aligns with security goals.

2 The Deceptive Simplicity of Capabilities (or Why We Study Systems)

Capabilities are a very simple primitive. The literature widely agrees that they combine *designation* of a resource and *authority* over said resource in an *unforgeable* and *tamperproof* way [13,84,87,98]. However, these basic properties lack sufficient context to establish any meaning of a capability. Dennis and van Horn's original definition has some context:

Definition 1 (Dennis and van Horn [20])**.** *Each capability [. . .] locates by means of a pointer some computing object, and indicates the actions that the computation may perform with respect to that object.*

Still, what are *computing object* and *computation*?

2.1 Capability Analysis Requires Systems

Dennis and van Horn's definition directly leads to our choice of discussing capability systems instead of capabilities. Lacking a definition of *computing object* and *computation* precludes discussion of all security relevant aspects. Accordingly, discussing capabilities is mostly meaningless without the context of the system they operate under.

Observation 1

Capabilities are just a mechanism, as most properties stem from the system using capabilities. An objective comparison needs to differentiate *capabilities* from *capability systems* and compare them at the system level.

2.2 Common Definitions

The following definitions are intended to aid in understanding the characteristics we present in §4 and modernize Dennis and van Horn's definition, but remain deliberately abstract to encompass six decades of diverging research.

Definition 2 (Object)**.** Everything addressable in the system.

Definition 3 (Subject)**.** Everything that can hold a capability.

Definition 4 (Active Protection Domain)**.** The active protection domain of a subject is defined by the authority it can exercise on the system. In a capability system, this is governed by the set of capabilities the subject currently holds.

2.3 Capabilities and How to Obtain Them

Dennis and van Horn's work [20] introduced the notion that capabilities should be used *explicitly* to eliminate ambient authority. Linden [60] and Hardy [35] further elaborate on the importance of this idea and how it separates capabilities from other access control systems. This leads us to an updated definition:

Definition 5 (Capability)**.** *A capability designates an object and the actions that any subject holding said capability may perform on that object* ***only when using the capability***.

Since the subjects "hold" capabilities, the question of how to obtain capabilities arises. The literature recognizes three ways:*Creation*: Creating an object necessarily creates a designating capability that the object creator needs to refer to and exercise authority over the object. *Delegation*: Capabilities can be shared between subjects. *Inheritance*: Capabilities can be inherited from parent subjects.

2.4 Ambiguity Leads to Diversity

The simplicity of the capability definition leaves a lot of room for interpretation for system designers. In order to resolve

this ambiguity, system designers must face crucial decisions including, but not limited to: *What can be addressed in a system? What is the unit of a subject? Should capabilities be the exclusive way to locate objects?*

Observation 2

Existing capability research seldomly states these decisions explicitly. This freedom of interpretation and lack of transparency, in combination with changing requirements and the evolution of technical possibilities, led to a vast and diverse landscape of capability systems [10,19,62,69,72,94]. Regrettably, these differences in design goals and characteristics are rarely considered when comparing capability systems, and research on them is lacking. Even Miller et al.ś work [71], which represents the most comprehensive analysis of capability systems to date, operates on a level of abstraction that glosses over such details.

3 Systematic Literature Search

Since discussing capabilities requires the context of a system (§2.1), we focus on capability-based operating systems and hardware-software co-designs[1]. Analyzing capabilities in the context of operating systems promises the most impactful insights for the research area. The system level encompasses both hardware capabilities, mostly used for memory protection [11, 37, 94, 97], and capabilities meant to protect high-level objects [62, 84, 98]. This leads to a framework that intends to be suitable for a generalized analysis of capability systems.

3.1 In- and Exclusion Criteria

The focus on operating systems excludes other areas where capabilities have found adoption. While capability-based languages [68, 69, 92] can aid in writing compartmentalized software that follows the principle of least privilege, we do not discuss them.

Hardware and operating systems are the foundation of the security architecture and define the constraints for the achievable security goals. Accordingly, capability based security built on runtime primitives will hold in said runtime, but can be undermined by processes outside the governance of the runtime. On the other hand, supporting capabilities at the operating system level will affect all software in the system and can in turn be used as a foundation for capability-language runtimes. This motivates our approach of trying to capture the system holistically and leads us to consider capability-based languages as well as the usage of capabilities for information flow or program verification [12, 30] as interesting, but orthogonal problems.

[1] Henceforth referred to as capability systems for brevity

Consequently, we define the following inclusion (+) and exclusion (-) criteria.

+ Must describe a system that uses capabilities as main access control mechanism
+ Must contain sufficient technical detail for fair assessment
+ Must be an operating system or kernel
- The work solely describes capability-enabling hardware without sufficient implementation
- The work describes capabilities that operate under ambient authority
- The work describes a capability language

3.2 Literature Collection Strategy

Collecting capability-based works is challenging. Originally envisioned to provide isolation between users and processes at the operating system level during the advent of multi-user systems, work on capabilities has branched out into many domains. Hardware design for memory safety [34, 57, 97], sandboxing of critical components [93], securing applications at the language level [67, 70], cyber-physical systems [23] as well as information flow and program verification [12, 30] are only a subset of the domains. This, along with the longevity of the field means that picking works from recent top tier conferences in security and systems is insufficient. Furthermore, keyword searches can be difficult as the term *capability* is very general and occurs in many domains.

To overcome these challenges, we used a graph-based approach. We assume that any capability system cites, directly or transitively, one of the early and highly influential capability papers. The seven early capability systems [11, 26, 42, 52, 76, 96, 98] analyzed in Levy's capability book [58] from 1984 and Dennis and van Horn's "Programming semantics for multiprogrammed computations" [20] that coined the term capability serve as our initial set.

Starting at these seed papers, we build a citation graph[2] using openAlex [80] and the semantic scholar API [48]. Since these are influential works, this citation graph contains ≈1,000,000 works from which we need to filter the works relevant to our scope. Using a combination of automatic and manual filtering, we arrive at a dataset of ≈ 500 works. We dot not claim, or indeed aim for completeness. However, we believe that the selection provides us with a broad overview of capability systems, leading to a diverse set of representative characteristics (§4).

3.3 Selection of Showcase Papers

As we systematize and compare the major directions of capability research, we focus on the works that took capability

[2] We limit the search to a depth of three edges

research in new directions. Accordingly, our selection of papers showcased in Table 1 groups some works that share similar characteristics and highlights systems that introduced novel characteristics. *We want to stress that our systematization framework is not solely based on the works listed in the table, but was built from a review of the filtered dataset.*

4 Capability System Characteristics

In this section, we present the capability system characteristics we extracted from literature. Characteristics §C1–§C5 represent the minimal requirements for systems to qualify as object capability systems.

Readers familiar with Miller et al.'s *Capability Myths Demolished* paper will observe similarities to the characteristics present in their work. While our characteristics are our own extensions of their work and are meant to be understood on their own, we attribute their strong influence by marking the corresponding characteristics with $^{\text{M}}$.

C1 Subject Aggregated Authority$^{\text{M}}$

Capabilities are defined as tokens that subjects can hold and that carry authority (Definition 5). Accordingly, the authority is aggregated at the subject, in its capability set. This is true (●) for all capability systems. Note that *set* is meant here in a general sense, rather than an implementation detail akin to the C-List introduced by Dennis and van Horn [20].

From a policy standpoint, subject aggregation of authority is a trade-off compared to object aggregation, which is dominant in today's systems (e.g., access control lists (ACLs) [75]). It allows for an easy inspection of a subject's active protection domain, but complicates listing the subjects that can access a specific object. Figure 1 contrasts an object aggregated access control list (Figure 1a) with subject aggregated capabilities (Figure 1b) that express the same policy.

C2 Designation Only Through Capabilities$^{\text{M}}$

Capabilities are a liaison of designation and authority. Accordingly, there is no need to designate objects in other ways. In systems with this property (●), authority over an object is required in order to designate it. This does not mean that internal usage of names is disallowed. However, each designation across protection domains is required to use capabilities rather than names.

C3 No Ambient Authority$^{\text{M}}$

As capabilities have to be used *explicitly* to wield their authority, ambient authority can be abolished. There simply is no need for it anymore, as each designation through a capability automatically carries the associated authority.

Some capability systems, however, rely on the kernel to ensure the integrity of their capabilities (we elaborate in §C7). These systems automatically manifest ambient authority *at the kernel level*. We do not differentiate such systems for C3 and assign ● in case all subjects *governed by the capability system* are without ambient authority. Systems that feature no ambient authority *at all* after bootstrapping the system are considered in a separate characteristic (§C12).

C4 Composability$^{\text{M}}$

Typically, subjects *authorize* subjects and *access* objects. In systems with object aggregated authority, this is a necessity.[3] For subject **A** to authorize another subject **B** to access object **O**, **A** has to modify **O's** ACL. Even though **B** was authorized, it was not involved in the transaction (see Figure 2a).

With subject aggregated authority, this separation between access and authorization disappears. In order for subject **A** to authorize another subject **B** to access object **O**, **A** has to delegate a capability referencing **O** to **B** (illustrated in Figure 2b). To delegate a capability, **A** has to designate **B**. Since all designation has to occur through capabilities (§C2), **A** needs to have a capability and, hence, authority over **B**. Effectively, the delegation from **A** to **B** is an access from **A** to **B**. Accordingly, each delegatee has to be considered as both a subject *and* an object. Definition 2 underpins this, as **B** has to be addressable, thus falling under the object definition.

In systems with composability (●), this lack of differentiation allows for the construction of networks of authority of arbitrary depth. This can be used for implementing separation of duties, strong encapsulation, and highly compartmentalized applications that follow the principle of least privilege [60,98].

C5 Explicit Delegation$^{\text{M}}$

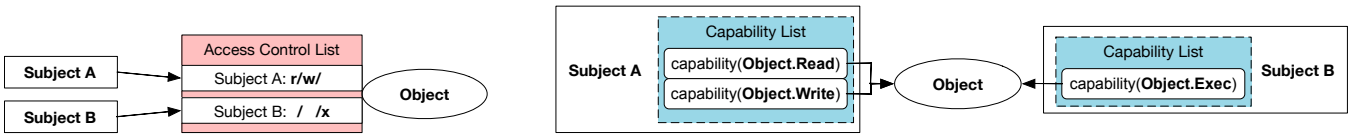
If all designation can occur through capabilities (§C2), all access to objects (and thus subjects) is access controlled. However, this does not determine whether the right to *access* should imply the right to *authorize* (while the opposite is implied (§C4)). Distinguishing whether an access across protection domains carries capabilities, i.e., is a *delegation*, is essential to enforce properties that build on confinement [14,71].

We differentiate systems where authorization has no preconditions (○), requires an established channel (◐) or is a privilege in itself (●).

C6 Context Independent Representation

The abstract concept of a capability combines designation with authority. It is possible that the instantiation of this abstract concept is in form of a single atomic object. However, the meaning of a capability is only given in the context of a

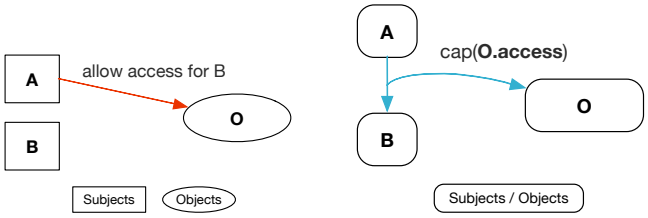
[3] Mechanisms that change the subject's protection domain, like `setuid` and SELinux domain transitions [40], are *not* authorizations in our definition.



(a) Object aggregated.

(b) Subject aggregated.

Figure 1: Differences between subject and object aggregated access control. Both examples express the same policy.



(a) No composability.

(b) Composability.

Figure 2: Differences between systems with and without composability. Both times, Subject **A** authorizes Subject **B** to access Object **O**.

system (§2), which affords system designers flexibility. The representation of a capability as seen by the subject (the *frontend*) may not contain sufficient information to make it usable as a capability. The system can keep this complementary, but required, information in a separate *backend* data structure. This is typically transparent to subjects. In such scenarios, keeping an inseparable bond between frontend and backend becomes a responsibility of the system.

When discussing the representation of a capability, we refer to its *frontend*. The distinction of frontend and backend highlights that the representation of a capability can take arbitrary forms. While a context independent representation (●) eases delegation, context dependent representations (○) can contribute to a capability's tamper resistance.

C7 Capability Protection

The extension of the mental model to consider *frontend* and *backend* of a capability separately also aids in understanding the different approaches found in the literature to ensure capabilities are *unforgeable* and *tamperproof*. However, as a subject aggregated form of access control, we need to assume that subjects can indeed try to forge or tamper with frontends. We observed five techniques across two categories in the literature to protect capabilities.

C7.1 Distinguishing Capabilities from Data

If the system distinguishes capabilities from data, data can not be passed off as capabilities, eliminating forgery.

Separation (**S**) is used in System 250 [24] to achieve this. In such systems, capabilities reside in capability segments and can only be loaded into capability registers. This total separation of capabilities and data ensures that capabilities can only be modified using well-defined instructions.

Memory Tagging (**T**), most prominent in the CHERI family [4, 5, 23, 25, 34, 77, 90, 94, 95], achieves the same effect without capability segments. Instead, capabilities and data are stored in the same segments and capabilities are distinguished by a hardware enforced tag-bit. If an attempt is made to modify a tagged capability, the tag bit is invalidated, causing the capability to become data.

Handles (**H**) are the final technique in this category [20, 84, 93, 98]. They are opaque frontends that map to a corresponding, subject-specific backend. This ensures that any modification of the frontend will result in an invalid lookup. Acquiring knowledge about frontends in other compartments does not allow forgery or uncontrolled duplication of the capability, as the lookup table validates each subject's frontends and can only be modified through controlled operations.

C7.2 Probabilistic Methods

In systems using probabilistic protections, forgery is *in principle* possible, but sufficiently unlikely.

Systems with atomic capabilities (**A**) unite frontend and backend in a single object [72, 89], i.e., subjects holding a capability have access to both frontend and backend. The authenticity and integrity of capabilities are ensured using a variety of probabilistic means (e.g. *passwords* or *HMACs*) which have to be verified when the capability is wielded.

We also observed newer systems with a form of probabilistic handles (**P**) [3, 47]. Probabilistic handles are still opaque, but not subject-specific. Instead, a central table is used for all lookups. Great care needs to be taken not to leak probabilistic handles or allow easy guessing. Although both is also true for atomic capabilities, it is easy to overlook for probabilistic handles, due to their similarities with actual handles.

C8 Policy Controlled Delegation

Capabilities can flow freely between subjects in the system (○), assuming that there is an established, access controlled channel (§C5). As capabilities are prima facie evidence of authority, checking policy on access would require a secondary access control mechanism. Thus, it may be desirable to control the spread of authority at delegation time,

using a (central) policy (●).

C9 Protection Domain Tracing

Generally, capabilities can be realized as a decentralized form of access control (○). However, some systems utilize a centralized mapping of subjects to their respective protection domains (●). Keeping a centralized map can counter the complexities of decentralized management. Further, it can aid with policy enforcement (§C8) and explicit revocation (§C14).

C10 Minimal Subject Unit

The expressiveness of any possible policy defined on a capability system is limited by the minimal subject unit. Since protection domains are defined per subject, smaller units tend to require less privileges and are better at enforcing the principle of least privilege [55]. This unit can vary dramatically between systems. We observed this ranging from users (**U**) or processes (**P**)[4] to individual method invocations (**M**).

In contrast to prior work that differentiated systems on whether subjects can be created dynamically [71], we focus on the unit of the subject instead, since we consider this a more meaningful differentiation. In a system with process as the smallest subject unit, one can still dynamically create new processes for every method call. However, systems with an inherent method-level subject unit will often automatically (and likely more efficiently) create new protection domains dynamically. Thus, we consider *dynamism* a derived property of the minimal subject unit.

C11 Single Level Storage

Access control mechanisms typically only cover one semantic level or type of object (e.g., different enforcement for files and memory segments). If different object types can be uniformly addressed, the access control mechanism can be agnostic of the object type. This is easier to achieve in systems with subject aggregated authority, as the objects do not need to carry metadata about subjects (e.g., ACL). In contrast, in systems with object aggregated access control, neither objects nor subjects can be agnostic about each other.

While capabilities could be used to build a form of uniform addressing for any resource imaginable, we never observed this. However, there are systems that unify the addressing of memory and storage to a single level storage (●). In these systems, files are considered long-lived memory segments, foregoing separate treatment.

C12 Ringless System Design

As alluded to in §C3, any system reliant on privilege rings has inherent ambient authority. It is entirely possible to utilize capabilities as a form of more fine-grained access control on top of privilege rings. However, capabilities alone are sufficient to convey all privilege in the system (●). This allows the elimination of ambient authority throughout the system.[5] Accordingly, a system without privilege rings can rely only on capabilities for all addressing, thereby eliminating all ambient authority.

C13 Persistence

It can be desirable to persist access control decisions, namely capability delegations, throughout reboots (●). This can avoid re-delegation of capabilities after each bootstrapping. However, this comes with a challenge: The capabilities need to either be valid across reboots, or reconstructable from data. This can result in constructions that emulate ACLs.

C14 Explicit Revocation

Not all systems feature explicit revocation (○), yet we still observed three different approaches for revocation.

Some capability systems use revocation solely for reclaiming resources (**R**). In these systems, revocation means losing the data in the designated object. Conceptually, all capabilities designating the object[6], including the capability used for revocation, are invalidated, and a new one is created. A second type of system features a revocation mechanism that retains the data for the subject performing the revocation (**D**). In these systems, the capability used for revocation is preserved, while all others designating the same object are invalidated. We also observed a system with targeted revocation, where the capability is not globally invalidated (including all delegations) but only a specific delegation is revoked (**T**).

C15 Exclusive Access

In systems that feature exclusive access, the subject holding a capability can ensure that it has exclusive access to the object for the time being. This represents a powerful primitive that can be used to achieve a high level of trustworthiness. We observed four distinct flavors of this property, even if the majority of works does not feature it (○).

Systems with *concurrency-inspired* exclusive access (**C**) use locks to achieve temporary exclusive access. On the other hand, systems with *destructive* exclusive access (**D**) use revocation with data retention (§C14: **D**). This ensures that the subject requesting exclusive access is the only one with a capability to the object, blocking all others from accessing it. The only way to lift the exclusivity is to re-delegate the capability, a trait shared with systems with *initial* exclusive access (**I**). In such a system, subjects are guaranteed to have exclusive access to objects until they delegate their own initial

[4] We do not differentiate between processes and threads here

[5] After bootstrapping, which requires some form of universal authority.

[6] In memory, overlapping objects have to be considered as the same object

capability. The last technique we observed are *linear capabilities* (**L**). These capabilities cannot be copied, only moved. Thus, at any time, only one capability can exist for a given object, ensuring exclusive access.

C16 Cooperative Access

It can be desirable to provide authority over objects in a way that requires the cooperation of two subjects.[7] In the literature, we often observed this realized as *type managers* that have access to the internals of objects of a certain type, whereas non-type-manager subjects can only designate the object. [27, 74, 98]. Conceptually, a type manager holds a meta-capability specifying the rights but not the object they may be exercised on. Split capabilities [46] may seem to be related but are orthogonal: They split object references and rights into two separate tokens for easier policy expression.

5 Taxonomy of Capability Systems

Applying the characteristics introduced in §4 to analyze the works identified in §3.3 reveals three capability system archetypes (see Table 1). Besides the known *Ticket Capability Systems*[8] (§5.1) and *Object Capability Systems* (§5.2), we additionally identified *IPC Capability Systems* (§5.3).

Observation 3

It is important to note that two systems exhibiting the same characteristics does not necessarily mean that they are similar systems. Capabilities have been used for a multitude of purposes [20,72,90,97]. This influenced not only their goals, but also the types of subjects and objects governed by capabilities in each system. Accordingly, we had to analyze the systems based on their defined scope. All other results would be highly speculative.

5.1 Ticket Capability Systems

Capabilities in such systems behave much like real-world tickets.[9] This means that they can be copied and shared freely between subjects, without pre-existing channels (§C5: ○).

They still aggregate authority at the subject (§C1: ●) and typically support designation only through capabilities (§C2: ●). While it is feasible for them to achieve composability, we did not observe this in the literature (§C4: ○). Still, most systems lack ambient authority (§C3: ●).

Due to their defining property, the similarity to real-world tickets, ticket capabilities are always probabilistically protected. Commonly, this is done using atomic capabilities (c.f. Table 1), but probabilistic handles are possible (§C7: **A** ∨ **P**)

[7] A form of Separation of Duties

[8] A.k.a. *capabilities as keys* [71] or password capabilities [6, 72]

[9] Old-timey tickets without copy protection like bar- or QR-codes.

The lack of explicit delegation makes policy controlled delegation and protection domain tracing impossible without a reference monitor (§C8: ○, §C9: ○). The same holds for cooperative access (§C16: ○): If the spread of authority cannot be limited, it is futile to require two subjects to cooperate. However, probabilistic protection mechanisms facilitate persistence and context independence (§C13: ●, §C6: ●).

Ticket capability systems are often constructed on top of existing systems. Consequently, we never observed single level storage (§C11: ○) in ticket capability systems. This also leads to their typical minimal subject unit being a process (§C10: **P**), and explains the lack of ringless systems (§C12: ○) and exclusive access primitives (§C15: ○).

Explicit revocation is not possible in the general case (§C14: ○) but can be achieved using a reference monitor. A changeable version number associated with an object could be considered a revocation primitive (§C14: **D**). However, it is equally expressive as copying the object and deleting the old version, which is why we do not consider it *explicit*.

Observation 4

Ticket Capability System are the easiest type of capability system to implement, as they do not require any hardware or operating support. This serves as both their greatest strength and weakness: Many considerations that are required to fulfill more complex characteristics can be eschewed. But many of the more complex characteristics cannot be achieved.

5.2 Object Capability Systems

Object Capability Systems are required to feature subject aggregated authority, designation only through capabilities, no ambient authority, and are able to achieve composability (§C1: ● ∧ §C2: ● ∧ §C3: ● ∧ §C4: ●) to count as such. This has been widely accepted at least since Miller et al. [71] specified it in their endeavor to refute myths about capability systems.

Explicit Delegation (§C5) requires further discussion. While a definitively marked *no* (§C5: ○) allows for arbitrary flow of privileges in the system and disqualifies a system from being considered an object capability system, we distinguish three levels for this property.

Both other levels (§C5: ◐ ∨ ●) require the delegation to be access controlled. Thus, they clearly achieve confinement and are equivalent when it comes to deciding whether or not a system is an object capability system

To understand why such a fine-grained distinction is still necessary, consider the *-property and simple-security property [21, 53]. They originated in the military and state that an entity should only be able to write to its level or higher levels and read from its level or lower levels. Naturally, as the policy comes from the real world, it is concerned with

	Group & Representatives	Years	Designation Only §C1	Subject Aggregated Authority §C1	No Ambient Authority §C2	Through Capabilities §C2	Context Independent §C3	Composability §C3	Explicit Delegation §C4	Capability Representation §C4	Policy Controlled §C5	Protection Domain §C5	Delegation §C6	Tracing §C7	Minimal Subject Unit §C8	Single Level Storage §C9	Ringless System §C10	Design §C10	Explicit Revocation §C11	Persistence §C12	Exclusive Access §C13	Cooperative Access §C14	§C15	§C16
Object Capability Systems	**The Original Capability Paper**																							
	Supervisor [20]	1966	●	●	●	●	●	○	H	○	○	P	●	○	○	T	C	○						
	Early Ringless Architectures																							
	System 250 [24]	1972	●	●	●	●	●	●	S	○	○	M	●	●	○	○	○	○						
	PSOS [74]	1975	●	●	●	●	●	●	T	○	○	M	●	●	○	○	I	●						
	Software Type Managers																							
	HYDRA [98, 99], Cal [52], StarOS [41], Cm* [42]	1974–1981	●	●	●	●	●	○	H	○	○	M	●	○	○	○	○	●						
	Hardware Type Managers																							
	Intel 432 [39, 39, 76]	1981–1984	●	●	●	●	●	●	T	○	○	M	●	○	●	○	○	●						
	iMAX [44, 78]	1981–1983	●	●	●	●	●	●	T	○	○	M	○	○	●	○	○	●						
	Providing Persistence																							
	CAP [96]	1979	●	●	●	●	●	○	S	○	○	M	●	○	●	T	○	○						
	Grasshopper [19, 82]	1994–1996	●	●	●	●	●	○	H	○	○	M	●	○	●	T	○	○						
	EROS [84, 85]	1999–2000	●	●	●	●	●	○	H	●	●	P	●	○	●	T	○	○						
	Tiling Architectures																							
	M^3 [9]	2016	●	●	●	●	●	○	H	○	●	P	○	○	○	D	D	○						
	SemperOS [8, 36]	2019	●	●	●	●	●	○	H	○	○	P	○	○	○	D	D	○						
	Modern Memory-Tagged Capability Systems																							
	CHERI BSD [94]	2015	●	●	●	●	●	●	T	○	○	M	○	○	○	○	○	●						
	CheriOS [25]	2020	●	●	●	●	●	●	T	○	○	M	○	◐[A]	○	R	I	●						
	Linear Capabilities																							
	Capstone [101]	2023	●	●	●	●	●	●	T	○	●	M	○	●	○	R	L	○						
	Probabilistic Handles																							
	Northcape [3]	2024	●	●	●	●	◐	●	P	○	○	M	○	●	○	R	C	○						
	RV-Cure [47]	2025	●	●	●	●	◐	●	P	○	○	M	○	○	○	R	○	○						
	Optional Capabilities[B]																							
	IBM System/38 [11]	1980	○	○	○	○	○	●	T	○	○	U	●	○	●	T	○	○						
	C^3 [56]	2021	○	○	○	○	○	●	A	○	○	P	○	○	○	○	○	○						
	Ticket Capability Systems																							
	Amoeba [72, 89], Password Caps [6]	1981–1990	●	●	●	○	○	●	A	○	○	P	○	○	○	○	○	○						
	Capacity [22]	2023	●	○	○	○	●	○	A	○	●	M	○	○	○	○	○	○						
	IPC Capability Systems																							
	Mach [2, 43], Accent [81], seL4 [49], Capsicum [93], DroidCap [18], Fuchsia [62]	1986–2024	●	●	●	○	●	○	H	○	○	P	○	○	○	○	○	○						
	The Gutenberg Operating System [16]	1986	●	●	●	○	●	○	H	○	●	P	○	○	●	T	○	○						

[A] A potential ringless design is discussed. However, the current design still has a privileged *nanokernel*.

[B] In both System/38 and C^3, the capability mechanism can be bypassed with legacy addresses. Otherwise, they would feature more characteristics.

Table 1: Taxonomy of capability systems, featuring older, well-studied works, and recent works with novel design combinations. We systematize 31 works spread over 3 archetypes. Notably, the characteristics exhibited by systems in one group are very similar with only minor differentiations.

data, not capabilities. If we distinguish capabilities and data on a system level, the policy is easy to realize. However, in case delegation is not explicit (§C5: ◐), data and capabilities are not considered separately. Accordingly, a low privileged entity could use its *write-low* capability to write the same capability into a *low* segment. Following the policy, a *high* entity can now use its *read-low* capability to read the *write-low* capability. From that point forward, the *high* subject can now write data to the *low* segment, violating the *-property and thus the policy.

While Miller et al. [71] argued that *any* object capability system would be able to achieve the *-property, we observed novel systems that allow delegation only over authorized channels, thus counting as object capability systems, but still violate the *-property. Therefore, delegation over any access controlled delegation channel (§C5: ◐) is sufficient for systems to qualify as object capability systems, whereas explicit delegation (§C5: ●) is required to comply with the *-property. We observed both variants of the property in the literature.

Our taxonomy in Table 1 shows that despite these common characteristics, the variety of object capability systems is huge. Commonly, the minimal subject unit is defined as a process or smaller (§C10: **P** ∨ **M**). Single level storage used to be a common characteristic, but has vanished as system design has evolved (§C11: ● → ○). A similar observation can be made about capability protection. As memory safety became increasingly important, handles were often replaced by memory tagging, although both are common (§C7: **H** ∨ **M**). In conjunction with the rising number of memory focused capability architectures came a shift in explicit revocation techniques. Away from targeted revocation to remove privileges towards a reclamation centered approach to free resources (§C14: **T** → **R**). Context independence was also largely dictated by the chosen protection mechanism (§C6: ● ∨ ○). Exclusive access and ringless system designs follow a different pattern. While their are some early examples, their renaissance is quite recent (§C15: ●, §C12: ●). All other characteristics have mostly fringe occurrences (§C8: ○, §C9: ○, §C13: ○).

Observation 5

The prevalent lack of policy-controlled delegation in object-capability systems (§C8: ○) might seem surprising. However, a probable explanation lies in the characteristics all object-capability systems share. The combination of *Designation Only through Capabilities* (§C2: ●) and *Explicit Delegation* (§C5: ◐ ∨ ●) means all sharing can only occur over preexisting capability relationships. Accordingly, there are only three ways to obtain a capability: *creation*, *inheritance*, and *delegation*. This limits any one computation to access only objects it created itself (*creation*), got delegated by its parent (*inheritance*) or by another computation. Furthermore, if §C5: ● is given, *confinement* can be achieved [51,60,71].

5.3 IPC Capability Systems

These systems rely on capabilities to protect inter-process communication and remote procedure calls. Heavy reliance on IPC communication is typically associated with *"appified"* mobile systems, e.g., smartphone [61] and IoT [62] systems. However, we also observed older systems that used *ports* [2,16] in essentially the same way, mostly for distributed systems.

These systems stand out because the objects they consider—IPC and RPC channels—are their defining features. By their very design to focus on IPC protection, they are hybrid capability systems, i.e., they mix capability-based access control for IPC with other forms of access control for everything else. We believe that, given mobile systems' ever-increasing importance and their inherent reliance on tremendous amounts of IPC connections, we may see even more of this type of capability system in the future.

When not viewed holistically, but tailored to their scope of focusing on IPC endpoints as objects, IPC capability systems exhibit most characteristics seen in object-capability systems. They have subject aggregated authority, no ambient authority, allow designation only through capabilities and feature explicit delegation (§C1: ● ∧ §C2: ● ∧ §C3: ● ∧ §C5: ●). The lack of ambient authority does not mean that addressing *all* objects in the system is done without ambient authority, only those that can be referenced using capabilities[10].

IPC capability systems *cannot* feature composability (§C4: ○). It would require treating subjects as objects. As all objects are IPC endpoints, this is impossible.

Observation 6

However, IPC endpoints cannot exist in a void. They are dependent on the process at the other end of the IPC-channel. This makes composability a curious case, as it is possible to construct a tree-like structure of processes connected by IPC that resembles composability.

Their capabilities are typically protected as handles (§C7: **H**). This makes for an easier integration with established systems and eliminates context independent representation (§C6: ○).

Ringless system design is generally not possible, as the IPC primitive requires a privileged kernel (§C12: ○). The same holds for exclusive access. As IPC endpoints are part of a channel, there is always at least one other subject with access (§C15: ○).

All other characteristics can in theory be fulfilled, even if we did not observe them in practice. The only exception is the Gutenberg Operating System [16], which employs protection domain tracing (§C9: ●) to achieve targeted revocation (§C14: **T**). This is a more pressing concern in the system, as in can persist its capability delegations (§C13: ●).

[10] A concession we had to make to compare six decades worth of systems

6 Open Challenges

Over the decades, a large variety of capability systems have been designed. Still, given the flexibility of capabilities as a mechanism, the design space is far from explored.

In fact, we noticed that capability systems tend to appear in groups. Either, they share a common goal, as the group of type manager did that we observed at the end of the 1970s [41, 42, 52, 98, 99]. Or, the literature revolves around a single design idea, as is currently the case with CHERI [97] and its descendants [4, 5, 23, 25, 28, 65, 77, 90, 94, 95, 100]. The sheer number of works listed here clearly shows a concentration on one mechanism. While we welcome the increased interest in capability systems, such a cluster will invariably cover less of the design space than independent ideas.

However, exploring the design space is essential in tackling the numerous challenges still open in capability systems research and moving the field forward. In the following, we discuss open challenges for capability system designs that we believe can contribute to this endeavor.

6.1 The Devil is in the Details

While the concept of a capability, the liaison of designation and authority, is very simple, designing systems around them is not (§2). The array of characteristics we presented shows how vast the design space really is. And while we introduced them separately, if possible, there are dependencies between them that are not yet fully explored. For example, the choice of capability protection mechanism has implications for other characteristics. It dictates whether a ringless system design is possible (§C7 $\implies$ §C12) and if explicit delegation can be achieved (§C7 $\implies$ §C5). Furthermore, it can preclude context independent representation (§C7: **H** $\implies$ §C6: ○) and guarantee explicit delegation (§C7: **H** $\vee$ **T** $\implies$ §C5: ●). Another example is cooperative access, which requires composability (§C16: ● $\implies$ §C4: ●).

These dependencies arise before even considering the scope of objects and subjects. We already discussed that IPC endpoints as objects preclude composability (§C4). However, subjects can just as easily influence properties. Systems with method invocation level subject domains cannot use handles as a protection mechanism (§C10: **M** $\implies$ §C7: $\neq$ **H**).

Theses are just some of the implications that might not be immediately obvious.

Challenge 1: Realizing Desired Characteristics

Capabilities really are deceptively simple. However, as the identified interdependencies show, even seemingly small design decisions can have big ramifications for the entire system. Combining the desired characteristics might even necessitate coming up with extensions to techniques present in the literature that eliminate some mutual exclusivity. Moving forward, it is essential to identify these interdependencies and to analyze them in depth when designing new systems.

6.2 Varying TCB

We observed vastly different choices regarding the trusted computing base (TCB) in capability systems. Naturally, systems that protect their capabilities as handles (§C7: **H**) will include the (monolithic) kernel in their TCB [84, 98]. In hardware based capability systems, trust in the kernel can be limited or completely avoided [25]. However, safe for a few exceptions [3, 8, 9], they still assume all hardware to be part of the TCB. Even ringless systems (§C12) may have a software TCB such as the bootloader [3] or interrupt handlers [90], requiring analysis at a case-by-case basis.

Challenge 2: Reducing the TCB

As capability systems are increasingly implemented on capability enhanced hardware, the trusted computing base keeps changing. Assessing it is not always easy, especially if characteristics known from trusted computing are desired. However, just like compartmentalization (in combination with capabilities) can reduce the necessary trust in code, reducing the TCB can help reduce the necessary trust in the system. This could even be combined with attempts to construct trusted execution environments from capabilities [90, 101] or enhancing capability systems with confidential computing features.

6.3 Contradicting Security Models

One may wonder why System/38 was not awarded §C1 through §C3 despite being a commercial object capability system. This is a symptom of a broader issue in capability systems: security models differ vastly.

Consider System/38 [37] as a case study. System/38 provides both *authorized pointers* and *unauthorized pointers*. Authorized pointers are the *frontend* of a tagging-based object capability system (§C7: **T**). Unauthorized pointers are simply pointers that resolve into an object name. Access control on unauthorized pointers occurs in the form of *mandatory access control* using the running process' security profile.

Crucially, System/38 allows using *authorized pointers and unauthorized pointers at the same time for the same object*. This creates a dilemma for our analysis: using *authorized pointers only*, System/38 could achieve §C1: ● through §C3: ●. However, the use of *unauthorized pointers* makes it possible to circumvent these properties, making the capability architecture essentially opt-in for each object. At the same time, using *unauthorized pointers only*, System/38 could achieve §C14: **D**. The ability to circumvent the capability system at will makes it difficult to honestly compare System/38

with more modern approaches, where this is not possible. Thus, we chose not to award §C1 through §C3.

Challenge 3: Complete Mediation

Some systems allow *circumvention* of all capability protections by allowing simultaneous addressing of the same object using capabilities and other means. This can be a justifiable design choice if some part of the TCB needs a different way of access. Capabilities protected by the kernel constitute one such case. However, access controlled by capabilities should *not* be circumventable by subjects that can use capabilities.

6.4 Avoiding Confused Deputies

Hardy's aptly titled work *"The Confused Deputy: (or why capabilities might have been invented)"* [35] already highlighted in 1988 how problematic confused deputies can be. Deputies are a common occurrence in system design to ensure safe and correct usage of resources. They intentionally use their authority on other processes behest. In Hardy's scenario, the correct authority is used on the wrong object. Confusion for such a deputy is avoidable by eliminating ambient authority (§C3) and forcing designation to use capabilities (§C2). This allows the deputy to differentiate between the caller's and its own authority, preventing confusion. Furthermore, the deputy is not forced to use its entire (ambient) authority on each operation. Note that the two characteristics ease the realization of developers to write unconfusable deputies, but cannot inherently prevent badly designed deputies from becoming confused [71].

The addition of middlewares full of intentional, privileged deputies to modern operating system stacks [33, 38] has exacerbated the predicament. In fact, we argue that Hardy's classic deputy and modern, *privileged deputies* found in middlewares need to be considered separately. Since the latter are designed to be privileged gateways for sensitive resources, the threat model is very different.

Whereas in Hardy's case, the compiler is a tool to be used within the client's authority, deputies in middlewares are higher privileged processed that encapsulate sensitive resources and offer an API to untrusted clients. Giving the capability to the deputy alone would not mitigate the risk of confusion, giving it to the client instead would inherently allow it to bypass the deputy entirely. Neither one is acceptable. Accordingly, avoiding ambient authority (§C3) and designation only through capabilities (§C2) is insufficient to eliminate their confusion.

In consequence, the required access rights need to be split between the client and the deputy so neither can act alone. This ensures proper encapsulation while allowing deputies to be built in a unconfusable way and is supported in system designs that support cooperative access (§C16: ●).

Different technical realizations of such capabilities are imaginable. In the literature, deputies employing this design are commonly referred to as type managers [98].

Challenge 4: Preventing Modern Confused Deputies

No ambient authority (§C3) is a prerequisite for capability systems as Definition 5 requires *explicit use* of capabilities. In combination with *All Designation Through Capabilities* (§C2), this allows avoiding deputies that could operate within the bounds of the caller's protection domain (e.g., tools or libraries). However, modern deputies are designed as privileged gateways outside these bounds. Addressing confused deputies requires cooperative access (§C16) in the system design, which is currently under-researched. In particular, in systems with extensive middleware, this is a promising step. It could lead to a dramatic reduction or even elimination of confused deputy vulnerabilities in these systems.

6.5 Limitations of the Access Control Matrix

The access control matrix is a well known, textbook visualization of access control decisions on a system level [7, 13, 87]. Generating the set of capabilities a subject should hold is commonly described as reading the corresponding row in the matrix. However, this is not as simple. In a capability system with composability, subjects are objects, too. Accordingly, they should appear as labels for columns and rows. While Butler Lampson's introduction of the capability matrix [50] made note of that, most newer textbooks oversimplify here and ignore this duality of subjects [7, 87].

The way capabilities are distributed in a system (see §2.3) complicates this even further. They are not assigned globally based on the identity of the subject. Instead, they are delegated from one subject to another. In a system without policy controlled delegation, this means that every subject which holds a capability to an object acts as a policy decision point for said object, as it could delegate its own capabilities at its own discretion. This can even occur in a circular way, leading to authority distributions that cannot be represented in an access control matrix.[11]

Challenge 5: Beyond the Access Control Matrix

The well known notation of policies as an access control matrix is not particularly suited for capability systems. However, a way of visualizing access control state is essential to developer understanding of policy decisions. We argue that a novel, more expressive representation of access control is required for capability systems.

[11] At least not a 2D version humans can comprehend

6.6 Capability Policies

As Wulf et al. [98] argue, capabilities can provide protection but not necessarily security. Since security must be defined according to a policy, it cannot be evaluated solely based on capabilities unless they implicitly encode a policy. However, most works focus on providing a *mechanism* through which a policy can be enforced and leave the *policy* itself to the application developer. While statements like

> [EROS [84]] *Capabilities are only given to processes that are authorized to hold them.*

are very open about this decision, they also highlight the fundamental problem: Without a clear policy, one can mostly just speculate about the potential security benefits.

The problem of research producing new mechanisms for solving a security problem while ignoring policy as an orthogonal problem is not unique to capability research.

For example, a related problem exists in software compartmentalization research:

> [Lefeuvre et al. [55]] *Because software compartmentalization is such a large and complex problem, [policy, abstractions and mechanism] are rarely, if ever, solved as a whole.*

The potential fine-grained nature of both subjects and objects (§C10), as well as the composability of authority (§C4) exacerbate the policy issue. Designing a security policy is significantly more complex than in more coarse-grained, non-composable systems where the policy may only cover users.

It might seem appealing to use systems with policy controlled delegation (§C8) to check each delegation against a policy and thus mostly ignore the inner workings of subjects. However, this immediately poses other problems. As capabilities are prima facie evidence of authority, there is no inherent need for any operation, including delegations, to include the identity of the caller. Thus, the policy would need to act solely on the capability itself and the identity of the callee. Depending on the unit of protection domain, the callee might not have a clear identification either.[12]

In consequence, representing capability policies is an unsolved problem.

Challenge 6: Representing Capability Policies

As delegation is simply an operation, investigating the expression of policy directly in code could be an avenue of research. If all access is governed by capabilities, then the superset of all capability delegations can be read as the entire access control policy: To our knowledge, such fine-grained policies expressed through the program code have not been the subject of research in the systems domain. However, it can offer beneficial ways of dealing with the complexity of managing capabilities throughout the entire software stack.

[12] At least not for the purpose of policy enforcement.

6.7 Comparing the Incomparable

Due to their ubiquity and the assumed familiarity of readers, access control lists are often used as a basis for comparisons with and explanations of capabilities.

Challenge 7: Incorrect Mental Models

The comparison of access control lists and capabilities, is inherently flawed. Capabilities are a *mechanism for access*. Access control lists are a form of *policy expression* for a single object. Capabilities are self-sufficient for access, as they combine designation and authority. Access control lists are essentially nothing more than lookup tables. We could go on, but suffice it to say they are not directly comparable. However, this is rarely mentioned in the literature. The current slew of capability systems in different areas makes it essential to foster correct mental models about capabilities in the research and education.

6.8 Revocation $\neq$ Revocation

Revocation has been mostly discussed as a way of rescinding a delegation. However, we observed revocation for multiple purposes in recent works. Some systems utilize revocation as a form of exclusive access. While this still boils down to removing previously delegated rights, the intention is different. For example, a family of CHERI descended works strives for temporal memory safety and to protect against use-after-free vulnerabilities [4, 5, 28, 95, 100]. Capstone [101] on the other hand provides exclusive access through linear capabilities that can be moved or split, but not copied. Their revocation mechanism enables the delegating process to reclaim the previously delegated memory *without* revealing the content. Northcape [3] has a similar design. It allows locking of memory for exclusive access and uses revocation as reclamation of locked segments.

Challenge 8: Repurposing Revocation

Revocation, reclamation, and exclusive access are interconnected and often used to solve similar problems, which needs to be taken into account. These new forms of revocation, most prominently exclusive access and its challenges, like reclaiming resources without sacrificing confidentiality guarantees, were rarely studied and deserve more attention.

6.9 To revoke, or not to revoke?

Our systematization shows that explicit revocation has not been a highly sought-after characteristic (see Table 1). Nevertheless, it has been discussed repeatedly in the literature. Both whether it is possible [15] in the general case and how it should be implemented [32, 45, 84] was debated.

Seeing how little adoption it has found in past systems, we believe that the more important question is whether *explicit* revocation is required at all. However, we need to distinguish the reasons for revocation we observed. Revocation in the traditional sense of retroactively removing privileges from a subject is what we discuss here. By design, this means that the subject was overprivileged at one point, i.e., a violation of the principle of least privilege. While it is necessary to pass capabilities to other subjects, and revocation has been named to ensure that this authority cannot be delegated indefinitely, the literature proposed alternative designs. Both Hydra [98], with its combination of templates and non-retainable capabilities, and CheriBSD [94], with their locked capabilities that can be used for domain switches, provide alternative ways of limited-time delegation. Their design is similar in effect to Linden's proposal to restrict capabilities to exist on the stack only [60]. Both primitives create a mechanism for delegating non-retainable capabilities. Accordingly, they can be shared with another protection domain without giving that protection domain permanent access to the underlying resource. Instead, it may typically only access the resource for the duration of a single invocation, creating a primitive for controlled and limited transfer of authority. Additionally, non-retainable capabilities are considerably simpler than the typically complex or expensive logic for revocation that can include intrusive techniques, like sweeping the entire memory [28, 95, 100].

The CAPSTONE architecture proposes linear capabilities as a different solution to temporary delegation, providing different tradeoffs [101]: Linear capabilities are capabilities that can be *moved* using capability operations but never *copied* [86, 91]. This approach forgoes revocation completely (§C14: ○) by making exclusive access the default (§C15: **C**).

Challenge 9: Selecting a Revocation Strategy

Revocation as a form of resource reclamation can be necessary in some systems. However, developers of new capability systems should contemplate whether they need explicit revocation in the traditional sense. Other approaches that limit the retention of capabilities are often simpler and can be more secure.

6.10 Backward Compatibility

To further adoption, many of the discussed capability systems implement some form of backward compatibility [52, 64, 72, 94]. Many other systems retrofit existing operating systems rather than designing a capability-first OS [18, 93, 94].

However, backward compatibility always weakens the capability model, and sticking with existing systems limits flexibility. David Wheeler famously said:

> [Wheeler [17]] *Compatibility means deliberately repeating other people's mistakes.*

While the compatibility layers typically only violate least privilege and increase the size of compartments, CHERI's binary compatibility mode bypasses the capabilities entirely.[13]

Challenge 10: Building from Scratch

Even the systems built with backward compatibility in mind have not seen widespread use so far [62, 72, 94, 97]. It might be time to break with established standards and research novel capability systems without the crutch of compatibility. Given capability systems ability to provide byte-granular memory safety, this could even end the *war in memory* as a side effect [88].

6.11 Evolving Security Necessitates Redesign

The potential issues presented by backward compatibility are only exacerbated by the evolving security needs of today's systems. Capabilities were invented at a time when protecting users from each other was the most common goal. However, the landscape of threats has clearly evolved, necessitating a stronger and more fine-grained protection mechanism for critical computations and data. Over the last decades, a plethora of solutions has evolved to compartmentalize software [55]. They create trust boundaries both horizontally (e.g., between different apps, between first and third party code, or between different kernel components) and vertically (i.e., between critical computations and the underlying OS). Typically, they are stacked on top of each other and all rooted in the concept of access control lists [59, 63].

Retrofitting systems with more strict or fine-grained access control mechanisms often leads to semantic gaps between individual solutions. Furthermore, splitting existing software into separate compartments for least privilege can lead to vulnerabilities at the newly created interfaces [54].

We think that this is an opportunity where capability systems could reign vastly superior over those with access control lists in reducing trust and creating security boundaries. Since capability systems feature strong encapsulation only broken by capabilities, less trust is required for third-party code. Such code is guaranteed to run in isolation, and even if a security boundary is breached in a controlled way, the lack of ambient authority makes it easier to use authority in the correct context (see §6.4). Additionally, if the system is implemented without privilege rings (§C12), less trust needs to be put into the system software as it has no kernel-like privileges but

[13] For our analysis, we skipped this functionality and only considered the capability mode operation.

is limited by capabilities. Moreover, as we have seen with CHERI [94] and its successors [77, 90, 95, 100], hardware capability systems can offer byte-granular memory safety. This, in turn, reduces the trust a developer is required to have in their own code, as memory corruption vulnerabilities are eliminated. CHERITrEE [90], CHERIoT RTOS [4] and CAPSTONE [101] even demonstrate how isolated computation functionality can be built from capability primitives rooted in hardware.

Challenge 11: Unified Compartmentalization

Capabilities are often dismissed as too hard to manage, too complicated and impossible to adopt into modern systems. However, the ever-increasing number of security features, layered on top of existing systems, clearly shows a need for more comprehensive security. To phrase it drastically, we are developing bigger and bigger buckets instead of fixing the leak in the ship.

We think a fundamental rethinking of how capabilities could provide a unified form of access control on top of highly compartmentalized systems could be a high-effort high-gain endeavor. A fully vertical and horizontal capability system, where all entities, subjects as well as objects, are as fine-grained as possible, has yet to be developed. Accordingly, research on utilizing capabilities throughout all layers of the system and compartmentalizing software, forms a formidable challenge.

7 Conclusion

There are countless more characteristics that we could have differentiated capability systems on (e.g. the capability ownership model). However, a disambiguation on a more fine-grained level would preclude us from analyzing differences present in over six decades of literature adequately. Furthermore, some characteristics we primarily observed in newer works are understudied. This reduces the generality of conclusions we could draw from separating them further. Ultimately, the systematization is tailored toward the goals we set in the beginning and naturally contain a certain subjective bias. We encourage the community to extend and refine it to accommodate future system designs.

The future, as Lampson put it (§1), is not yet here. Contemporary approaches for secure computer architecture often focus on reducing the TCB. However, research is starting to realize that TCB size might matter less than interface security in highly compartmentalized systems [31, 54]. Capabilities are considered one of the possible solutions. Furthermore, this provides potential for seamless integration across different application domains. These trends lead us to believe that, while the future is not there yet, it is getting closer. We envision this first holistic systematization of capability operating systems to aid in that journey towards a pure object capability system across all system layers.

Ethical Considerations

We identified four different groups of stakeholders that could be affected by our work. Conducting the work can impact the research team conducting the analysis and the scholarly services we queried for our citation tree.The results on the other hand can impact the authors of the papers we systematized, as well as the readers of our SoK.

The Research Team

Analyzing hundreds of papers can put a lot of strain on anyone. To ease the burden of the research team, we automated as much of the filtering as possible. Despite some hurdles, like keyword filtering for *capability* being almost entirely futile, we managed to reduce the dataset to a manageable ≈ 500 works, down from $\approx 1,000,000$.

The Scholarly Services

Retrieving information from scholarly sites at large quantities is demanding on the infrastructure. Furthermore, not all scholarly website allow such an automated retrieval.

To avoid violating their terms of services, we omitted Google Scholar as a source. They prohibit automated retrieval at the scale we require.

Instead, we used the Semantic Scholar API [48] and OpenAlex [80]. In both cases, we reduced the load on their infrastructure as far as possible. For semantic scholar, we requested an API key and limited our queries to the rate limit assigned with it. For OpenAlex, we set up a local copy of their database and performed all queries on our own infrastructure.

The Authors of Systematized Works

Both during the selection process and in representing them, the authors of prior work could be impacted. While filtering papers, it would have been possible to focus on clusters written by the same author, if we identified them as a "capability researcher". However, doing so might discriminate against authors of lesser-known works who potentially only had this single foray into the area of capabilities. To avoid this, we filtered the works based on the structure of the citation graph and content only, without taking authors into account.

Furthermore, in representing a work in our taxonomy (Table 1), we inherently shape discourse about it. Thus, fairness, accuracy and impartiality are paramount. We strive for these goals and thus had every work considered independently by two researchers, before a discussion lead to the unanimous choice of representation.

The Reader

We recognize that a Systematization of Knowledge has the potential to shape the research area it is conducted in for the foreseeable future. Accordingly, it is important to consider the impact on readers as well. However, readers benefit from the same considerations put towards authors of past work: A fair, accurate and impartial representation of information. Accordingly, the same steps taken apply here as well.

Open Science

Code

The code for our paper collection is hosted on Zenodo [66] (`https://doi.org/10.5281/zenodo.20435520`). The options for the tool are documented when passing the `-help` option.

Data

We cannot provide the dataset of `.pdf` or metadata files. Many pdfs were obtained from the respective publishers using a campus license and can thus not be made publicly available. However, it is possible to recompute the dataset using our provided code by utilizing the semantic scholar API and querying it for our seed papers. The resulting `.json` files containing metadata about the papers will have a url for most pdfs. These can include open access versions as well as paywalled links.

In order to do so, use the following command line:

Expected Runtime: $\approx$ 1 week

```
python main.py crawl -l 2 -r 1 full_dataset
    21678be84430f56942cf5172c281b1861b9ac7a0
    22dd6d79baf65f26364e42b3d5389b8ca3b139bc
    04a409c6fd63f65c308be8865211bb576b213515
    e5fcec0cf2b59aee0365079362b90782f52694ad
    5b18397474dea51b92d406ff0b9f517da37956be
    df18e9c9498a497f1d5063de220f9edd0c0b3fc6
    5a89b3c48d50b003569876939b52fbd76b6d3361
    b42d2f71c3369cbb8634c263a3b792ddf443f638
```

References

[1] *Capabilities(7) - Linux Manual Page*, 2024.

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

[3] Eric Ackermann, Noah Mauthe, and Sven Bugiel. Work-in-Progress: Northcape: Embedded Real-Time Capability-Based Addressing. In *IEEE European Symposium on Security and Privacy Workshops*, 2024.

[4] Saar Amar, Tony Chen, David Chisnall, Nathaniel Wesley Filardo, Ben Laurie, Hugo Lefeuvre, Kunyan Liu, Simon W. Moore, Robert Norton-Wright, Margo Seltzer, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. CHERIoT RTOS: An OS for Fine-Grained Memory-Safe Compartments on Low-Cost Embedded Devices. In *SOSP '25*, 2025.

[5] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. CHERIoT: Complete Memory Safety for Embedded Devices. In *MICRO '23*, 2023.

[6] M. Anderson, R. D. Pose, and C. S. Wallace. A Password-Capability System. *The Computer Journal*, 29(1), 1986.

[7] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 2020.

[8] Nils Asmussen, Till Miemietz, Sebastian Haas, and Michael Roitzsch. Distrusting cores by separating computation from isolation. *Journal of Systems Architecture*, 159, 2025.

[9] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *ASPLOS '16*, 2016.

[10] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *SOSP '09*, 2009.

[11] Viktors Berstis. Security and protection of data in the IBM System/38. In *ISCA '80*, 1980.

[12] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Capabilities for information flow. In *PLAS '11*, 2011.

[13] Matt Bishop. *Computer Security: Art and Science*. 2018.

[14] W. E. Boebert. On the Inability of an Unmodified Capability Machine to Enforce the *-Property. In *DoD/NBS Computer Security Conference*, 1984.

[15] A. Chander, D. Dean, and J.C. Mitchell. A state-transition model of trust management and access control. In *CSFW '01*. IEEE, 2001.

[16] Panayiotis Chrysanthis, Krithi Ramamritham, David Stemple, and Stephen Vinter. The Gutenberg operating system kernel. In *ACM Fall Joint Computer Conference*, 1986.

[17] Piotr Cofta. *Trust, Complexity and Control: Confidence in a Convergent World*. 2007.

[18] Abdallah Dawoud and Sven Bugiel. DroidCap: OS Support for Capability-based Permissions in Android. In *NDSS '19*, 2019.

[19] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing systems*, 7(3), 1994.

[20] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3), 1966.

[21] Department of Defense / National Computer Security Center. Trusted computer system evaluation criteria. Technical Report DoD 5200.28-STD, 1983.

[22] Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures. In *CCS '23*, 2023.

[23] Michael Dodson, Alastair R. Beresford, Alexander Richardson, Jessica Clarke, and Robert N. M. Watson. CHERI Macaroons: Efficient, host-based access control for cyber-physical systems. In *IEEE European Symposium on Security and Privacy Workshops*, 2020.

[24] D. M. England. Architectural features of system 250. In *State of the Art Report on Operating Systems*, volume 14. 1972.

[25] Lawrence Esswood. *CheriOS: Designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor*. PhD thesis, University of Cambridge, 2020.

[26] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7), 1974.

[27] R. Feiertag and P. Neumann. The foundations of a provably secure operating system (PSOS). In *International Workshop on Managing Requirements Knowledge*, 1979.

[28] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert Norton, David Chisnall, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *ASPLOS '24*, 2024.

[29] Noam Gallmann, Pirmin Vogel, Pasquale Davide Schiavone, and Luca Benini. From swift to mighty: A cost-benefit analysis of ibex and CV32E40P regarding application performance, power and area. In *Workshop on Computer Architecture Research with RISC-V*, 2021.

[30] Aïna Linn Georges*, Armaël Guéneau*, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cerise: Program verification on a capability machine in the presence of untrusted code. *Journal of the ACM*, 71(1), 2024.

[31] Adrien Ghosn and Marios Kogias. Why I Stopped Caring About the TCB. In *SysTEX '25*, 2025.

[32] L. Gong. A Secure Identity-Based Capability System. In *S&P '89*, 1989.

[33] Sigmund Albert Gorski and William Enck. ARF: Identifying re-delegation vulnerabilities in Android system services. In *WiSec '19*, 2019.

[34] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System. *IEEE Micro*, 43(3), 2023.

[35] Norm Hardy. The Confused Deputy: (Or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4), 1988.

[36] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. SemperOS: A Distributed Capability System. In *ATC '19*, 2019.

[37] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM System/38 Support for Capability-Based Addressing. In *ISCA '81*, 1981.

[38] Sigmund Albert Gorski III, Seaver Thorn, William Enck, and Haining Chen. FReD: Identifying file Re-Delegation in android system services. In *USENIX Sec '22*, 2022.

[39] Intel Corporation. *Introduction to the iAPX 432 Architecture*, 1981.

[40] Mirek Jahoda, Barbora Ančincová, Ioanna Gkioka, and Tomáš Čapek. Understanding Domain Transitions: Sepolicy transition. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/security-enhanced_linux-the-sepolicy-suite-sepolicy_transition, 2024.

[41] Anita K. Jones, Robert J. Chansler, Ivor Durham, Karsten Schwans, and Steven R. Vegdahl. StarOS, a multiprocessor operating system for the support of task forces. In *SOSP '79*, 1979.

[42] Anita K. Jones and Edward F. Gehringer. The Cm* Multiprocessor Project: A Research Review. Technical Report CMUCS80131, Carnegie-Mellon University, Department of Computer Science, 1980.

[43] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. *SIGPLAN Not.*, 21(11), 1986.

[44] Kevin C. Kahn, William M. Corwin, T Don Dennis, Herman D'Hooge, David E. Hubka, Linda A. Hutchins, John T. Montague, Fred J Pollack, and Michael R Gifkins. iMAX: A multiprocessor operating system for an object-based computer. *SOSP '81*, 1981.

[45] Paul Ashley Karger. Improving security and performance for capability systems. Technical Report UCAM-CL-TR-149, University of Cambridge, Computer Laboratory, 1988.

[46] Alan H Karp, Guillermo J Rozas, Arindam Banerji, and Rajiv Gupta. Using split capabilities for access control. *IEEE software*, 20(1), 2003.

[47] Yonghae Kim, Anurag Kar, Jaewon Lee, Jaekyu Lee, and Hyesoon Kim. RV-CURE: A RISC-V Capability Architecture for Full Memory Safety. *IEEE Transactions on Computers*, 74(10), 2025.

[48] Rodney Kinney, Chloe Anastasiades, Russell Authur, Iz Beltagy, Jonathan Bragg, Alexandra Buraczynski, Isabel Cachola, Stefan Candra, Yoganand Chandrasekhar, Arman Cohan, Miles Crawford, Doug Downey, Jason Dunkelberger, Oren Etzioni, Rob Evans, Sergey Feldman, Joseph Gorney, David Graham, Fangzhou Hu, Regan Huff, Daniel King, Sebastian Kohlmeier, Bailey Kuehl, Michael Langan, Daniel Lin, Haokun Liu, Kyle Lo, Jaron Lochner, Kelsey MacMillan, Tyler Murray, Chris Newell, Smita Rao, Shaurya Rohatgi,

Paul Sayre, Zejiang Shen, Amanpreet Singh, Luca Soldaini, Shivashankar Subramanian, Amber Tanaka, Alex D. Wade, Linda Wagner, Lucy Lu Wang, Chris Wilhelm, Caroline Wu, Jiangjiang Yang, Angele Zamarron, Madeleine Van Zuylen, and Daniel S. Weld. The Semantic Scholar Open Data Platform. 2023.

[49] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP '09*, 2009.

[50] Butler Lampson. Protection. In *Fifth Princeton Symposium on Information Sciences and Systems*, 1971.

[51] Butler W. Lampson. A note on the confinement problem. *Communications of The Acm*, 16(10), 1973.

[52] Butler W. Lampson and Howard E. Sturgis. Reflections on an operating system design. *Communications of The Acm*, 19(5), 1976.

[53] Carl E. Landwehr. Formal models for computer security. *Acm Computing Surveys*, 13(3), 1981.

[54] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chen, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In *NDSS '23*, 2023.

[55] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. SoK: Software compartmentalization. In *S&P '25*, 2025.

[56] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. Cryptographic Capability Computing. In *MICRO '21*, 2021.

[57] Henry M. Levy. *8 - The IBM System/38*. Elsevier, 1984.

[58] Henry M. Levy. *Capability-Based Computer Systems*. Elsevier, 1984.

[59] Arm Limited. Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[60] Theodore A. Linden. Operating System Structures to Support Security and Reliable Software. *Acm Computing Surveys*, 8(4), 1976.

[61] Google LLC. Android developer documentation. https://developer.android.com/.

[62] Google LLC. Fuchsia Developer Documentation. https://fuchsia.dev/.

[63] Google LLC. Security-Enhanced Linux in Android. https://source.android.com/docs/security/features/selinux, 2024.

[64] Google LLC. Starnix. https://fuchsia.dev/fuchsia-src/concepts/components/v2/starnix, 2025.

[65] A. Theodore Markettos, John Baldwin, Ruslan Bukin, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Position paper: Defending direct memory access with CHERI capabilities. In *Hasp '20*, 2021.

[66] Noah Mauthe, Sven Bugiel, and Eric Ackermann. Artifact for "SoK: Capability Operating Systems: Is the Future Finally Here?". https://doi.org/10.5281/zenodo.20435520, 2026.

[67] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A Security-Oriented Subset of Java. In *NDSS '10*, 2010.

[68] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-Based Financial Instruments. In *Financial Cryptography*, 2001.

[69] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized JavaScript. Technical report, Google, Inc., 2008.

[70] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in e as plan coordination. In *Trustworthy Global Computing*, 2005.

[71] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability Myths Demolished. Technical Report SRL-2003-02, Johns Hopkins University, Systems Research Laboratory, 2003.

[72] S.J. Mullender, G. Van Rossum, A.S. Tanenbaum, R. Van Renesse, and H. Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5), 1990.

[73] Peter G. Neumann. Principled Assuredly Trustworthy Composable Architectures. Technical Report SRI Project P11459; DARPA Contract N66001-01-C-8040, SRI International, Computer Science Laboratory, 2004.

[74] Peter G. Neumann, Lawrence Robinson, Karl N. Levitt, Robert S. Boyer, and Ashok R. Saxena. A provably secure operating system. Technical Report ADA088601, Stanford Research Institute, 1975.

[75] Glen Newell. An introduction to Linux Access Control Lists (ACLs). https://www.redhat.com/en/blog/linux-access-control-lists, 2020.

[76] Elliott I. Organick. *A Programmer's View of the Intel 432 System*. 1983.

[77] Shaurya Patel, Sidharth Agrawal, Alexandra Fedorova, and Margo Seltzer. CHERI-picking: Leveraging capability hardware for prefetching. In *PLOS '23*, 2023.

[78] Fred J. Pollack, Kevin C. Kahn, and Roy M. Wilkinson. The iMAX-432 Object Filing System. In *SOSP '81*, 1981.

[79] Justin Pombrio. Preventing Log4j with Capabilities. https://justinpombrio.net/2021/12/26/preventing-log4j-with-capabilities.html, 2021.

[80] Jason Priem, Heather Piwowar, and Richard Orr. Openalex: A fully-open index of scholarly works, authors, venues, institutions, and concepts. https://arxiv.org/abs/2205.01833, 2022.

[81] Richard F. Rashid. Experiences with the accent network operating system. In *Networking in Open Systems*. 1987.

[82] John Rosenberg, Alan Dearle, David Hulse, Anders Lindström, and Stephen Norris. Operating system support for persistent and recoverable computations. *Communications of the ACM*, 39(9), 1996.

[83] SCI Semiconductor. Press release – iceni family. https://www.scisemi.com/company/press-release-iceni-family/, 2024.

[84] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *SOSP '99*, 1999.

[85] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *S&P '00*, 2000.

[86] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019.

[87] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice*. 2018.

[88] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *S&P '13*, 2013.

[89] A. Tanenbaum and S. Mullender. An overview of the Amoeba distributed operating system. *ACM SIGOPS Operating Systems Review*, 15, 1981.

[90] Thomas Van Strydonck, Job Noorman, Jennifer Jackson, Leonardo Alves Dias, Robin Vanderstraeten, David Oswald, Frank Piessens, and Dominique Devriese. CHERI-TrEE: Flexible enclaves on capability machines. In *EuroS&P '23*, 2023.

[91] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proceedings of the ACM on Programming Languages*, 3(ICFP), 2019.

[92] Kenton Varda. Cap'n Proto cerealization protocol. https://capnproto.org/.

[93] Robert N.M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical Capabilities for UNIX. In *USENIX Sec '10*, 2010.

[94] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *S&P '15*, 2015.

[95] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal Safety for CHERI Heaps. In *S&P '15*, 2020.

[96] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and Its Operating System*. 1979.

[97] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. *ACM SIGARCH Computer Architecture News*, 42(3), 2014.

[98] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6), 1974.

[99] William Allan Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/c.Mmp: An Experimental Computer System*. 1981.

[100] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *MICRO '19*, 2019.

[101] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. Capstone: A capability-based foundation for trustless secure memory access. In *USENIX Sec '23*, 2023.