Confidence in Confinement: An Axiom-free, Mechanized Verification of Confinement in Capability-based Systems

by

M. Scott Doerrie

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

July, 2015

Abstract

Confinement is a security policy that restricts the outward communication of a subsystem to authorized channels. It stands at the border of mandatory and discretionary policies and can be used to implement either. In contrast to most security policies, confinement is composable. In capability-based systems, confinement is validated by a simple decision procedure on newly minted subsystems. However, there is a long-standing debate in the literature as to whether confinement is enforceable in capability-based systems. All previous attempts to demonstrate confinement have arrived at negative results, either due to flawed system models or to proof errors that have not survived inspection.

This dissertation presents SDM: a formal, general, and extensible system model for a broad class of capability-based systems. SDM includes: 1) a mechanical formalization for reasoning about capability-based systems that produces a machine-checked proof of the safety problem, 2) the construction of a system-lifetime upper-bound on potential information flow based on the safety property, and 3) an embedding of the confinement test for capability-based systems and the first mechanically verified proof that such systems support confinement. All proofs in SDM are constructed using the Coq proof assistant without using or declaring axioms that are not part of the core logic. In consequence, there is no portion of the specification which relies on an uninstantiable assertions. SDM further distinguishes itself from other efforts by enabling the formal specifications of security-enforcing applications to be embedded without being injected into the system semantics.

Acknowledgments

To Jonathan Shapiro, my advisor, for his mentorship near and far. His guidance has been instrumental to the completion of this work and his encouragement has been inspirational.

To my parents, Mary Lou Doerrie and Steve Hardy, for encouraging me to persevere and for supporting me through difficulty. This would not have been possible without your help.

To Sabina Lindley, for her understanding and encouragement during many stressful days and late nights.

To Harriet Doerrie, Mary Lou Doerrie, Peggy Doerrie, Benjamin Hardy, Steve Hardy, Dave Paxson, and Nancy Paxson, for all their love and encouragement.

To my dissertation committee, Jonathan Shapiro, Scott Smith, and Samuel Weber, for providing critical guidance and feedback through this process.

I would also like to thank the many people who have helped me evolve this work: Nathaniel Wesley Filardo, Benjamin D. Follis, Paul Landahl, Hari Menon, Mark S. Miller, Eric Northup, Zachary Palmer, Justin Payne, Eric Perlman, Alex Rozenshteyn, Jonathan Shapiro, Scott Smith, Swaroop Sridhar, Samuel Weber, and Charles Wright.

For their help reviewing this dissertation, I would like to thank Benjamin Hardy, Nathaniel Wesley Filardo, Zachary Palmer, Justin Payne, Jonathan Shapiro, Scott Smith, and Samuel Weber. Special thanks go to Zachary Palmer for his help with LaTeX and the Johns Hopkins dissertation formatting criteria. Special thanks also go to Benjamin D. Follis for his help with PDF/A-1b formatting.

To everyone else supporting me through this effort: You are amazing! Thank you!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation considers the enforcement of confinement in capability-based systems. In capability-based systems satisfying a trivial requirement, a simple decision procedure is sufficient to determine whether a new subsystem, upon construction, will be confined. [Har86] The requirement to support confinement is that the capability system architecture maintain a Harvard-style, type-based, or similarly enforced strong separation between capabilities and data. Most modern capability systems satisfy this requirement. However, there has been substantial controversy as to whether capability-based systems can enforce confinement. This dissertation definitively resolves this issue by providing the first axiom-free, mechanically verified proof that confinement is enforceable in the majority capability-based systems.

1.1 Confinement

Lampson defined confinement in 1973 as the policy ensuring that a program "cannot transmit information to any other program except its caller" [Lam73]. It has since been generalized to restrict the transmission of information only via authorized channels. A system structured to effectively exploit confined subsystems gives users and programs the ability to scope authority securely to the places it is needed. Such a system largely resolves the problem of "agency," providing a structure in which a program wielding a user's authority can reasonably be known to act on behalf of the user, rather than some other potentially hostile party. The pervasive use of confinement in capability-based systems offers a strategy to provide defense-in-depth seldom realized in other systems.

Therefore, in systems where it is feasible, confinement offers a useful and foundational security-structuring tool. Confinement straddles the mandatory/discretionary policy border: it is discretionary for an enforcing program and mandatory for the program being confined. Given a composable confinement mechanism, traditional security policies such as isolation, privilege separation, and Bell-LaPadula or multi-level security, may all be constructed by a trusted security manager application running in user-mode. [MS03] Different portions of the system can operate under distinct security policies implemented by independent security managers, and mutually suspicious security managers can precisely limit the interaction of the subsystems under their control. High-level design patterns also emerge from confinement, often as the most straight-forward approach. For example, the “Open File” dialog can run under the authority of the user without giving the user’s authority to the program requesting a file. [SM02] Similar approaches are major undertakings in systems unable to enforce confinement, such as OpenSSH privilege separation in Unix systems. [Pro03]

While confinement is a useful primitive, there has been a great deal of concern as to whether confinement is enforceable. Lampson presented a system for which confinement is *not* enforceable in his article “Protection” [Lam74]. His system introduced the access control matrix, which portrays a static view of the system protection state, alongside a general permission-based semantics. The access control matrix structurally equates snapshots of systems using both capabilities and access control lists, and has long been incorrectly cited as evidence that both systems are equally expressive. Although the semantics accompanying Lampson’s access control matrix render confinement impossible in Lampson’s system, this is not evidence that it is impossible in sufficiently constrained systems, such as capability systems.

Harrison, Ruzzo, and Ullman brought the enforceability of many security policies under fire. [HRU76] They argue that permission-based analysis of security is a necessary precondition for security of any sort. In a system where the propagation of permissions cannot be decidably constrained, no control over access and authority is possible. Their model defines the first formal presentation of the safety problem: the decidability of determining whether one security domain will come to hold an arbitrary permission to another. They demonstrated that the answer depends on the semantics of the system and is generally undecidable. Therefore, security policies must also be undecidable in the general case. The answer is *decidable* for nearly all finite systems. Unfortunately, with the notable exception of capability-based systems, the majority of finite systems cannot prevent a permission transfer. It is critical for a safety result to be decidable and also to produce results that do not trivially defeat policy expression.

Because capability-based systems are the only general-purpose systems known to satisfy the safety property, this dissertation is focused on the enforcement of confinement in capability-based systems.

The most frequently cited argument against capability-based systems' ability to support security policies is Boebert's "On the Inability of an Unmodified Capability Machine to Enforce the *-Property." [Boe84] The *-property was introduced as part of the Bell-LaPadula access control model [BL73]: a mathematical definition of multi-level security with categories in accordance with the TCSEC standards being simultaneously developed. [UsL85] The Bell-LaPadula model partitions the system into finite domains each labeled with a numeric clearance level and set of categories. Information motion in the Bell-LaPadula model is restricted according to the *-property: information at high level domains may not reach lower levels and information between categories is restricted by subset ordering. As a purely mandatory security policy, the *-property makes no mention of how permissions or authority change. Boebert argued that the *-property could not be enforced in unmodified capability systems, but omitted any definition of what unmodified capability systems were. In response, Kain and Landwehr defined a taxonomy of capability systems, including those to be considered unmodified, and reiterated Boebert's argument. [KL87] On the basis of these, the field largely abandoned capability systems as any system incapable of enforcing the *-property cannot enforce mandatory security policies. In retrospect, this abandonment was premature.

Boebert's argument relies upon a common misconception when structuring capability systems: in systems where a subject holding a "read" capability is authorized to fetch another capability, it is possible that the fetched capability authorizes "write" access to another object, violating certain transitive expectations of information flow. Boebert constructs a system with an omnipotent security oracle and places all objects in two security domains: Low and High. As a simplified case of the *-property, his security oracle must permit information to only move from Low to High and prevent flow from High to Low. He assumes that the security oracle must grant subjects (programs) in High "read" capabilities to objects in Low and must also grant subjects in Low "write" access to other objects in Low. A trojan horse in High may now "read" a "write" capability to Low and use this newly acquired capability to violate the *-property. Boebert claims that, at this point, if the oracle interferes in any way, the system is no longer an unmodified capability system.

The two flaws with Boebert's argument are 1) unmodified capability-based systems do not exist in practice and 2) the security oracle in the example grants too much authority to enforce the *-property. Unmodified capability-based systems do not exist in practice because the system protection mechanism preserves a separation between data and capabilities that prevents usable capabilities from being transmitted though data channels. This is often preserved as a Harvard-style separation, but may also be managed through supervisor protection or type separation. Systems may safely reveal the bit-string of a capability to applications as long as capabilities cannot be fabricated from data. The ability to distinguish capabilities from data guarantees a security enforcing-application the opportunity to restrict capability transfers.

Boebert's security oracle grants too much authority and does not implement appropriate operations for managing the *-property. If a "read" capability authorizes fetching a capability from another object, the security oracle should not be granting any capabilities between Low and High using his own example as proof. However, this is not sufficient evidence that the *-property cannot be preserved generally, but only for this system arrangement. Instead of an omniscient oracle, a simple security application should be placed between Low and High that prevents all capability transfers and permits information flow in precisely one direction.

1.2 Systems Providing Confinement

The first capability system to implement confinement in capability systems was PSOS: the provably secure operating system. [FN79] Confinement in PSOS was care-fully phrased to avoid covert channels as “there shall be no inferring of protected information.” Protected information was defined as information to which a domain does not possess a capability. The authors then claimed that inference of protected information was impossible given the invariant: “The information that [domain] A has about some object for which A does not possess a capability (possibly belonging to the system) cannot increase by A calling any system function or any properly written [security-enforcing domain].” This invariant is met by PSOS because capabilities prevent direct access, the system offers no such function increasing authority, and properly written security applications have the ability restrict access as desired by restricting capability transfers. The PSOS Confined Subsystem Manager (CSM) enforces confinement by instantiating subsystems that can produce no outward communication with any domain other than their invoker and are incapable of retaining information between invocations. However, as a system whose principal concern was a strong mathematical foundation, little practical advice was given on how to structure a system around the CSM.

The work of Hardy et al. provided a general, practical implementation of confinement in the KeyKOS capability operating system. [BFH+92] [Har85] In KeyKOS, a Factory is the program charged with constructing new instances of a specific program, known as its yield, and attesting that their yield is confined. [Har86] Factories in KeyKOS implemented a confinement test as a simple decision procedure invoked with respect to an authorized set of capabilities. A successful result indicated that all yields of the factory are confined only to the outward information flow present in the authorized capabilities. Absent any authorized capabilities, no outward information flow is authorized. By including a capability to authenticate Factories as part of the system's universal trusted code base, all programs could reliably determine if a Factory's yield was appropriately confined before requesting instantiation. This confinement mechanism has been carried forward into the EROS and Coyotos operating systems.

Because Hardy's work was not widely known, the perceived failure of capability systems to enforce confinement went unchallenged in the literature until 2000 with the formulation of the SW model. [SW00]. Unfortunately, the verification in the SW model is flawed, and subsequent hand-executed verifications have erroneously arrived at negative results. [CDM01] Software continues to be developed and deployed to reason about high-level security policies in capability systems [Spi07] [EKE08]. Until SDM, there has been no definitive resolution to the confinement question.

1.3 Confidence through Automated Verification

The goal of this dissertation is to establish robust confidence in the ability of capability-based systems to enforce confinement. Confidence is the product of comprehension and observation. As there have been a number of capability-based systems built demonstrating practical implementations of confinement, this dissertation seeks to demonstrate the enforceability of confinement through rigor. While published proofs can increase confidence in system behavior, they have failed to do so in the case of confinement for capability-based systems.

Proofs fail to produce confidence in a variety of ways. They may be overtly flawed in their specification or in their execution. Flaws of specification can either exist as an inability to model the actual system under examination or by misstating the problem goal. Due to the nature of examining complex models, proof execution flaws can be very difficult to discover and may go unnoticed for many years. [Gut00] Even when no flaws have been uncovered, confidence in proofs remains inversely proportional to their complexity. This has undermined confidence in existing proofs of the enforceability of security policies in many systems, including capability-based systems.

Machine-checked verification is used to improve confidence in proofs by increasing rigor while simultaneously decreasing the material checked by review. Though reviewers are still obligated to comprehend the model definitions and assumptions, and the problem statement, they may forgo comprehension of the proof execution. Instead, reviewers may infer confidence in the proof execution from existing confidence in an automated proof assistant. Additionally, the formal rigor necessary to present a proof in a machine-manipulable form is substantially higher than with traditional proofs, further increasing confidence.

Unfortunately, mechanized proofs also introduce a host of issues impeding confidence. The same formal rigor required to construct a proof in a computational system can also decrease confidence as theorems and definitions become increasingly obscure. Proof developers are more prone to introduce specification flaws as they attempt to ease proof obligations. Model embeddings often axiomatize assumptions in ways that can also undermine confidence.

By increasing confidence via decreasing complexity for reviewers, the largest gains from automated verification can be made where proof execution is most complex. However, the very complexity of these proofs makes them difficult to mechanically produce and check. This often drives developers to construct models and problem statements that are easier to verify. Unfortunately, the portions of a system most amenable to verification are those which benefit confidence least. Multiple proofs have been performed with the effect of separating a few system concerns [KZB^{+}90] [YH10], but rarely discussing system policy and those that do have sometimes mischaracterized the problem [EKK06].

Proof developers are also tempted to encapsulate complex problems as axioms, which erodes confidence by increasing complexity. Axioms are an exceptionally powerful mechanism for theory abstraction but can inadvertently introduce all possible specification flaws. They may hide issues of decidability and construction in seemingly plausible declarations. They may also interact with core logic or other axioms in unanticipated ways, producing constrained forms of inconsistency. Many error patterns become impossible when concrete definitions are provided, and this further reduces complexity.

The safety property and confinement proofs are mission-critical properties of capability-based systems that have been controversial in the literature. Both properties are excellent candidates for automated verification as each is a simple property with far-reaching consequences. The safety property is a necessary precondition for any information flow security policy, and it is decidable in capability-based systems. Confinement is a complex and pervasive policy in any system, yet the confinement test is simple a decision procedure in capability-based systems. As confinement forms the primitive for agency and security in capability-based systems, it is unclear whether other policies can be enforced without confinement. Therefore, this dissertation establishes robust confidence in the safety property and confinement for capability-based systems through automated verification.

1.4 This Dissertation

This dissertation presents SDM: a formal, general, and extensible system model for a broad class of capability-based systems. Most capability systems can be encoded using SDM's primitives, including the Chicago Magic Number Machine [Fab74], KeyKOS [Har85], EROS [SSF99] [SSF97], Coyotos [SA08], and seL4 [KAE^{+}14].

The first contribution of this dissertation is a mechanical proof of the safety property for capability-based systems. It defines a decidable function computing *potential* *access* and describes how potential access may evolve over system operations. The proof that the potential access of the system is attenuating is a demonstration of the safety property for any capability-based system satisfying this model.

The second contribution of this dissertation is the construction of a least upper bound on potential information flow in capability systems. All information flow stems directly from permissions and the conservative approximations for permissions do not transitively alter information flow. Therefore, the attenuation of authority is directly extended across all operations to place a useful upper bound on information flow.

The third contribution of this dissertation is providing the first mechanical verification that capability-based systems support confinement. The confinement test is embedded as a post-condition of subsystem construction and includes a set of authorized capabilities. The arrangement of all possible subsystems arising from the authorized set is also defined. SDM then demonstrates the correctness of the confinement test by verifying that the mutability of all possible authorized subsystems is a subset of the mutability of any subsystem passing the confinement test.

The fourth contribution of this dissertation is providing an axiom-free proof. Axioms are a source of deep concern for any verification as they can directly encode direct impossibilities or interact with other definitions to produce inconsistencies. To relieve readers from the burden of unresolved proof obligations and thereby increase confidence in the result, SDM ensures that every abstraction can be concretized.

A distinguishing characteristic of SDM is that the definition of confinement is not embedded into the system model itself. Instead, SDM offers the ability to embed the behavior of security-enforcing applications by predicating operation sequences directly in the proof environment. This permits developers to closely model the behavior of applications in the trusted computing base, which is leveraged as part of the confinement verification. SDM also includes internal object structure pertaining to capabilities permitting future predicates to directly model named capability invocation.

SDM considers only overt confinement. Covert channels are not addressed. In systems with covert channels, it is possible to transmit information around permission boundaries. Mechanisms for mitigating or eliminating covert channels do not follow permission-based reasoning as they usually involve timing attacks. Therefore, SDM does not consider the impact of covert leakage. Mitigation of covert leakage is taken as an orthogonal problem.

This dissertation is structured in four main sections. Chapter 2 discusses how confinement is constructed in capability-based systems. The work featured herein is a part of the Coyotos project and Chapter 2 casts confinement in that light. Chapter 3 presents the confinement verification as an informal but intuitive mathematical model to provide the scaffolding in the mechanical verification. Chapter 4 addresses verification as a tool and presents Coq [BC04] as a tool for building proofs. It also covers some of the pragmatic issues encountered while using Coq to perform this verification. A high-level walk-through of the verification in Coq is presented in Chapters 5 to 9. Chapter 10 describes how SDM can be applied to existing and future systems. Chapter 11 discusses future work while related work is covered in Chapters 12 and 13, with Chapter 12 focusing on SDM's relationship with the SW verification. This dissertation concludes with a review of the work in Chapter 14.

Chapter 2

Capabilities, Confinement, and the Constructor

This chapter introduces a policy mechanism by which capability-based systems may instantiate confined subsystems. First, this chapter presents capability-based systems as they pertain to SDM. This chapter discusses how security is structured in capability-based systems and then revisits confinement in that context. Then, it presents a concrete implementation of confinement in the Coyotos Constructor domain. It concludes with a description of how confinement can be used to produce other security policies and behavior.

2.1 Capability-based Systems

The concept of capabilities and a system supervisor based entirely upon them was first envisioned by Dennis and Van Horn [DVH66] and would form the foundation of the MIT PDP-1 system [AP67]. In their words, capabilities are a structure that "locates by means of [an effective name] some computing object, and indicates the actions that the computation may perform with respect to that object." [DVH66] While their model contains many practical details for constructing an operating system, they can be distilled to a few general mechanisms that illustrate how capabilities are used. Each process was associated with a capability list, or C-list, which it could use only through supervisor implemented meta-instructions. Every action taken by a process must be authorized by a capability in its C-list, often specified by index through other meta-instructions. The supervisor implemented capabilities permitted various modes of access to built-in objects such as memory segments, processes, input/output devices, and capability storage called directories. Capabilities could be transferred when creating new processes, during inter-processes communication, or loaded and stored in directories. In addition to objects provided by the supervisor, the Dennis and Van Horn system was "extensible." Processes could provide software-defined objects through a meta-instruction constructing *entry* capabilities from a segment capability defining a protected procedure and a collection of capabilities to be made available to the procedure during operation. When invoked, these *protected entry points* were initialized with the state specified in the entry capability along with additional capabilities permitting access to the caller, presumably already sufficiently reduced to protect its parent. Levy presents an excellent history of early capability architectures in his book “Capability-Based Computer Systems.” [Lev84]

Keys are a frequently used metaphor for describing capability-based systems. Capabilities are like keys to locked objects; entering a home or starting a car ignition requires possessing the right key. New objects come with their own lock and key, unique to that object. Like keys, capabilities can be copied and distributed to other people, locked in boxes, and sent through the mail. Though they may be easily copied, good keys are extremely hard to forge and good locks are difficult to pick. In secure systems, these actions should not only be difficult, they should be impossible. Although critical to capability-based security, the metaphor is not easily extended to the construction of new agents created with their own lock and key.

A capability is an unforgable and tamper-proof binding of both an object identifier and permissions to that object. All structures exposed by the system are capability-protected objects; all operations on objects must be authorized by a capability. There must be no objects in the system which can be accessed without a capability. The system may optionally provide an extension mechanism for creating new software-defined objects also protected by capabilities. These simple rules are the only general requirements for a capability-based system.

The system must distinguish and preserve a separation between capabilities and data. Through the ability to distinguish capabilities from data, applications may restrict the transfer of capabilities in their possession. This allows applications to reliably manage not only the transmission of data, but also the transmission of permissions. By limiting both data and capabilities, applications can become effective security enforcing agents within capability-based systems.

In most capability-based systems, the separation between capabilities and data is often preserved as a Harvard-style separation. Similar to the separation of instructions and data in a Harvard architecture, these systems partition memory and expose different operations with independent addressing mechanisms for both capabilities and data. However, the separation of capabilities and data may also be managed through supervisor protection or type separation. Should the data representation of a capability be revealed by the system, this representation must remain insufficient to authorize any operations within the system. Therefore, capabilities cannot be obscurely passed through any data channels. Channels which permit the transmission of both capabilities and data must continue to preserve their separation.

2.2 Ambient Authority and Covert Channels

Ambient authority occurs when the invocation of an operation is not required to simultaneously designate an object and specify the object-specific permission authorizing the operation. For example, in most systems using access control lists, file names and domain identifiers are designators that can be used along with the system's ambient authority. If D_i has "owner" access to D_j, then D_i is permitted to grant any other domain any permission to D_j. Although an access control decision did occur to the *other* domain, D_i makes use of ambient authority because it does not need any access-control relationship to the *other* domain to grant access. All that was required was the name of the domain.

Ambient authority does not arise in capability-based systems. Capability-based systems eliminate ambient authority by combining names and permissions into a single entity: a capability. If designators are the exclusive means of wielding authority, then the absence of designation precludes authorization and, consequently, operation. When capabilities are the sole means of expressing and conveying permissions, any permission to access an object must also carry the name of that object. Therefore, by contraposition, a subject cannot come to hold an object or resource name without also holding the permission to access that object or resource.

Covert channels are not addressed by SDM. In "Confinement," Lampson characterized covert channels as "those not intended for information transfer at all." [Lam73] The hazard of this definition is that "intent" is a difficult proposition to quantify. Many *overt* channels and ambient authority have been mislabeled as "covert" with an explanation of intent. SDM considers supervisor state and access control structures to be part of the overt channels in the system. Like all other storage in SDM, these locations can only be accessed via a capability. In this regard, covert channels are more narrowly scoped and generally involve timing attacks that cannot be mitigated with access control.

2.3 The Discretionary / Mandatory Dichotomy

Most discussions of security mechanisms devolve into a discussion of mandatory and discretionary access control. The terms were widely discussed, but first appeared in the Department of Defense standard: “Trusted Computer System Evaluation Criteria” published in 1985. Discretionary access control polices are those security policies defined by the users of the system, while mandatory access control was defined by the security administrator. Discretionary policies are not robust because because their subjects have the ability to subvert them. Mandatory policies are not robust as they require the constant interaction of the security administrator. Neither approach helps us bring the granularity of policy down to something that is practically helpful.

Another definition of mandatory access control is “that a security officer may constrain the owner of an object in determining who may have access rights to that object.” [HKN05] Access control mechanisms that rely only upon an unprotected name for authorizing an action often introduce ambient authority. This can be as simple as being able to deny access based only upon a name, as previously mentioned. In systems that rely on public unprotected identities for access control decisions, it is unclear how to completely eliminate ambient authority.

Every system enforcing mandatory access control contains some user representing the security administrator. This user, or superuser, has complete control of the system and the entire system security policy is discretionary to this user. Therefore, a generalized view is that mandatory access controls are imposed upon subjects without their consent, but subjects of discretionary access control choose to abide by the policy. From this perspective, mandatory and discretionary access controls have to do with which side of a policy subjects sit upon.

When discretionary/mandatory access control mechanisms are viewed as policy border mechanisms, it is possible to view the system policy as a hierarchy or lattice of policies imposed by different subjects. The notion that a user can successfully enforce a policy and might collaborate in their own defense is absent from the TCSEC definitions. This property is crucial for systems to enforce robust composable policies, providing defense-in-depth. Therefore, this dissertation refines the definition of a mandatory policy as one that an application cannot escape, while a discretionary policy is one that an application consents to abide by. Note that the focus here shifts from the users of the system to the agents of the system, the applications.

Confinement is a composable policy that sits at the border between mandatory and discretionary policies. Confinement is mandatory for the subsystem being constructed but applied at the discretion of the constructing subsystem. By leveraging confinement during initial system construction, it has been used as a building block to implement mandatory controls. [Raj89] It permits applications to enforce security policies and permits different portions of the system to operate under different security managers. Confinement also allows users to establish clear and fine-grained distinctions between programs that act on behalf of a user and sub-programs that presumably do not. Confinement integrates tightly with the encapsulation and modularity boundary. Viewed as a building block, confinement allows us to restructure systems in a way that fundamentally reduces their attack surface by selectively localizing authority into objects that have narrow, validating APIs.

2.4 Confinement Revisited

This dissertation is concerned with confinement as a constructive, perimeter-enforcing security policy preventing unauthorized outward information flow. Confinement is a constructive policy, imposed upon freshly minted subsystems. Confinement straddles both mandatory and discretionary access control, erecting a mandatory policy from discretionary authority. Confinement ensures all outward information flow is authorized by the constructing subsystem, and an absence of outward information flow is expressible.

Confinement is not implemented as part of the system protection mechanism but is instead assembled from the underlying capability system as a software design pattern. Although the remainder of this chapter will focus on a particular confinement mechanism, it is likely that other mechanisms exist. It is possible for any application to produce a confined subsystem by construction, without direct reliance upon its trusted computing base. To produce a confined subsystem, it suffices that an application's TCB not interfere with it's operation while instantiating a subsystem and the system will continue to enforce confinement without further intervention.

2.5 Coyotos: a Concrete Example

Coyotos is a micro-kernel object-capability operating system. Unlike traditional kernels, or monolithic kernels, micro-kernels are designed to be minimalist and extensible. They are designed to permit applications to safely extend the system while incurring a minimal overhead. Therefore, a substantial portion of system software is separate from the kernel and is not run in supervisor mode. This includes device drivers, storage managers, portions of the scheduler, and security policies.

Objects in Coyotos are exclusively accessed by invoking kernel-protected capabilities. This includes memory pages and page tables, processes, endpoints for interprocess communication, access to interrupts and I/O, and even the scheduler. Coyotos is an object-based system and, whether implemented by the kernel or application, invoking a capability is tantamount to a remote procedure call, potentially handled by the kernel. Kernel objects and application objects share the same message marshaling interface. The kernel directly processes capability invocations to primitive objects.

When a capability naming a software-defined object is invoked, the kernel performs an inter-process communication rendezvous with the implementer. As capability invocation accounts for virtually all operations, Coyotos has only three system calls: invoke a capability, copy a capability, and yield the processor. Even memory loads and stores can be modeled as capability invocations.

Coyotos is also an atomic action kernel; from the perspective of the system, all kernel-implemented actions occur indivisibly. The kernel does not implement any long-running operations that would obligate it to return control to an application. Therefore, in a single-processor implementation, no locking is necessary as all operations can successfully complete before scheduling a process. In a multi-processor environment, all necessary locks must be obtained before an observable change to the system can occur. Should some locks be in use elsewhere, the kernel must avoid deadlock by ensuring that at least one system call can complete or by releasing the locks for this call and dispatching another request.

Coyotos implementations preserve atomic actions at the abstraction of capability invocation, which may not always unify with a single system operation. Each system operation exposed is permitted to consist of multiple atomic micro-operations and it is these micro-operations which can be serialized, even when system calls cannot. For example, memory loads and stores may require a traversal of the process' root memory mapping structure. From the perspective of the process, the entire load or store is a single system operation, but it is not indivisible. During a traversal, the

Figure 2.1 Physical representations of Coyotos capabilities.

$AllocCount_{(20)}$	$restr_{(5)}$	P	$type_{(6)}$
$ProtectedPayload_{32}$ or $GptMapData_{(32)}$			
$OID_{(64)}$			

system guarantees only that each step of translation via a capability is indivisible. It is possible that other computation may modify address space objects along the current translation path such that no serialization of system calls is possible. However, the atomic micro-operations of the system may be consistently serialized and it is these micro-operations under consideration in SDM.

A capability in Coyotos is a system-protected 16-byte structure containing enough information for the system to perform an appropriate invocation. Each capability contains a 6-bit field indicating the type of the capability. There are some capability types that are unique, but types that have more than one object also contain a unique 64-bit object identifier and an allocation count. The object identifier is unique to the object and does not change over the life of the object, though it may be reclaimed after the object is destroyed. The allocation count is used by the system reclamation mechanism to determine if the capability is still valid.

Coyotos uses a virtual memory management mechanism inspired by Liedtke's *guarded page table* proposal. [LE96] The Coyotos kernel must ensure that the guarded page table, or GPT, structure is the authoritative address translation mechanism on architectures dictating the memory management structure. There are three basic types of memory object: pages to hold data, CapPages to hold capabilities, and

GPTs to construct address spaces. Capabilities naming memory objects contain a set of access restrictions and a guard. Access to a page or CapPage is restricted by all access restrictions along traversal path through memory capabilities. As large address spaces are typically sparsely populated, guards are used as a mechanism for compactly representing invalid translations without a page table and do not have any other access control impact.

The primary access restrictions are "read-only," "no-execute," and "weak." A capability with no restrictions is a "read-write" capability and permits both loads and stores. The "read-only" and "no-execute" restrictions are familiar, respectively prohibiting a store or an instruction fetch[1]. The "weak" restriction ensures that any capability read through this path will be selectively downgraded to ensure transitive read-only authority. Memory capabilities fetched via this path are downgraded by introducing the "weak" restriction. The system returns a null capability for all capabilities where no appropriately downgraded capability exists.

The system provides no rights-amplifying operations. Capabilities are a system protected structure and cannot be fabricated by applications. This is enforced by marking virtual memory mappings for CapPages with supervisor-only access. Thus, being able to view or copy capabilities as data does not confer their authority. The Coyotos kernel provides the *KeyBits* capability to permit applications to view the canonical data of a capability. Transferring capabilities cannot be used to transmit

[1]The no-execute restriction is ignored on architectures where it cannot be enforced.

any information not already transmissible via the same channel. Therefore, the KeyBits capability is considered sensitive only as it reveals some of the inner workings of the system and not because it admits communication between applications.

There are a few exceptions where universal system software does not run in supervisor mode. The atomic kernel design motivates leveraging application software to perform operations involving memory-allocating bookkeeping or lengthy blocking I/O requests. The notable subsystems included in the universal trusted computing base are the storage manager, the *Space Bank*, and the authenticated subsystem builder, the *Constructor*. The kernel provides sensitive capabilities with the expectation that they are exclusively held by these subsystems. Enforcing this constraint is managed by these subsystems and not by the kernel.

2.6 The Space Bank

The *Space Bank* is the part of the universal TCB responsible for managing storage. It is admitted to perform this task as it uniquely holds the range capability, which grants access to all storage. Coyotos considers all allocatable structures as storage: Processes, Endpoints, GPTs, Pages, and CapPages. The primary responsibility of the Space Bank is to perform allocation requests while maintaining memory safety. In this context, memory safety obliges the Space Bank to not respond to a request with a capability naming an object that is already live. This behavior of the *Space*

Bank is so fundamental to the behavior of the system that it will be considered part of the system model for the remainder of this document.

The second responsibility of the Space Bank is to manage quotas. Because the system has a finite number of resources, the Space Bank itself has a quota. All other quotas are simply another constraint on the system, and implement the same Space Bank interface. These sub-banks are implemented as different capabilities to the Space Bank. Each quota can be deallocated as a single unit, effectively destroying all allocations within it and returning the storage to their parent.

2.7 The Constructor

In Coyotos, constructors are the applications that instantiate new subsystems. Constructors contain a subsystem image which they use to produce a subsystem upon request, called their yield. They may also be queried to determine if their internal image is confined. An affirmative result indicates that the yield of the constructor cannot exceed the information flow inherent in capabilities provided by the application requesting the yield. The result of a confinement test may be included in an application's predicate for determining which subsystems are safe to instantiate.

The first step in a constructor's life-cycle is as the yield of the meta-constructor. Once instantiated and initialized, a constructor returns its builder capability and begins operating in the builder phase. As a builder, the constructor accepts commands which populate its internal subsystem image until it receives a seal command. The constructor responds to a seal command by invalidating its builder capabilities and returning its constructor capability.

A sealed constructor will no longer perform updates to its subsystem image, but can be used to yield new subsystems. The constructor requires its client to provide the storage capabilities for each yield. It then allocates a new process and populates it according to its internal subsystem image. Finally, the constructor uses the process capability to fabricate an initial entry capability and invokes the new subsystem. As it does so, the constructor does not perform the standard call-return procedure. Instead, it passes the return capability specified by the caller requesting its yield. When the yield has finished initializing, it should reply directly to the original caller. After yielding a subsystem, the constructor guarantees that it will not leak any capabilities regarding the yield, usually by overwriting them upon receiving its next request.

Before a process instantiates a subsystem using a constructor, it can ask the constructor to check whether that subsystem is confined. The constructor performs the confinement test, ensuring that all outward information flow of the yield follows only from authorized capabilities. Specifically, the only capabilities that may produce outward information flow are those granted to the yield by the requesting process, and consequently no information flow is authorized when no capabilities are granted. This test is performed simply by checking that all capabilities within the constructor's subsystem image are weak, trivially non-mutating, or name a recursively confined constructor.

A parameterized confinement test is provided by the KeyKOS Factory. [Har86] The KeyKOS Factory operates similarly to the Coyotos Constructor with slight variations on interfaces and terminology that have been altered for consistency. During its builder phase, as capabilities are added, the KeyKOS Factory maintains a list of "holes:" those capabilities for which it cannot statically guarantee confinement. The only capabilities not added to this list are those which are trivially non-mutating and those that are weak capabilities. As with Coyotos' Constructors, an application may query a Factory regarding the confinement of its yield before requesting the yield to be instantiated. In KeyKOS, this request also includes an *authorized set* of capabilities. To be confined, the Factory requires each of the holes to be within the authorized set or name a Factory whose yield is confined under the same authorized set.

Many capability-based systems, including EROS and Coyotos, do not include a parameterized confinement test and the authorized set is empty. In these systems, all of the present system security structure can be constructed using the simplified confinement test, which increases confidence in the result. However, to produce a proof widely applicable across many capability systems, SDM supports the parameterized confinement test.

Constructors are equipped with the *brand* capability which they use to provide a verification interface. During subsystem construction, they brand their yield with a value unique to the constructor, for example, an HMAC of the subsystem image.

Upon request, a constructor will verify whether they created a process by checking the process' brand.

The meta-constructor is the constructor that yields constructors. It has read-only access to its instructions and duplicates this access as part of its subsystem image. The meta-constructor is initially sealed, causing all constructors to have identical behavior. Because constructors can verify another process, access to the meta-constructor allows constructors to be verified. This procedure is used to allow the constructor to verify the authenticity of other constructors when querying them for confinement. The ability to fabricate and verify confined subsystems is so critical to robustness that universal access to the meta-constructor is considered part of the system-wide TCB.

Regardless of what operations the yield of this constructor performs, the only overt information flow it may cause follows from the use of capabilities in the authorized set. If this were not the case, then there must be some capability not in the authorized set which may be used to produce an outward information flow as capabilities are the only mechanism to produce overt information flow. But the only capabilities not in this authorized set are those known to never produce outward information flow, or sealed constructors whose result is recursively confined under the same set. Confinement requests cannot form a cycle as the constructor capability is not produced until it seals its system image. By induction, no such capability can exist, and the yield of the constructor must be confined.

The Constructor and Space Bank are designed to work in tandem to provide additional guarantees. A constructor requires parents to provide a Space Bank capability from which it will allocate its yield. The common behavior of parents is to allocate a new sub-bank for the new subsystem. As the allocator of this sub-bank, the parent has the ability to destroy the entire subsystem. In addition to granting the parent the ability to confine the subsystem, this ensures that the parent may limit the duration the subsystem is present. A common pattern of creating "memory-less" services is to have the parent destroy them after each use.

Space Banks are not initially constructed subsystems, and provide their own verification interface. The constructor is able to verify Space Banks as it holds a capability to the root bank. It passes this ability on to is children by ensuring that the bank capability used to allocate the new subsystem is verified by the root bank. Because the constructor relies on the correct behavior of a Space Bank to construct its yield, this check also protects the constructor from abuse.

2.8 Initial System Construction

The construction of an initial system image is handled by the Coyotos *mkImage* utility. mkImage is an imperative scripting interpreter with built-in commands to describe and generate a system image. Conceptually, mkImage can be thought of as a link editor that operates over Coyotos objects.

Because mkImage is run prior to the initial system boot, it incorporates behavior from both the Space Bank and constructor domains. Accounting data for the Space Bank is maintained by mkImage when new sub-banks are constructed and objects are allocated. To construct new domains, new constructors are built from executable images as though they were the yield of the meta-constructor. They require a capability to a sub-bank for allocation, include a capability to the meta-constructor in their system image, and are appropriately branded for later checking by the meta-constructor. A similar procedure constructs yields from individual constructors. Therefore, the Space Bank and all constructor domains must be able to start from a pre-initialized state.

Security policies in the initial system image that provide no authorized interface for alteration become immortal. A *user* in a capability system is simply a program managing a long-running, authenticated session with an individual. In a system image without a capability authorizing a policy change, either by program design or by image construction, there can be no user or super-user who can effect such a change. This is often more powerful than the traditional mandatory policy as not even a security administrator can alter such a policy. Metaphorically, this is like loosing your keys in a universe prohibiting locksmiths.

The mkImage tool does not provide a confinement test, but this does not make confinement less relevant. The confinement test does not require the constructor to execute it and may be performed by anyone. Therefore, the system developer should ensure that every capability issued as part of the initial system image is intentional.

2.9 Confinement as a Keystone

Confinement gives applications the ability to scope authority to exactly those subsystems where it is needed and is a foundational tool for building other security policies. Isolation is a mandatory security policy requiring that all information flow is authorized between subsystems. To enforce this policy, the description of what information flow is authorized is encoded by generating authorized capability sets for each subsystem. Before the system image instantiator builds these subsystems, it can check each subsystem for confinement against the approved information flow list. Because we have checked every outbound information flow, all inbound information flow must be approved as well.

A simple reference monitor mediating confined subsystems can produce the Bell-LaPadula [BL73] mandatory access control, or Multi-level security. As previously mentioned, the Bell-LaPadula model labels all domains with a clearance level and set of categories; access and information flow is only permitted between domains based on integer and subset ordering, respectively. To construct such a system using confinement, each domain is confined with respect to a security monitor. The security monitor manages all requests based on the originating domain and ensures that all requests are isolated by prohibiting the exchange of capabilities across domains.

Confinement is not only useful for building and enforcing mandatory policies, but also constructing dynamic, application-defined policies. The KeyKOS Receptionist is the domain invoked by a device, usually a terminal, that authenticates a user's credentials and connects the device to the user's compartment. This is performed simply by passing the device interface along to the user's compartment, effectively eliminating the Receptionist from the communication path. While easy to describe in object-based capability systems, structurally ensuring the same privilege separation guarantees for SSH running on Unix systems was a substantial undertaking.

The "Open File Dialog" is an example of one of the more powerful dynamic policies to emerge from object-capability systems. In traditional systems, applications running on behalf of a user have the ability to access all of that user's files. When well-behaved, these applications will prompt the user for which files they intend the application to access. Unlike the traditional model where this behavior is subject to the whim of the application, it can be enforced as a security policy in object-based capability systems. A user may construct an "Open File Dialog" as a subsystem that holds all of their authority and they trust to identify itself when prompting them about capabilities. When instantiating a new untrusted process, the user need not grant them any interesting capabilities other than a new open file dialog with a unique identifier. Regardless of what behavior the new process performs, the user will be able to identify when a prompt is originating through some action initiated by the new process. Using a trustworthy dialog, the user can remain in control of which, if any, capabilities are granted.

The "Open File Dialog" example highlights how capabilities can be leveraged by developers to produce robust patterns of collaboration through modularity and encapsulation. The constructor mechanism permits applications to confine authority directly where it is needed and verify whether applications were constructed faithfully. This paradigm permits developers to reliably construct new software objects with clearly articulated boundaries and interfaces which include fine-grained security decisions. When deployed pervasively, the system can be structured to provide users the ability to comprehend the various contexts in which applications operate on their behalf. By giving the user the ability to scope these contexts using confinement, these systems provide users trustworthy agents to act on their behalf.

Chapter 3

Proof Sketch

This chapter contains a high-level proof sketch to assist the reader while examining the comprehensive proof of SDM. It provides simplified versions of key definitions and theorems that cover a wide range of capability-based systems. The use of mechanically manipulable specifications is eschewed in favor of familiar, hand-written mathematics. The proof sketch starts by describing an abstract form of the model structures in SDM. It then presents the model semantics as state updates along with the potential for data motion. Possible system states and information flow that can happen within the system are defined inductively over sequences of these operations.

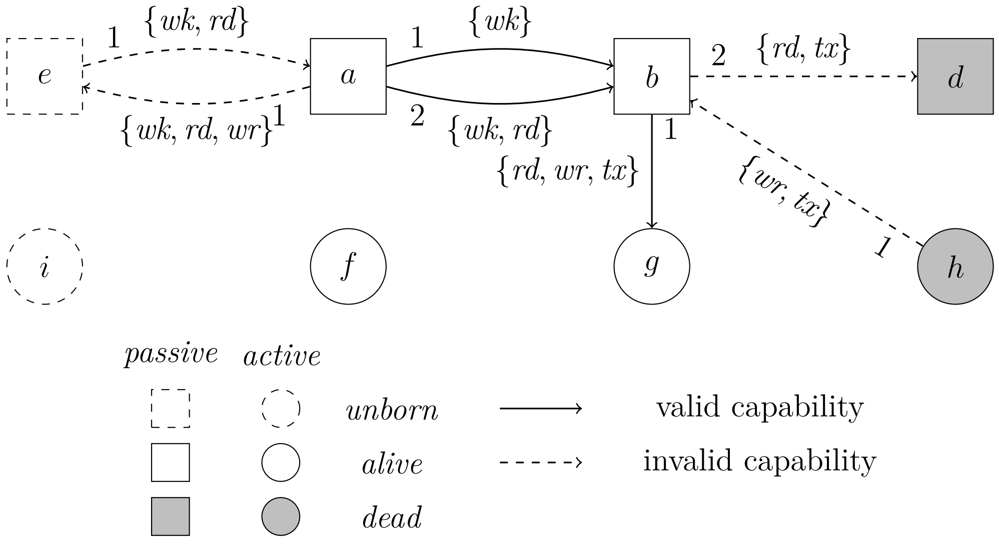
The first major theorem presented is the safety property from Section 1.1. This is accomplished using a simplified structure of systems, the *access graph*, that reduces complexity to access between objects. Using access graphs, this sketch defines *direct access* and *potential access* as the access that is present in the system and that which maximally can be present in the future. Finally, it defines functions that conservatively approximate both direct and potential access over the model operations and uses these functions to demonstrate that the potential permissions for pre-existing objects never increases over the life of the system, a property called *attenuating permissions*. Because potential access is attenuating in SDM, it consequently answers the safety question.

The confinement test does not examine the entire system, but only the capabilities to be placed in constructed subsystem. The safety property provides an upper bound on permissions, but does not directly yield an upper bound on information flow. The next major theorem demonstrates that potential access can produce a conservative approximation of potential information flow. This proof opens with an inductive definition of what is *mutated* in the system and defines a simple predicate for deciding what is *mutable* using access graphs. It then demonstrates that the actual mutation of the system is bounded by potential mutability and shows that potential mutability between existing objects can never increase. Next, this sketch introduces the confinement problem as a whole-system post-condition. Finally, it introduces a representation of the confinement property using access graphs and demonstrate that if the post-condition is satisfied, the subsystem must be confined.

Figure 3.1 Relevant definitions for system states.

$$\begin{aligned}
R &\equiv \{tx, wr, rd, wk\} \\
Cap &\equiv Ref \times R^2 \\
Obj &: Idx \to Cap \\
L &\equiv \{unborn, alive, dead\} \\
T &\equiv \{active, passive\} \\
S &: Ref \to Obj \times L \times T
\end{aligned}$$

Figure 3.2 Example system state.



3.1 System State

The permission state of a capability-based system at any instant is modeled by a *system state* in SDM. A system state is represented as a finite map of finite maps defined in Figure 3.1. Each *object reference* is mapped to an object, object label, and object type; each *object* is a map from an index to a capability. An object's *type* indicates whether it is a process or passive storage. An object's *label* captures a section of its life-cycle, which is permitted to transition only from *unborn* to *alive* and *alive* to *dead*. *Indices* label the cells within an object, which contain capabilities. A *capability* consists of a target object reference and a set of access rights.

The *access rights* (or permissions) in the system are *rd*, *wr*, *wk*, and *tx*. Access rights are checked as part of the preconditions for the semantic operations in Section 3.2. The *rd* and *wr* permissions enable the ability to directly read or write information in the target. The *wk* permission is a sub-type of the *wk* permission that authorizes transitive read-only authority. The *tx* access right authorizes message passing containing both capabilities and data along with an optional reply capability.

SDM does not explicitly represent object data or intra-object computation. Because all possible operation sequences will be analyzed, tracking which data are moving is unnecessary. The model only tracks which objects could be *modified* by the motion of data.

When diagramming a system state, the convention herein uses the shape of the object to indicate the object's type and style to represent life-cycle. Active objects (processes) are circles and passive storage objects are squares. Objects with solid borders are *alive*, objects with dashed borders are *unborn*, and gray objects are *dead*. Capabilities are represented as arrows within the graph, each capability is labeled with their access right set along the arc center. For clarity, capabilities that are permitted structurally but which have no semantic interpretation are given a dashed line to differentiate them from semantically relevant capabilities. The index at which each capability is stored is denoted along its arc close to the object. An example is included in Figure 3.2.

3.2 Semantics

The system state evolves by executing a sequence of *operations* through which data and capabilities may flow. Each *operation* is defined in three parts: a precondition, a transformation of system state, and an upper bound on information flow. Figure 3.4 the preconditions that expresses the sanity requirements for each operation to ensure safety. In particular, processes can only specify the target of an operation by invoking a capability at a specific index. These preconditions also check this capability for the presence of a necessary access right, capturing an access control decision. Therefore, a system state transition and potential data flow occur only when the precondition is satisfied.

The operation state transitions are defined in Figure 3.5. The notation $S \xrightarrow{op} S'$

indicates that executing *op* in system state S results in state S'. Executing a sequence of operations is represented by the notation $S_0 \ldots \xrightarrow{op_{m-1}} S_{m-1} \xrightarrow{op_m} S_m$.

Information flow is modeled using the *readFrom* and *wroteTo* functions defined in Figure 3.6. During a successful operation, data may potentially flow from each of the objects in the *readFrom* function to the objects named by *wroteTo*. While each operation varies with respect to its target, the model presumes that the acting subject of an operation is always in the *readFrom* set.

Operations should be considered traces of system execution and do not represent system calls. For example, in real systems, the *send* operation contains a managed rendezvous between the sender and recipient. The acting subject performing a send operation specifies which capabilities and data should be transmitted, but the recipient indicates where they should be placed. Also, processes in real systems do not choose, and cannot observe, which new object is allocated; the system selects it on their behalf.

Figure 3.3 defines some commonly used functions, though the following have been omitted for brevity. *mkCap* constructs a new capability. *multiCapCopy* inductively copies capabilities by examination of a list of index pairs where the source is the first argument and the target is the second. Each operation has an acting subject, labeled a in the definitions, and is selected by the *acting* function. The *removeCapsByRef* function removes all capabilities naming a specific reference front the system and is used to sanitize the system before allocating an object. Maps are sets and the notation $k \overset{M}{\longmapsto} v$ indicates that k is mapped to v in map M. When examining a map, an underscore "_" will indicate that the value is ignored. An asterisk "*" used while updating a map leaves the previous value unaltered and map erasure is indicated using epsilon "ϵ" for the value.

These figures use many common symbols; the relevant ones are listed here. Object references are often labeled by a single character o, a when it is the reference to the acting object, or n for a new object reference. The variables *src* and *tgt* are also used to denote object references in appropriate contexts. The objects themselves are often simply *obj*, or *aObj* for the acting object. Generic indices are labeled *i*. When they identify the capability being invoked, they are labeled t as they name the target object. When being accessed by the capability at t, they are also labeled c. Maps represented as lists of index pairs are often represented by m, though later sections will use this for mutation. System states are denoted by S, and access graphs denoted by I and A. P often denotes a potential access graph and D ranges over direct access graphs.

The *read* and *write* operations model data reads and writes to an object and require the *rd* and *wr* access right, respectively. Because non-self data motion is modeled by *readFrom* and *wroteTo*, these operations have no impact on the system state. *readFrom* contains the capability target for a read operation along with the invoking subject. *wroteTo* contains the capability target for a write operation or the invoking subject.

Figure 3.3 Helper functions.

$$\begin{array}{rcl}
\mathit{hasRight}(o,i,S,r) & \equiv & \exists \mathit{obj}, o \overset{S}{\mapsto} (\mathit{obj},\_,\_)\wedge \\
 & & \exists \mathit{tgt}, \mathit{arset}, i \overset{\mathit{obj}}{\longmapsto} \mathit{mkCap}(\mathit{tgt},\mathit{arset})\wedge \\
 & & r \in \mathit{arset} \\
\mathit{isLabel}(o,S,l) & \equiv & o \overset{S}{\mapsto} (\_,l,\_) \\
\mathit{isUnborn}(o,S) & \equiv & \mathit{isLabel}(o,S,\mathit{unborn}) \\
\mathit{isAlive}(o,S) & \equiv & \mathit{isLabel}(o,S,\mathit{alive}) \\
\mathit{isAlive}(o,S) & \equiv & \mathit{isLabel}(o,S,\mathit{dead}) \\
\mathit{isType}(o,S,\mathit{typ}) & \equiv & o \overset{S}{\mapsto} (\_,\_,\mathit{typ}) \\
\mathit{isActive}(o,S) & \equiv & \mathit{isType}(o,S,\mathit{active}) \\
\mathit{targetIsAlive}(o,i,S) & \equiv & \exists \mathit{obj}, o \overset{S}{\mapsto} (\mathit{obj},\_,\_)\wedge \\
 & & \exists \mathit{tgt}, i \overset{\mathit{obj}}{\longmapsto} \mathit{mkCap}(\mathit{tgt},\_)\wedge \\
 & & \mathit{isAlive}(\mathit{tgt},S) \\
\mathit{preReqActor}(a,S) & \equiv & \mathit{isAlive}(a,S) \wedge \mathit{isActive}(a,S) \\
\mathit{preReqCommon}(a,t,S) & \equiv & \mathit{preReqActor}(a,S) \wedge \mathit{targetIsAlive}(a,t,S) \\
\mathit{objTarget}(o,t,S) = \mathit{tgt} & \Longleftrightarrow & o \overset{S}{\mapsto} (\mathit{obj},\_,\_) \wedge t \overset{\mathit{obj}}{\longmapsto} \mathit{mkCap}(\mathit{tgt},\mathit{arset}) \\
\mathit{hasCap}(o,\mathit{cap},S) & \equiv & o \overset{S}{\mapsto} \mathit{obj} \wedge \exists i, i \overset{\mathit{obj}}{\longmapsto} \mathit{cap} \\
\mathit{replyCap}(\mathit{obj},i,o) & \equiv & \mathit{obj}[i \mapsto \mathit{mkCap}(o,\{\mathit{tx}\})]
\end{array}$$

Figure 3.4 Operation preconditions.

$$\begin{array}{rcl}
\mathit{preReq}(\mathit{read}(a,t),S) & \equiv & \mathit{preReqCommon}(a,t,S)\wedge \\
 & & (\mathit{hasRight}(a,t,S,\mathit{rd}) \vee \mathit{hasRight}(a,t,S,\mathit{wk})) \\
\mathit{preReq}(\mathit{write}(a,t),S) & \equiv & \mathit{preReqCommon}(a,t,S) \wedge \mathit{hasRight}(a,t,S,\mathit{wr}) \\
\mathit{preReq}(\mathit{fetch}(a,t,c,i),S) & \equiv & \mathit{preReqCommon}(a,t,S)\wedge \\
 & & (\mathit{hasRight}(a,t,S,\mathit{rd}) \vee \mathit{hasRight}(a,t,S,\mathit{wk})) \\
\mathit{preReq}(\mathit{store}(a,t,c,i),S) & \equiv & \mathit{preReqCommon}(a,t,S) \wedge \mathit{hasRight}(a,t,S,\mathit{wr}) \\
\mathit{preReq}(\mathit{revoke}(a,t,c),S) & \equiv & \mathit{preReqCommon}(a,t,S) \wedge \mathit{hasRight}(a,t,S,\mathit{wr}) \\
\mathit{preReq}(\mathit{destroy}(a,t),S) & \equiv & \mathit{preReqCommon}(a,t,S) \wedge \mathit{hasRight}(a,t,S,\mathit{wr}) \\
\mathit{preReq}(\mathit{allocate}(a,n,m,\mathit{typ}),S) & \equiv & \mathit{preReqActor}(a,S) \wedge \mathit{isUnborn}(n,S) \\
\mathit{preReq}(\mathit{send}(a,t,m,x),S) & \equiv & \mathit{preReqCommon}(a,t,S) \wedge \mathit{hasRight}(a,t,S,\mathit{tx}) \\
\mathit{weaken}(\mathit{mkCap}(\mathit{tgt},\mathit{arset})) & \equiv & \left\{ \begin{array}{cl} \mathit{wk} & | \quad \{\mathit{wk},\mathit{rd}\} \cap \mathit{arset} \neq \emptyset \\ \emptyset & | \quad \text{otherwise} \end{array} \right.
\end{array}$$

Figure 3.5 State transitions.

$$
\begin{array}{rcl}
S \xrightarrow{read(a,t)} S' & \Longleftrightarrow & S' = S \\
S \xrightarrow{write(a,t)} S' & \Longleftrightarrow & S' = S \\
S \xrightarrow{fetch(a,t,c,i)} S' & \Longleftrightarrow & a \overset{S}{\mapsto} aObj \wedge \\
 & & t \xmapsto{aObj} tCap \wedge \\
 & & tgt = objTarget(a, t, S) \wedge \\
 & & tgt \overset{S}{\mapsto} tObj \wedge \\
 & & c \xmapsto{aObj} cap \wedge \\
 & & cap' = \text{if } rd \in tCap \text{ then } cap \text{ else } weaken(cap) \wedge \\
 & & S' = S[tgt \mapsto (tObj[i \mapsto cap'], *, *] \\
S \xrightarrow{store(a,t,c,i)} S' & \Longleftrightarrow & a \overset{S}{\mapsto} aObj \wedge \\
 & & tgt = objTarget(a, t, S) \wedge \\
 & & tgt \overset{S}{\mapsto} tObj \wedge \\
 & & c \xmapsto{tObj} cap \wedge \\
 & & S' = S[a \mapsto (aObj[i \mapsto cap], *, *)] \\
S \xrightarrow{revoke(a,t,c)} S' & \Longleftrightarrow & tgt = objTarget(a, t, S) \wedge \\
 & & tgt \overset{S}{\mapsto} tObj \wedge \\
 & & S' = S[tgt \mapsto (tObj[c \mapsto \epsilon], *, *)] \\
S \xrightarrow{destroy(a,t)} S' & \Longleftrightarrow & tgt = objTarget(a, t, S) \wedge \\
 & & tgt \overset{S}{\mapsto} tObj \wedge \\
 & & S' = S[tgt \mapsto (*, dead, *)] \\
S \xrightarrow{allocate(a,n,m,typ)} S' & \Longleftrightarrow & S_{clean} = removeCapsByRef(n, S) \wedge \\
 & & a \xmapsto{S_{clean}} aObj \wedge \\
 & & newObj = multiCapCopy(aObj, \emptyset, m) \wedge \\
 & & S' = S_{clean}[n \mapsto (newObj, alive, typ)] \\
S \xrightarrow{send(a,t,m,x)} S' & \Longleftrightarrow & a \overset{S}{\mapsto} aObj \wedge \\
 & & tgt = objTarget(a, t, S) \wedge \\
 & & tgt \overset{S}{\mapsto} tObj \wedge \\
 & & tObj1 = multiCapCopy(aObj, tObj, m) \wedge \\
 & & tObj2 = \text{if } x \text{ then } replyCap(tObj1, x, a) \text{ else } tObj1 \wedge \\
 & & S' = S[t \mapsto (tObj2, *, *)] \\
\text{otherwise} & & \\
S \xrightarrow{op} S' & \Longleftrightarrow & S = S'
\end{array}
$$

Figure 3.6 Information flow.

$$
\begin{array}{rcl}
\mathit{readFrom}(\mathit{read}(a,t),S) & \equiv & \{a, \mathit{objTarget}(a,t,S)\} \\
\mathit{readFrom}(\mathit{write}(a,t),S) & \equiv & \{a\} \\
\mathit{readFrom}(\mathit{fetch}(a,t,c,i),S) & \equiv & \{a, \mathit{objTarget}(a,t,S)\} \\
\mathit{readFrom}(\mathit{store}(a,t,c,i),S) & \equiv & \{a\} \\
\mathit{readFrom}(\mathit{revoke}(a,t,c),S) & \equiv & \{a\} \\
\mathit{readFrom}(\mathit{destroy}(a,t),S) & \equiv & \{a\} \\
\mathit{readFrom}(\mathit{allocate}(a,n,m,\mathit{typ}),S) & \equiv & \{a\} \\
\mathit{readFrom}(\mathit{send}(a,t,m,x),S) & \equiv & \{a\} \\
\\
\mathit{wroteTo}(\mathit{read}(a,t),S) & \equiv & \{a\} \\
\mathit{wroteTo}(\mathit{write}(a,t),S) & \equiv & \{\mathit{objTarget}(a,t,S)\} \\
\mathit{wroteTo}(\mathit{fetch}(a,t,c,i),S) & \equiv & \{a\} \\
\mathit{wroteTo}(\mathit{store}(a,t,c,i),S) & \equiv & \{\mathit{objTarget}(a,t,S)\} \\
\mathit{wroteTo}(\mathit{revoke}(a,t,c),S) & \equiv & \{\mathit{objTarget}(a,t,S)\} \\
\mathit{wroteTo}(\mathit{destroy}(a,t),S) & \equiv & \{a\} \\
\mathit{wroteTo}(\mathit{allocate}(a,n,m,\mathit{typ}),S) & \equiv & \{a,n\} \\
\mathit{wroteTo}(\mathit{send}(a,t,m,x),S) & \equiv & \{\mathit{objTarget}(a,t,S)\}
\end{array}
$$

when preconditions are satisfied, otherwise

$$
\begin{array}{rcl}
\mathit{readFrom}(\mathit{op},S) & \equiv & \{\} \\
\mathit{wroteTo}(\mathit{op},S) & \equiv & \{\}
\end{array}
$$

The *fetch* and *store* operations model capability motion and have the same predicates and information flow properties as the *read* and *write* operations. The difference is that *fetch* and *store* operations read or write capabilities instead of data. These operations update the system model by transferring a capability from or to the specified index in the target object. The fetch operation has a special case when the capability contains the *wk* permission, but not the *rd* permission. In this case, it is still possible to *fetch* a capability from the target, but the resulting capability will be weakened using the *weaken* function. A weakened capability has an access right set of $\{wk\}$ only when the target capability has either the *rd* or *wk* right. This has the effect of causing the *wk* access right to enforce transitive read-only access.

The revoke operation erases a mapping within an object. Because this is almost identical to overwriting an existing capability using *store*, it requires the *wr* permission. The *wr* access right also authorizes the destroy operation as the acting subject may already overwrite all data and revoke all capabilities held by the target. Both of these operations modify the target object adding them to the *wroteTo* set.

The *allocate* operation models new object allocation. As allocation is modeled as part of the universal TCB, whether in the kernel or as an application, the *allocate* operation does not require a capability to perform. It requires only that the object to be allocated is in the *unborn* state. During allocation, the allocator specifies the new object's initial data and capabilities, which is encoded using a pairwise map from source index to target index. Although the operation encodes which object is to be

Figure 3.7 Definition of *mutated*.

$$
\begin{aligned}
mutated(E, S_0) &\equiv E \\
mutated(E, S_0 \ldots \xrightarrow{op_{m-1}} S_{m-1} \xrightarrow{op_m} S_m) &\equiv \\
\text{let } M = mutated(E, S_0 \ldots \xrightarrow{op_{m-1}} S_{m-1}) &\text{ in} \\
M &\mid \text{ if operation preconditions are not met} \\
M &\mid \text{ if } E \cap readFrom(op, S_{m-1}) = \emptyset \\
M \cup wroteTo(op, S_{m-1}) &\mid \text{ otherwise}
\end{aligned}
$$

allocated, this is not considered visible to or within the control of the allocator. Once allocated, the allocator receives a capability with total authority of the fresh object. The information flow requirements add the allocated object to the *wroteTo* set.

The *send* operation models the mechanism of communication and the protection extension mechanism. A capability with the *tx* access right permits its holder to transmit a message containing both capabilities and data to the target, optionally fabricating a reply capability for use with client-server models. The transfer is encoded as with the *allocate* operation. In real system implementations, the system is expected to implement a rendezvous mechanism allowing the recipient to specified where the data and capabilities will be stored. As *send* transfers both data and capabilities, the target object is in the *wroteTo* set.

Operation sequences are simply executed sequentially over the system state. Because operation preconditions preclude erroneous transitions, they can be composed automatically. Tracking information flow through operation sequences is computed by the *mutated* function in Figure 3.7. *mutated* considers any subsystem to be self-mutating as a base case. For each operation successfully performed, *mutated* increases

Figure 3.8 Direct access graph.

$$\begin{aligned} src \xrightarrow{ar} tgt \in dirAcc(S) \iff & \\ & isAlive(src, S) \wedge \exists arset, ar \in arset \wedge \\ & hasCap(src, mkCap(tgt, arset), S) \wedge \\ & isAlive(tgt, S)) \end{aligned}$$

what was mutated by the *wroteTo* set if the intersection of the *readFrom* set and the mutated set are non-empty.

3.3 Access Graphs and Potential Access

SDM uses *access graphs* to reason about nearly all safety and information flow properties. Access graphs reduce system states to access relations between objects, representing multiple system states simultaneously. Structurally, an *access graph* is simply a finite set of access edges, each a triple in $Ref \times Ref \times R$. The access edge denoted $src \xrightarrow{ar} tgt$ indicates that object *src* holds right *ar* to object *tgt*. As the access graph is a set, each edge appears only once, collapsing the amount of redundant information.

A *direct access graph* is an access graph representing the permission information of a specific system state. The direct access graph of a system state does not include capabilities held by unborn or dead objects as these capabilities may not be transferred or invoked. The direct access graph function *dirAcc* is described by Figure 3.8.

The next major goal is to define an upper bound on the worst-case authority present in an access graph that can be used when verifying properties about access

Figure 3.9 *transfer* definition.

$$
transfer(A,B) \iff \begin{cases}
src \xrightarrow{ar} tgt \in A & \wedge \quad add(src \xrightarrow{ar'} src, A) = B \\
src \xrightarrow{ar} tgt \in A & \wedge \quad add(tgt \xrightarrow{ar'} tgt, A) = B \\
src \xrightarrow{ar} tgt \in A & \wedge \quad add(tgt \xrightarrow{ar'} tgt, A) = B \\
src \xrightarrow{rd} tgt \in A & \wedge \quad tgt \xrightarrow{ar} tgt' \in A \wedge \\
 & \quad add(src \xrightarrow{ar} tgt', A) = B \\
src \xrightarrow{wr} tgt \in A & \wedge \quad src \xrightarrow{ar} tgt' \in A \wedge \\
 & \quad add(tgt \xrightarrow{ar} tgt', A) = B \\
src \xrightarrow{tx} tgt \in A & \wedge \quad src \xrightarrow{ar} tgt' \in A \wedge \\
 & \quad add(tgt \xrightarrow{ar} tgt', A) = B \\
src \xrightarrow{tx} tgt \in A & \wedge \quad add(tgt \xrightarrow{tx} src, A) = B \\
src \xrightarrow{wk} tgt \in A & \wedge \quad tgt \xrightarrow{ar} tgt' \in A \wedge \\
 & \quad (ar = wk \vee ar = rd) \wedge \\
 & \quad add(src \xrightarrow{wk} tgt', A) = B
\end{cases}
$$

Figure 3.10 Potential transfer definition.

$$
potTransfer(A,C) \iff \begin{cases}
A = C \\
\exists B, potTransfer(A,B) \wedge transfer(B,C)
\end{cases}
$$

Figure 3.11 Maximal and potential access .

$$
\begin{aligned}
maximal(P) &\equiv \forall A, potTransfer(P,A) \Rightarrow P = A \\
potAcc(I,P) &\equiv potTransfer(I,P) \wedge maximal(P)
\end{aligned}
$$

and data motion. The definition of worst-case authority is built on *transfer* in Figure 3.9. *Transfer* is a micro-operation of permission transfer based on access graphs. Unlike the operational semantics which operates at the granularity of whole capabilities, transfer justifies a single permission transfer. If A and B are access graphs, *transfer*(A, B) indicates that a permission transfer is possible from A to B through the addition of some edge. Two access graphs related by any, potentially empty, sequence of transfer steps is defined by the *potential transfer* relation *potTransfer* in Figure 3.10.

Access graphs are related by *transfer* based entirely on individual access rights. The *rd* and *wr* access right authorize edges to be transferred in opposite directions. Similar to the *rd* case, the *wk* access right is authorized to transfer a *wk* edge from a *wk* or *rd* edge. The *tx* access right behaves like the *wr* permission but includes a second case for constructing a reply. To make *transfer* a reflexive relation, two cases exist to permit self-targeting edges. These cases require some other edge to refer to the objects to prevent the addition of new object references and keep analysis finite.

It is possible to construct a least upper bound between any two access graphs which share an initial access graph. Given an initial access graph I and access graphs A and B such that *potTransfer*(I, A) and *potTransfer*(I, B), there must exist an access graph C such that *potTransfer*(A, C) and *potTransfer*(B, C). Transfer captures the ability to add a single edge using existing edges as justification. Therefore, for any access graph C, the underlying justification is not altered by adding edges to C.

Figure 3.12 Potential access always exists.

$$\forall I, \exists P, \mathit{potAcc}(I, P)$$

Transfer may be transposed with set addition: if $\mathit{transfer}(C, D)$ and $\mathit{add}(x, D) = E$, then $\mathit{add}(x, C) = D'$ and $\mathit{transfer}(D', E)$. Because *transfer* and *potTransfer* are non-decreasing, they are commutative. The least upper bound is easily computed by set union, and all transfers performed are still valid.

From these definitions, each access graph must have a supremum by *potTransfer*. This *potential access* graph is the worst-case approximation of access in an initial access graph. Defined by *potAcc* in Figure 3.11, it is the access graph that is reachable via *potTransfer* and is *maximal*. If I is an initial access graph, then P is a potential access graph of I if and only if $\mathit{potAcc}(I, P)$. Because all access graphs have a maximal access graph and have a least upper bound, any potential access graph must be the supremum. From set union over finite sets, it follows that all potential access graphs of I are unique.

Computing the potential access graph can be performed by induction over the *complete access graph*, the graph containing all possible edges given the object references already present. Because each transfer adds edges between previously existing objects, it cannot exceed the complete access graph. The complete access graph less the initial graph can be used as a work list when considering a new potential edge. Exhaustively scanning this list for candidate edges, testing them with transfer, and recursing on the resulting set will eventually produce the potential access graph. As the set difference between the complete access graph and the accumulator is always decreasing, this computation is guaranteed to eventually terminate. potential access it must always exist, as in Figure 3.12, because it is computed by a function on I. Therefore, the remainder of this sketch will use symbol *potAcc* as both the computable function and the judgment.

Analysis in the remaining sections relies on the ability to preserve and reorder *transfer*s with other approximating functions. Because the family of *transfer* functions are themselves additive, they can be transposed without loss of generality. This provides a mechanism for describing different approaches to computing potential access with respect to related access graphs.

3.4 Access Approximations

All operations in SDM have the potential to overwrite capabilities and some delete them outright. Computing precise functions between direct and potential access graphs describing these system states introduces complexity, so SDM defines the concept of “conservatively approximating” functions. Conservatively approximating functions must be composable so that they remain approximating inductively over multiple operations. They must compose with set addition to permit transfers to be reordered around them. The graph in Figure 3.13 illustrates this concept.

Figure 3.13 Direct access approximation F_{dirAcc}.

$$\begin{tikzcd}
\boxed{S_a} \arrow[r, "dirAcc"] \arrow[d, "op"'] & I_a \arrow[r, dash, "\subseteq"] & I'_a \arrow[d, "F_{dirAcc}(S_a)"'] \\
\boxed{S_b} \arrow[r, "dirAcc"] & I_b \arrow[r, dash, "\subseteq"] & I'_b
\end{tikzcd}$$

□ system state ——— relation

◯ access graph ⟶ computable relation

Figure 3.14 Potential access approximation F_{potAcc}.

$$\begin{tikzcd}
\boxed{S_a} \arrow[r, "dirAcc"] & I_a \arrow[r, dash, "\subseteq"] & I'_a \arrow[r, "potAcc"] \arrow[d, "F_{dirAcc}(S_a)"'] & P_a \arrow[r, dash, "\subseteq"] & P'_a \arrow[d, "F_{potAcc}(S_a)"'] \\
 & & I'_b \arrow[r, "potAcc"] & P_b \arrow[r, dash, "\subseteq"] & P'_b
\end{tikzcd}$$

Theorems of this sort are difficult to read, but easy to comprehend through illustration. In access relationship graphs, functions are represented as arrows and relations as lines. System states are represented by squares with shadows and access graphs with circles with shadows.

The simplest approximation is one over direct access graphs. A direct access approximation, F_{dirAcc}, is a monotonically non-decreasing function between direct access graphs indexed by operation and initial system state. A potential access approximation is defined similarly over a direct access approximation, but with sufficient information to recover the initial system state Figure 3.14 details the relationships visually.

Figure 3.15 Safety induction strategy.

$$\begin{tikzcd}
\boxed{S_0} \arrow[r, "dirAcc"] \arrow[d, "op_1"'] & I_0 \arrow[r, dash, "\subseteq"] & I'_0 \arrow[r, "potAcc"] \arrow[d, "F_{dirAcc_1}(S_0)"'] & P_0 \arrow[r, dash, "\subseteq"] & P'_0 \arrow[d, "F_{potAcc_1}(S_0)"'] \\
\boxed{S_1} \arrow[r, "dirAcc"] \arrow[d, "op_{N-1}"'] & I_1 \arrow[r, dash, "\subseteq"] & I'_1 \arrow[r, "potAcc"] \arrow[d, "F_{dirAcc_{N-1}}(S_{N-1})"'] & P_1 \arrow[r, dash, "\subseteq"] & P'_1 \arrow[d, "F_{dirAcc_{N-1}}(S_{N-1})"'] \\
\cdots \arrow[d, "op_N"'] & \cdots & \cdots \arrow[d, "F_{dirAcc_N}(S_N)"'] & \cdots & \cdots \arrow[d, "F_{dirAcc_N}(S_N)"'] \\
\boxed{S_N} \arrow[r, "dirAcc"] & I_N \arrow[r, dash, "\subseteq"] & I'_N \arrow[r, "potAcc"] & P_N \arrow[r, dash, "\subseteq"] & P'_N
\end{tikzcd}$$

Figure 3.16 Direct access operation.

$$\begin{aligned}
\mathit{dirAccOp}(S, \mathit{read}(a,t)) &\equiv \mathit{id}\\
\mathit{dirAccOp}(S, \mathit{write}(a,t)) &\equiv \mathit{id}\\
\mathit{dirAccOp}(S, \mathit{revoke}(a,t,i)) &\equiv \mathit{id}\\
\mathit{dirAccOp}(S, \mathit{destroy}(a,t)) &\equiv \mathit{id}\\
\mathit{dirAccOp}(S, \mathit{fetch}(a,t,c,i)) &\equiv \mathit{edgeCopy}(S, \mathit{objTarget}(a,t,S), a, ((c,i)))\\
\mathit{dirAccOp}(S, \mathit{store}(a,t,c,i)) &\equiv \mathit{edgeCopy}(S, a, \mathit{objTarget}(a,t,S), ((c,i)))\\
\mathit{dirAccOp}(S, \mathit{send}(a,t,m,x)) &\equiv \mathit{edgeCopy}(S,a,t,m) \circ \mathit{reply}(x,a,t)\\
\mathit{dirAccOp}(S, \mathit{allocate}(a,n,m,\mathit{typ})) &\equiv \mathit{edgeCopy}(S,a,n,m) \circ \mathit{insert}(a,n)
\end{aligned}$$

when preconditions are satisfied, otherwise

$$\mathit{dirAccOp}(S, \mathit{op}) \equiv \mathit{id}$$

with

$$\mathit{id}(A) \equiv A$$

From these pieces, approximations for a sequence of operations can be assembled as shown in Figure 3.15.

Figure 3.16 defines *dirAccOp*, the function approximating direct access graphs over operations. Approximations of worst-case authority do not need to consider the elimination of permissions;. Therefore, all operations whose sole effect is to remove capabilities are approximated by an identity function. The other direct access approximations correspond directly to the potential additional access rights for each operation.

The definition has omitted the simple functions *edgeCopy*, *reply*, and *insert*. If $a \overset{S}{\mapsto} aObj$ and $t \overset{S}{\mapsto} tObj$, then $\mathit{edgeCopy}(S,a,t,m)$ adds access edges corresponding to updating *tObj* as $\mathit{multiCapCopy}(aObj, tObj, m)$. That is, it examines S and adds

Figure 3.17 Potential access operation.

when preconditions are satisfied:

$$\begin{aligned} \mathit{potAccOp}(S, \mathit{allocate}(a, n, m, \mathit{typ}))(P) &\equiv \mathit{endow}(a, n) \\ \mathit{potAccOp}(S, \mathit{op})(P) &\equiv P \qquad \text{for } \mathit{op} \neq \mathit{allocate} \end{aligned}$$

when preconditions are not satisfied:

$$\mathit{potAccOp}(S, \mathit{allocate}(a, n, m, \mathit{typ})) \equiv \mathit{id}$$

with

$$\mathit{endow}(a, n) \equiv \mathit{potAcc} \circ \mathit{insert}(a, n)$$

edges to a corresponding to all capabilities in t at the indices in the first position in m. *reply* is analogous to *replyCap* and adds the edge $a \xrightarrow{tx} t$ when x is *True*. The *insert*(a, n) function adds the new object n to the access graph by adding all possible edges between a and n.

With the exception of *allocate*, the other operations are approximated by a function built from *potTransfer*. The special case to approximate the *allocate* operation cannot be modeled using *potTransfer* because it adds a new object. The *endow* function is defined to approximate *allocate* and is defined in two parts. First, *endow* invokes *insert*, which adds all possible edges between parent and child. Having accomplished this, it then computes *potAcc* covering the all reflexive edges and any transfers that could occur. Section 3.5 will discuss the relevance of *endow* in greater detail.

As potential access is the least upper bound over *potTransfer*, each *dirAccOp* that is captured by potential transfer, i.e. the non-allocate operations, may be approx-

Figure 3.18 Access graph projection.

$$projection(a, n)(P, P') \equiv$$

$$\forall(src, tgt, ar), src \xrightarrow{ar} tgt \in P' \iff \begin{cases} src \xrightarrow{ar} tgt \in P \\ src \xrightarrow{ar} a \in P \land tgt = n \\ a \xrightarrow{ar} tgt \in P \land src = n \\ src = a \in P \land tgt = a \\ src = n \in P \land tgt = a \\ src = a \in P \land tgt = n \\ src = n \in P \land tgt = n \end{cases}$$

imated by an identity function over potential access. Because the *endow* function recomputes potential access, it must approximate the capability copies during the *allocate* operation. This definition demonstrates that the only access not approximated by potential access occurs during object allocation.

3.5 Projections and Safety

access graph projections define a mechanism describing how potential access operations evolve with new objects. Endow is the only non-trivial potential access operation. Consider the result of fully connecting a fresh object to an existing one as in the case of *endow*. Any edges held by the original object might come to be be held by the allocated object through transmission. Additionally, some edges originally targeting the allocating object might be added such that they target the fresh object. It is also possible that new self-referential edges may come to exist, along with some other uninteresting corner cases. Any access graph related by these properties is called a *projection*, using the *projection* relation given in Figure 3.18.

Figure 3.19 Lemma: projection approximates endow.

$$\forall P, \mathit{maximal}(P) \Rightarrow \mathit{projection}(a,n)(P,P') \Rightarrow \mathit{endow}(a,n)(P) \subseteq P'$$

Figure 3.20 Attenuating authority for capability systems.

$$\begin{array}{l}\forall S, P, \mathit{potAcc}(\mathit{dirAcc}(S), P) \Rightarrow \\ \forall E, (\forall e \in E, \neg \mathit{isUnborn}(e,S)) \Rightarrow \\ \forall e \in E \Rightarrow \\ \forall o \notin E \Rightarrow \\ \forall P', \mathit{projection}(e,o)(P,P') \Rightarrow \\ \mathit{restrict}(P', E) \subseteq P\end{array}$$

The lemma in Figure 3.19 states than an *endow* operation performed on a maximal access graph must form a projection. This proof requires a great degree of case analysis, but is conceptually very simple. Consider any access edge authorized by a transfer after a projection. This edge must contain the child object reference, either as source or target. If this were not the case, there must exist other edges in the graph prior to projection which would authorize the new edge without the presence of the child. However, this graph was assumed to be maximal, making this impossible. Therefore, *endow* is approximated by a projection to the fresh object when operating on a maximal access graph.

Finally, the safety property can be described using potential access through projection. The safety decision is initially determined by potential access. As the system is not in a position to know the relationships that new objects will have, it cannot determine what access they may come to hold. However, the existence of all new objects can be approximated via projection. Projection only adds edges which re-

Figure 3.21 Definition of mutable.

$$\textit{mutable}(E)(A) \equiv$$
$$\{m | \exists e \in E \wedge (m \xrightarrow{wk} e \in A \vee m \xrightarrow{rd} e \in A \vee e \xrightarrow{wr} m \in A \vee e \xrightarrow{tx} m \in A)\}$$

late the system to new objects, leaving all existing potential access unchanged. The *restrict*(P, E) operation eliminates all edges in P with both elements not in E and is used to compare pre-existing relationships. The potential access of the system is preserved for all existing relationships and remains maximal over the life of the system. This property, stated formally in Figure 3.19, is called *attenuating authority* and represents a decision to the safety problem for capability-based systems.

3.6 Mutability

The definition of the *confinement test* for object-capability systems relies upon a decision over permissions, not over information flow. To reason about security policies expressed using permissions, it must be the case that permissions are representative of information flow. Surprisingly, this is not true for most systems [HRU76]. However, this property does hold for object-capability systems satisfying SDM.

The definition *mutable* in Figure 3.21 determines the objects where information in a given subsystem might flow. It is computed by induction over the edges of an access graph. Objects are *mutable* by a subsystem in three cases. In the base case, the subsystem is self-mutating and is included in *mutable*. Second, writing or transmitting data push information out of the subsystem, so any target of a *wr* or *tx*

Figure 3.22 Theorem: mutable approximates mutated.

$$\mathit{potAcc}(\mathit{dirAcc}(S_0), P) \Rightarrow \mathit{mutated}(E, S_0 \ldots \xrightarrow{op_m} S_m) \subseteq \mathit{mutable}(E)(P) \cap \mathit{extant}(S_0)$$

where

$$\mathit{extant}(S) \equiv \{o | \mathit{isAlive}(o, S) \wedge \mathit{isDead}(o, S)\}$$

edges held by an element of the subsystem is *mutable*. Finally, any object holding *rd* or *wk* edges to a member of the subsystem can pull information out of the subsystem. Because *mutable* is parameterized over any access graph, it is applied to the direct access or potential access of a system to produce respective meaning. The terms *direct mutability* and *potential mutability* describe the mutability of direct access or potential access graphs.

Mutable preserves subset variance with both the subsystem and with the access graph. Though not formally presented, these variance properties form the basis for most approximations of mutability in the rest of this sketch.

For *mutable* to be meaningful, it must satisfy the theorem in Figure 3.22. This theorem states that what is mutated[1] by a subsystem over any execution, when restricted to initially extant objects, is conservatively approximated by what is potentially mutable from the initial configuration. Naively, this would be directly satisfiable by induction. All objects require a capability for data motion. These capabilities are conservatively approximated by direct access, which in turn is conservatively approximated by potential access. The allocate operation is safe because each projection

[1]Recall Figure 3.7

Figure 3.23 Definition of mutable induction.

$$\begin{tikzcd}
S_0 \arrow[r, "dirAcc"] \arrow[d, "op_1"'] & I_0 \arrow[r, dash, "\subseteq"] & I'_0 \arrow[r, "F^g"] \arrow[d, "dirAccOp_1"'] & P_0 \arrow[r, dash, "\subseteq"] & P'_0 \arrow[r, "mutable(E)"] \arrow[d, "F^p_1"'] & M_0 \arrow[d, "mutableInd(S_0, op_1)"'] \\
\cdots \arrow[d, "op_{N-1}"'] & \cdots & \cdots \arrow[d, "dirAccOp_{N-1}"'] & \cdots & \cdots \arrow[d, "F^p_{N-1}"'] & \cdots \arrow[d, "mutableInd(S_{N-1}, op_{N-1})"'] \\
S_{N-1} \arrow[r, "dirAcc"] \arrow[d, "op_N"'] & I_{N-1} \arrow[r, dash, "\subseteq"] & I'_{N-1} \arrow[r, "F^g"] \arrow[d, "dirAccOp_N"'] & P_{N-1} \arrow[r, dash, "\subseteq"] & P'_{N-1} \arrow[r, "mutable(M_{N-2})"] \arrow[d, "F^p_N"'] & M_{N-1} \arrow[d, "mutableInd(S_n, op_n)"'] \\
S_N \arrow[r, "dirAcc"] & I_N \arrow[r, dash, "\subseteq"] & I'_N \arrow[r, "F^g"] & P_N \arrow[r, dash, "\subseteq"] & P'_N \arrow[r, "mutable(M_{N-1})"] & M_N
\end{tikzcd}$$

only extends the allocator's mutability into the child.

The naive approach is hiding a subtle induction problem. It relies on the safety property for its inductive explanation of why *mutable* was not exceeded by *mutated*. However, the inductive definition of computing *mutable* does not match the inductive definition of *mutated*. Figure 3.23 defines an inductive definition of *mutable*, *mutableInd*, matching the induction of *mutated*. This inductive specification of what is mutable must always conservatively approximate what is mutated. Potential inductive mutability only grows by the newly allocated object exactly when the parent is in the inductively mutable subsystem. By distributing intersection across union, all objects that were not initially extant are excluded from this set. Therefore, the static definition of *mutable* conservatively approximates mutation over the life of the system.

Figure 3.24 Extant subsystems.

$$\mathit{extantSub}(S, E) \equiv E \subseteq \mathit{extant}(S)$$

Figure 3.25 Constructive subsystems.

$$\mathit{constructiveSub}(S, E) \equiv \forall \mathit{src} \overset{S}{\mapsto} (\mathit{obj}, \_, \_), i \overset{\mathit{obj}}{\longmapsto} \mathit{mkCap}(\mathit{tgt}, \_) \land \mathit{tgt} \in E \Rightarrow \mathit{src} \in E$$

3.7 Subsystem Refinements

The definition of subsystems heretofore has been a simple set of object references. This is convenient, as many proofs do not rely upon any information about the form of a subsystem. However, this general definition is insufficient for the confinement test as real subsystems are necessarily more constrained. This proof sketch defines two additional predicates of subsystems in addition to the confinement test.

The semantics do not make any guarantees about the allocation relationships of unborn objects. Any unborn object might be legally allocated as part of an allocate operation and subsequently become the child of any other object. The inclusion of unborn objects in a subsystem can inadvertently link two otherwise independent subsystems through an allocation, as is the case in the SW model [SW00]. Rather than make assumptions about where new objects will arise, subsystems are restricted consisting of only alive or dead objects. These subsystems are called *extant subsystems* and are defined in Figure 3.24.

Since the confinement test is always performed before subsystem construction, the

Figure 3.26 Confinement predicates.

$$\begin{aligned}
\mathit{authorizedSet}(C,E) &\equiv \forall \mathit{mkCap}(\mathit{tgt}, \_) \in C \Rightarrow \mathit{tgt} \notin E\\
\mathit{confinementTest}(S,E,C) &\equiv \forall e \in E, e \overset{S}{\mapsto} \mathit{eObj}, \_ \xmapsto{\mathit{eObj}} \mathit{mkCap}(\mathit{tgt}, \mathit{arset}) \Rightarrow\\
&\quad\ \mathit{mkCap}(\mathit{tgt}, \mathit{arset}) \in C \vee\\
&\quad\ \mathit{arset} = \emptyset \vee\\
&\quad\ \mathit{tgt} \in E \vee\\
&\quad\ \neg \mathit{isAlive}(\mathit{tgt}, S) \vee\\
&\quad\ \mathit{tgt} \notin E \wedge \mathit{arset} = \{\mathit{wk}\}
\end{aligned}$$

Figure 3.27 Confinement definition.

$$\begin{aligned}
\mathit{confinedSub}(S,E,C) &\equiv \mathit{authorizedSet}(C,E) \wedge\\
&\quad\ \mathit{extantSub}(S,E) \wedge\\
&\quad\ \mathit{constructiveSub}(S,E) \wedge\\
&\quad\ \mathit{confinementTest}(S,E,C)
\end{aligned}$$

subsystem cannot have yet interacted with the system in any way. Additionally, the constructor is obligated to revoke its authority to the newly fabricated subsystem and must not have passed it on elsewhere. The definition in Figure 3.25 generalizes this concept to extend beyond the trusted constructor, requiring that there must not exist a capability held outside the subsystem that names an element within the subsystem. These subsystems are called *constructive subsystems* as they arise naturally from construction.

3.8 Confinement

A subsystem is confined exactly when all outward information flow is authorized. That is, regardless of the actual structure of the subsystem, all potential outward

Figure 3.28 The fully authorized access graph.

$$\begin{aligned} \mathit{fullAuthAG}(A, E, C) &\equiv \mathit{completeAG}(E) \cup \\ &\quad\ \mathit{authAG}(E, C) \cup \\ &\quad\ \mathit{restrict}(A, \mathit{agObjRefs}(A) - E) \end{aligned}$$

with

$$\begin{aligned} \mathit{authAG}(E, C) &\equiv \{src \xrightarrow{ar} tgt | src \in E \wedge \mathit{mkCap}(src, arset) \in C \wedge ar \in arset\} \\ \mathit{agObjRefs}(A) &\equiv \{a | a \dashrightarrow \_ \in A \vee \_ \dashrightarrow a \in A\} \end{aligned}$$

information flow is derived by capabilities in the *authorized set*. This sketch embeds the confinement test as a post-condition on the system, but it should be noted that this test can be performed by previous conditions and local inspection. In addition to being extant and constructive, the authorized set of capabilities must not target elements of E and the confinement test must pass. The confinement test in Figure 3.26 is almost a direct transcription of the constructor's confinement test from Section 2.7, without the case admitting recursively confined constructors. The complete definition of a confined subsystem is given in Figure 3.27.

To describe confinement as a system property, this proof sketch defines how the authorized set of capabilities comes to authorize information flow. A *fully authorized subsystem* is one in which all objects hold: 1) fully permissive capabilities to all objects in the subsystem and 2) all of the authorized set of capabilities. The confinement proof proceeds by fixing the subsystem set E before considering subsystems with varied sets of objects. Rather than choosing a canonical subsystem, confinement is described using access graphs.

The *fully authorized access graph* represents all fully authorized subsystems with

Figure 3.29 Confinement for access graphs.

$$\begin{aligned}
\mathit{agSimpConf}(E)(P_{\mathit{base}}, P_{\mathit{conf}}) &\equiv P_{\mathit{base}} \subseteq P_{\mathit{conf}} \wedge \\
&\quad (\forall \mathit{src} \xrightarrow{\mathit{ar}} \mathit{tgt} \in (P_{\mathit{conf}} - P_{\mathit{base}}), \mathit{src} = \mathit{tgt} \vee \\
&\quad \mathit{ar} = \mathit{wk} \wedge \mathit{src} \in E \wedge \mathit{tgt} \notin E) \\
\mathit{agConf}(E)(P_{\mathit{base}}, P_{\mathit{conf}}) &\equiv P_{\mathit{base}} \subseteq P_{\mathit{conf}} \wedge \\
&\quad (\forall \mathit{src} \xrightarrow{\mathit{ar}} \mathit{tgt} \in (P_{\mathit{conf}} - P_{\mathit{base}}), \mathit{src} = \mathit{tgt} \vee \\
&\quad \mathit{ar} = \mathit{wk} \wedge \mathit{exFlow}(P, E, \mathit{src}) \wedge \neg \mathit{exFlow}(P, E, \mathit{tgt})) \\
\text{with} & \\
\mathit{exFlow}(A, o, s) &\equiv s \in \mathit{mutable}(A)(\{o\})
\end{aligned}$$

the same collection of objects constructed from an initial system state. Given an access graph, *fullAuthAG* in Figure 3.28 returns an access graph where E is fully connected, all elements of E contain the authorized set of alive objects, and the edges in the original access graph are restricted to elements not in E. This last clause, performed by the *restrict* function, removes all edges where either the source or target are not elements of an approved set of object references

The confinement test is lifted to access graphs as *agSimpConf* in Figure 3.29. Confinement permits more access than is authorized, but ensures that this access creates no additional information flow. For access graphs, it is stated as a comparison between a *base* access graph and a *confined* access graph. The base access graph is a subset of the confined access graph and restricts which additional edges are in the confined access graph. By inspection, a fully authorized access graph resulting from the direct access of a system state with a confined subsystem E will satisfy *agSimpConf* over the same parameters.

Unfortunately, computing potential access on a simply confined access graph will not preserve *agSimpConf*. The more general predicate *agConf* solves this problem. *agConf* subsumes *agSimpConf* and is also preserved through potential transfers, and ultimately potential access. The definition of *agConf* relies on the definition of *exFlow*, which captures the existence of point-wise mutability. When the base access graph is the potential access graph of a fully authorized access graph, all authorized information flow has been captured by *mutable*. Therefore, *agConf* preserves mutability by restricting which edges may be added to the confined access graph. It requires that all edges not in the base access graph must be *wk* edges where there exists an information flow from the confined subsystem to the edge source and there are no flows from the confined subsystem to the edge target, or the edge is impotently self-targeting. By case analysis, the mutability of these two access graphs must be identical.

Figure 3.30 Visualization of the confinement lemma

Given $A = \mathit{fullAuthAG}(I, C, E)$ and $\mathit{confinedSub}(S, C, E)$,

$$\begin{tikzcd}
S \arrow[r, "dirAcc"] & D \arrow[rrr, "potAcc"] \arrow[dr, dash, "\subseteq"'] & & & P_D \arrow[rr, "mutable(E)"] \arrow[dr, dash, "\subseteq"'] & & M_D \arrow[dr, dash, "\subseteq"'] & \\
& & I \arrow[r, "potTransfer"] \arrow[d, dash, "agSimpConf(E)"'] & I' \arrow[rr, "potAcc"] \arrow[d, dash, "agConf(E)"'] & & P_I \arrow[rr, "mutable(E)"] \arrow[dll, dash, "agConf(E)"] & & M_I \arrow[d, dash, "="'] \\
& & A \arrow[r, "potAcc"] & P_A \arrow[rrrr, "mutable(E)"] & & & & M_A
\end{tikzcd}$$

The overall proof of confinement is visually described in Figure 3.30. The top row illustrates the computation of the potential mutability of a system state with subsystem E confined to authorized set C. Likewise, the bottom row describes the computation of the potential mutability of the fully authorized access graph A. These computations are related by the middle row with values that preserve information flow satisfying confinement.

The relationships between the bottom and middle rows form the majority of the confinement verification. The right-most property has already been described in the description of *agConf*. Given, $agConf(P_{base})(P_{conf},,)$ the mutability of P_{base} and P_{conf} are identical. The left-most property can be validated directly from previous theorems. First, as previously mentioned, *agSimpConf* is subsumed by *agConf*. Second, all access edges valid for *transfer* in the base are also valid in the confined access graph. Therefore, these edges may be added to I to produce a valid potential transfer to I'. By inspection, adding any access edge to both the base and confined access graph preserves *agConf*. Consequently, *agConf* must also hold in this specific case.

Once the base access graph is maximal, the middle triangle becomes solvable. The definition of *agConf* only permits new *wk* edges that don't create new information flow. Intuitively, *wk* edges only propagate other *wk* edges in *transfer*. When initially constrained by *agConf*, the *transfer* case for *wk* edges can not violate *agConf*. Therefore, *agConf* with a maximal base access graph must hold while computing the potential access of the confined access graph.

Having discharged the bottom row, the relationship between the top row and the middle row is demonstrated by subset variance. Simply choosing $I = D \cup A$ will satisfy both $D \subseteq I$ and $\mathit{agSimpConf}(E)(A, I)$. With this initial condition, *potAcc* preserves subset relationships which are then preserved by *mutable*.

Therefore, any subsystem E passing the confinement test is confined to the fully authorized access graph. When the confinement test succeeds, all outward information flow that is possible from the yield at the moment allocation occurs is the sole consequence of the capabilities provided in the authorized set.

Though not formally presented in this sketch, the confinement proof can be extended to cover any set of objects. As E varies, the mutability of two fully authorized access graphs does not change with respect to E, provided E does not include additional objects originally extant in the underlying system state. Therefore, the choice of E is irrelevant and all possible subsystems are confined.

Chapter 4

The Coq Proof Assistant

The SDM verification is formalized using the Coq proof assistant. This chapter begins by motivating Coq as a good candidate for a verification system. It then presents a brief overview of the term language of Coq with a focus on some the features used in SDM.

4.1 Why Coq

This section discusses the reasons Coq was chosen as the mechanized verifier for SDM. It describes how higher-order logics can simplify first-order problems by permitting developers to generalize proofs. It argues that intuitionistic logics are natural choices for automated verification, as truth values are represented as programs inhabiting a type. It discusses how Coq implements both of these features and articulates the useful features of the Coq standard library for the SDM verification.

The high degree of automation available in first-order systems makes them appear very desirable. However, the ability to generalize theorems in higher-order proof systems makes them more practical than first-order systems. Any finite problem can be expressed in a first-order logic. However, without the ability to generalize theorems, developers quickly become burdened with a multitude of specialized proof obligations. Many first-order proof systems offer the ability to write functions to generalize these proofs. However, these functions cannot be verified, increasing the unverified surface of the proof and consequently undermining confidence. Although the safety problem and confinement policy for capability-based systems can be expressed as first-order problems, this verification effort uses Coq because it is a higher-order proof assistant.

Most higher-order logics available fit into two categories: classical higher-order predicate logic and intuitionistic type theory. For proofs that can be automatically verified, intuitionistic and classical logics have the same expressive power. They differ in how they view the concept of truth. Classical logics reason about truth and falsehood directly, relying upon meta-analysis like term rewriting to prove a theorem. Intuitionistic logic reasons about construction and refutation directly, requiring a witness for every truth. These witnesses make statements in intuitionistic logic stronger than classical logic.

The concept of how truth is constructed directly informs computational verification. Proofs as constructions and proofs as meta-inferences necessarily operate in different ways. Constructions and theorems have a direct relationship with software terms and types according to the Curry-Howard isomorphism. In Coq, proofs as programs can be directly written by the developer and can be directly manipulated by other programs without resorting to meta-theory, simple type checking may suffice. This reduces the amount of meta-logic necessary to express properties and forms very natural proofs.

Verifying a theorem in Coq is the act of constructing a program satisfying a type representing the theorem. To any developer acquainted with parametric polymorphism and (co)inductive data types, reading and manipulating these programs will be familiar. Base definitions and functions are executable programs that can be readily understood by developers unfamiliar with proofs, reducing mental overhead. While specifying a program precisely can give the developer a great deal of power, Coq also includes a wide array of tools to help construct programs automatically. Coq includes a tactic meta-language and pattern-matching system to assist the developer when searching for a program. These are combined into a hint system that can be combined to produce a high degree of automation for domains that are highly syntactically driven.

The SDM verification utilizes a number of features of the Coq standard library. Boolean decidability is used to place most propositions into Boolean logic, making them computationally decidable. Decidability is a critical problem because undecidability hides the unknowable; a theorem with an unknowable assumption can still be verified. By ensuring that all propositional hypotheses are isomorphic to Boolean functions, SDM ensures that the results are always known.

The Coq standard library also provides meta-theory support for equivalence relations. The rewrite tactic in Coq can perform automated rewriting of any equivalence relation, not just built-in equality. It requires only that relevant terms respect the equivalence relation for relevant types. This permits the model to reasoning about potentially different, but semantically identical, types and terms.

The last major features of the Coq standard library utilized by SDM are the axiom-free finite set and map libraries. [FL04] These are very large productions modeled after the OCaml Set and Map libraries. They include a collection of supplemental libraries containing useful theorems pertinent to their interface as well as implementations including a fully concrete definition using lists.

4.2 The Coq Term Language

Chapters 5 to 9 contain a detailed theorem walk-through of the confinement verification in Coq. As previously stated in Section 1.3, one goal of this dissertation is to produce confidence in the confinement verfification result. This dissertation presumes that reviewers have confidence in the proof assistant and it therefore does not discuss the mechanics of the proof construction. As such, this section focuses exclusively on the useful portions of the Gallina term language of Coq.

Gallina is a higher-order functional language and constructive dependent type system that implements higher-order intuitionistic logic. The syntax and semantics of Gallina are based on OCaml functions and data-types, but with dependently typed parametric polymorphism. It contains the usual functions, anonymous functions, and fixpoints along with cofixpoints (sometimes called lazy fixpoints). (Co)Inductive types are a generalized notion of (co)data-type with (co)constructor definitions. Gallina also includes a highly type-generalized pattern matching system for (co)constructors.

All terms in Coq belong to one of three sorts: `Set`, `Prop`, and `Type`. `Set` is the sort of "specifications" or programs, and `Prop` is the sort of "propositions" or theorems. The difference between the two is how they handle the type $* \to *$. The type `Set` → `Set` is not in the sort `Set`, but the type `Prop` → `Prop` is in the sort `Prop`. This makes `Prop` impredicative, in that its terms may be self-defining, and `Set` predicative, in that it is not. The sort `Type` is stratified and somewhat complex. `Type` is used very little in this effort and may be considered parametric for either `Prop` or `Set`.

Unlike general programming languages, all functions in Coq are obliged to terminate. Definitions and anonymous functions do not permit recursion and simply terminate. Fixpoints permit structural recursion that can be automatically inferred. General functions permit the developer to specify both the function and the termination measure, though it may be possible for Coq to infer this as well. Because all functions must terminate, (co)inductive data types are often used to express general propositions as dependent type families.

The connection between programs and proofs can be summarized in relationships with functions and inductive types. Implication and universal quantification are expressed by dependent function types, respectively with or without a named parameter. $A \rightarrow B$ is syntactically the same as $\forall\ (\_{:}A),\ B$ where $\_$ is any free variable. *True* is the universally inhabited type and *False* is the universally uninhabited type. Negation, written $\neg\ A$ is syntactically $A \rightarrow$ *False*.

Most other constructions are inductive types or involve pattern matching. Conjunction, disjunction, and existential quantification are all inductive types. The type (*and* A B), written $A \wedge B$, has one constructor *conj* requiring terms of types A and B. The type (*or* A B), written $A \vee B$, has two constructors *proj1* and *proj2* requiring only one term of type A or B, respectively. Existential quantification over a predicate has one introduction constructor, *ex_intro*, that can only be constructed by a witness satisfying the quantified proposition.

4.3 Model Abstraction

SDM is constructed as an abstract implementation to allow it to be used as a framework for future system verifications. Operating systems are not the only capability-based system; capability-based systems also include virtual machines, language runtimes, and distributed systems. As an abstract model, SDM focus on the heart of the confinement problem, producing a result applicable across all domains.

SDM utilizes the Coq module type system as an abstraction mechanism. This decision was motivated by the use of module type abstraction in the axiom-free finite set library. Because abstractions and axioms are the same structure in Coq, SDM also provides a trivial implementation to produce an axiom-free result. It is often the case that the abstract module types are verbatim software from implementation modules. However, it is not possible to produce them by type inference in Coq version 8.3pl2. As a work-around, the project includes simple tool to syntactically create a module type based on each trivial implementation through very simple annotations.

This verification includes the very primitive Perl script "typeify.pl" to automatically produce precise module types from module functors. It does not process the full language of Coq, but processes the commands line-by-line using a very small state transition routine over a very strict module format. The module must contain only one internal module functor, declared on a single line, and the functor parameter list must match the module type parameter list. Theorems are abstracted by replacing the keyword "Theorem" with "Parameter" and removing all lines between commands "Proof." and "Qed." Although Coq allows theorems to nest and elide the "Proof" command, we do not handle these cases. Two commands are supported as comments to provide better abstraction in the generated types. The "(* ABSTRACT *)" command processes a "Definition" into an appropriate "Parameter" allowing other modules to override these definitions with other implementations. The "(* TYPE_REMOVE *)" command eliminates the subsequent line in the generated type altogether and is often used to eliminate helper theorems and lemmas. Any potential errors introduced by this transformation will be caught when the original module against the generated signature.

It is necessary that each module functor and signature be pure, in that all dependencies are completely captured by parametricity. The use of existentially declaring a module type loses type information in Coq 8.3 in ways that would be available in Coq 8.2. Therefore, updated versions of the *FMap* finite map libraries have been constructed to produce appropriate types. The pattern of constructing pure functors from the trivial implementation is prevalent throughout SDM and produces an axiom-free proof with a type signature that can be satisfied in future efforts, While this syntactic type construction is used wherever possible, there are certain portions of the proof which are encoded manually.

The following conventions regarding module names are used throughout this dissertation. Modules that are also files have the **FileModule** font face where as inner modules have the *InnerModule* font face. The locations of each inner module should be clear from the surrounding context. File modules containing functor implementations are suffixed with -**Impl**, while modules constructing a fully complete implementation by functor application are suffixed with -**Appl**. File modules containing type signatures, or those which have no abstraction, are given no suffix. Convenience file modules with supplemental libraries are suffixed with -**_Conv** and are further suffixed

Table 4.1 Declaration and location of common module names.

Instance	Declaration and location
ARSet	*FSet* of *AccessRight*
Ref	*ReferenceType* module of **References**.v
RefS	*RefSetType* module of **RefSets**.v
RefSet	*Reference FSet* of *RefS*
Edges	*AccessEdgeType* module of **AccessEdges**.v
AccessGraph	*AccessGraphType* module of **AccessGraphs**.v
AG	*FSet* of *AccessGraphType*
Seq	*SeqAccType* module of **SequentialAccess**.v
Cap	*CapabilityType* of **Capabilities**.v
CC	*CapabilityConv* of **Capabilities_Conv**.v
CapS	*CapSetType*
CapSet	*Capability FSet* of *CapS*
Ind	*IndexType* of **Indicies**.v
Obj	*ObjectType* of **Objects**.v
OC	*ObjectConv* of **Objects_Conv**.v
Sys	*SystemStateType* of **SystemState**.v
SC	*SystemStateConv* of **SystemState_Conv**.v
SemDefns	*SemanticsDefinitionsType* of **SemanticsDefinitions**.v
Sem	*SemanticsType* of **Semantics**.v
SemConv	*SemanticsConv* of **Semantics_Conv**.v
Exe	*ExecutionType* of **Execution**.v
Mut	*MutationType* of **Mutation**.v
Sub	*SubsystemType* of **Subsystem**.v

as above. All abstract module types passed as parameters and convenience modules share the same naming convention throughout the proof, which is summarized in table 4.1.

Chapter 5

System Embedding

This chapter discusses how capability-based systems are embedded in SDM. It begins by presenting how capabilities are modeled and then walks the system state structure, the snapshot of the security state of the system. It concludes with the operational semantics of the model including semantics for both the security state and potential information flow for each operation.

5.1 Names, Access Rights, and Capabilities

There are two forms of names in SDM: names for objects called *object references* and names for storage locations within an object called *indices.* Both names must have a total ordering which is both computable and based on equality. This ordering

Figure 5.1 Object references type and implementation.

References Module

```
Require Import OrderedType.
Require Import OrderedInclude.
Module Type ReferenceType := UsualOrderedTypeWithHints.
Module ReferenceTypeFacts := OrderedTypeFacts.
```

ReferencesImpl Module

```
Require Import References.
Require Import OrderedTypeEx.
Module NatReference >: ReferenceType := Nat_as_OT.
Module NatReferenceFacts := ReferenceTypeFacts NatReference.
```

requirement was chosen to facilitate faster proofs by easing the requirements of the rewrite tactic. These requirements do not obligate any implementation to use such a strong encoding directly; all names reside behind other abstractions.

The Coq standard library includes modules that directly capture the requirements for names. The *UsualOrderedType* module from the **OrderedTypeEx** standard library is almost perfect. An *OrderedType* is a module containing an equivalence, an ordering relation, and a proof of total ordering while a *UsualOrderedType* extends the *OrderedType* to require term equality for the equivalence relation. Unfortunately, it does not include the tactic hints from *OrderedType* module that assist proof automation. Therefore, names are built from *UsualOrderedTypeWithHints* module from the **OrderedInclude** module, which reintroduces these hints. The trivial implementation supplied by SDM uses natural numbers via the *Nat_as_OT* module, also included in the **OrderedTypeEx** library.

Figure 5.2 AccessRights module.

```
Inductive accessRight :=
| wk : accessRight
| rd : accessRight
| wr : accessRight
| tx : accessRight.

Module ProjectedAccessRight >: ProjectedToNat.

  Definition t := accessRight.

  Definition project_to_nat r:=
    match r with
      | wk ⇒ 0
      | rd ⇒ 1
      | wr ⇒ 2
      | tx ⇒ 3
    end.

  Theorem project_to_nat_unique: ∀ x y:t,
    (project_to_nat x) = (project_to_nat y) ↔ x = y.

End ProjectedAccessRight.

Module AccessRight := POT_to_OT ProjectedAccessRight.
```

The system contains four access rights: *rd*, *wr*, *wk*, and *tx*, each conferring the ability to perform various system operations. Section 5.3 discusses how these permissions are used in detail, but their intent is simple. The *wr* permission enables the ability to directly write and overwrite information in the target, or destroy the target altogether. The *rd* permission allows a subject to read information and capabilities from the target. The *wk* permission is a sub-type of the read permission that authorizes transitive read-only authority. Message passing between subjects is authorized by the *tx* permission.

The implementations for most enumerated types in SDM are created by the *POT_to_OT* functor found in the **ProjectedOrderedType** module. This functor produces an *OrderedType* from a *ProjectedToNat* module. Modules satisfying *ProjectedToNat* must contain some base type, an injection function from that type to the natural numbers, and a proof of injectivity. This allows different inductive terms to be quickly defined and used as *OrderedType*s.

In capability-based systems, a *capability* is an unforgable binding of a name and permissions. This is effectively a product of an object reference and a finite set of access rights. However, the capability abstraction hides such an implementation behind accessor functions and two proofs of equivalence. Capabilities have the constructor *mkCap* and accessor functions *target* and *rights* with the following properties: 1) injection: two capabilities are equivalent if and only if their accessors have an equivalent target and right and 2) equivalence: invoking accessors on the constructor produces

Figure 5.3 Capabilities.

```
Module Type CapabilityType (Ref:ReferenceType).

  Include Type OrderedType.OrderedType.

  Parameter target: t → Ref.t.
  Parameter rights: t → ARSet.t.
  Parameter mkCap: Ref.t → ARSet.t → t.
  Parameter weaken: t → t.

  Parameter target_eq: ∀ (c1 c2:t), eq c1 c2 → Ref.eq (target c1) (target c2).
  Parameter rights_eq: ∀ (c1 c2:t), eq c1 c2 → ARSet.eq (rights c1) (rights c2).

  Parameter target_rights_eq: ∀ (c1 c2:t),
    Ref.eq (target c1) (target c2) →
    ARSet.eq (rights c1) (rights c2) →
    eq c1 c2.

  Parameter mkCap_eq: ∀ tgt rgts c,
    Ref.eq tgt (target c) ∧ ARSet.eq rgts (rights c) ↔
    eq (mkCap tgt rgts) c.

  Parameter weaken_eq: ∀ c,
    eq (weaken c)
    (mkCap (target c)
      (if orb (true_bool_of_sumbool (ARSetProps.In_dec wk (rights c)))
        (true_bool_of_sumbool (ARSetProps.In_dec rd (rights c)))
        then (ARSet.singleton wk)
        else ARSet.empty)).

End CapabilityType.
```

equivalent inputs.

The capability type also requires a *weaken* function to produce a weakened form of a capability. Weakening a capability produces a capability with the same target, but eliminates all access rights except *wk*. Because the *rd* also entails the *wk* access right, a weakened capability produces *wk* from a *rd*-only capability. The *weaken* function is used by the operational semantics to cause *wk* to be a system-enforced, transitive, read-only access right.

5.2 Objects and System State

The *system state* represents a snapshot of the permission state in the system at a fixed instant in time. Structurally, the system state is a finite map from a reference to a quadruple of an object, object label, object type, and object schedule. All of these types, with the exception of objects, are enumerations defined by injection to the natural numbers using the *ProjectedToNat* type. An *ObjectLabel* indicates what phase of the object life-cycle the object is in; it may be *unborn*, *alive*, or *dead*. The distinction between *active* objects, such as processes or threads, and *passive* storage objects is captured by the *ObjectType*. The *ScheduleType* is presently a placeholder and will not be used in the remainder of the proof.

An *object* represents the permission state of a single object in the system and is a finite map from *indices* to *capabilities*. While SDM captures the potential for data

Figure 5.4 Objects.

```
Module Type ObjectType (Ref: ReferenceType) (Cap:CapabilityType Ref)
(Ind:IndexType).
  Declare Module MapS : Sfun Ind.
  Include (Sord_fun Ind MapS Cap).
  Parameter eq_dec : ∀ x y, { eq x y } + { ¬ eq x y }.
End ObjectType.
```

Figure 5.5 System state.

```
Module Type SystemStateType (Ref: ReferenceType) (Cap:CapabilityType Ref)
(Ind:IndexType) (Obj:ObjectType Ref Cap Ind).
  Declare Module MapS : Sfun Ref.
  Module P3 := PairOrderedType Obj ObjectLabel.
  Module P2 := PairOrderedType P3 ObjectType.
  Module P := PairOrderedType P2 ObjectSchedule.
    Include (Sord_fun Ref MapS P).
    Parameter eq_dec : ∀ x y, { eq x y } + { ¬ eq x y }.
End SystemStateType.
```

motion in the operational semantics, actual data are not represented. All objects may contain both capabilities and data. System implementations are free to implement a partition at object granularity. SDM makes no guarantee about whether capabilities are an opaque or visible data structure and conservatively approximates them as visible.[1]

The object and system state interfaces are a specialization of the *Ordered FMap* interface, *SOrd*. In addition to being an *FMap* with keys as an *OrderedType*, an Ordered *FMap* is also required to have an ordering on its value or data. This allows both of these types to be *OrderedType*s with decidable equivalences over them and

[1]See Section 2.1 for more information.

permitting them to be nested. To allow objects and system states to also be an *OrderedType*, they include the missing decidable equivalence theorem. The system state data type is implemented by composing *ProductType* to produce a quadruple. The trivial implementations use the **FMapList** to construct these types.

As mentioned in Section 4.3, updates to the Coq module system have caused subtyping issues in the *FMap* library. It is possible to work around this issue by using pure functors, rather than the type capturing modules. Unfortunately, while these types exist for the plain *FMap*, they do not exist for *SOrd*. SDM includes the **FMap2*** libraries, which are a modification of the *FMap* libraries, but contain a pure form of the *SOrd_fun* type and functors.

Many modules have convenience modules to assist in creating legible results and reusable proofs. The definitions within these modules will not be discussed until they appear elsewhere and will only be provided if their implementation is not trivial given their description.. There are four convenience modules, one for each underlying module: **Capabilities**, **Objects**, **SystemState**, and **Semantics**.

5.3 Operational Semantics

The operational semantics over the system state is defined as the execution of an operation sequence. Each operation is defined in three parts: an operation precondition, a system state update function, and an upper bound on information flow

Figure 5.6 Definition of all operations.

```
Inductive operation : Type :=
| read: Ref.t → Ind.t → operation
| write: Ref.t → Ind.t → operation
| fetch: Ref.t → Ind.t → Ind.t → Ind.t → operation
| store: Ref.t → Ind.t → Ind.t → Ind.t → operation
| revoke: Ref.t → Ind.t → Ind.t → operation
| send: Ref.t → Ind.t → list (Ind.t × Ind.t) → option Ind.t → operation
| allocate: Ref.t → Ref.t → Ind.t → list (Ind.t × Ind.t) → operation
| destroy: Ref.t → Ind.t → operation.
```

between objects. The semantic model defines the preconditions for each operation and how the system state is altered upon success. Additionally, each operation contains an upper bound on information flow, for both success and failure.

There are eight operations that form the operational semantics of SDM. The *read* and *write* operations model only data flow and do not alter the system state in any way, while *fetch* and *store* operations model capability reads and writes from various objects. New objects are allocated using the *allocate* operation, and objects are destroyed using the *destroy* operation. Executing a *revoke* command will erase an entry in an object map. All message-passing protection mechanisms for capability systems are encoded using the *send* operation.

All operations have approximately the same structure. The first parameter of each operation is always the object reference of the invoking subject. The second parameter, with the exception of allocate, is the index of the capability being invoked. In the case of allocate, no such capability exists, so the parameter specifies the unborn object being allocated. The indices being accessed and updated are either passed directly or as a list of index pairs when multiple capabilities may be transferred. The index option permits a send operation to fabricate a reply.

These operations should not be thought of as a system call performed by the invoking subject, or specified in parts by all parties. They should be considered a specification of which actions the system may take when updating the system state or shuffling data. A well constructed system should not ever perform an invalid operation. The only reason invalid operations are defined is to permit operations and operation sequences to be pure data structures; the semantics will skip over nonsense operations.

Each of these operations, with the exception of *allocate*, requires a capability containing an appropriate permission. As operations that inspect an object, the *read* and *fetch* operations require *rd* or *wk* access. The *wr* access right is required by the fetch, store, revoke, and destroy operations as they modify object state externally. The *send* operation is handled with its own permission as it potentially produces a bi-directional relationship via an optional reply capability. Object allocation through *allocate* is considered universally available and does not require a capability. [2]

The definitions for SDM operational semantics are distributed across two modules. The library **SemanticsDefinitions** contains all of the operation prerequisites and proofs of their decidability. These proofs are later used in decision procedures in the **Semantics** library when defining operation success and failure. This separation

[2]Most systems configurations presume processes have the ability to allocate and manage storage. See section 2.6

Figure 5.7 Operation preconditions. From **SemanticsDefinitions**.

```
Definition preReqCommon a s := SC.is_alive a s ∧ SC.is_active a s.
Definition preReq a t s :Prop := preReqCommon a s ∧ target_is_alive t a s.
Definition read_preReq a t s := preReq a t s ∧
  (option_hasRight (SC.getCap t a s) rd ∨ option_hasRight (SC.getCap t a s) wk).
Definition write_preReq a t s := preReq a t s ∧
  option_hasRight (SC.getCap t a s) wr.
Definition fetch_preReq a t s := preReq a t s ∧
  (option_hasRight (SC.getCap t a s) rd ∨ option_hasRight (SC.getCap t a s) wk).
Definition store_preReq a t s := preReq a t s ∧
  option_hasRight (SC.getCap t a s) wr.
Definition revoke_preReq a t s := preReq a t s ∧
  option_hasRight (SC.getCap t a s) wr.
Definition send_preReq a t s := preReq a t s ∧
  option_hasRight (SC.getCap t a s) tx.
Definition allocate_preReq a n s := preReqCommon a s ∧ SC.is_unborn n s.
Definition destroy_preReq a t s := preReq a t s ∧
  option_hasRight (SC.getCap t a s) wr.
```

allows these decision procedures to be included as Boolean decisions in the model signature and implementation.

There are a few omitted definitions from Figure 5.7 . The (*target_is_alive t a s*) function checks that the target of the capability at index *t* in object *a* has object label alive in system state *s*. Likewise, (*target_is_active t a* s) checks that the object type is active in the same manner. The system state convenience library offers the *SC*.(*getCap t a s*) function to return an *Option Cap.t* to find a capability at index *t* in object *a*. *option_hasRight* simply shorthand for mapping *CC.hasRight* over an *Option Cap.t* and returning *False* in the *None* case.

The **SemanticsDefinitions** library is dependent on the *SystemStateType*, and all prerequisites. It begins by defining a common operation prerequisite (*preReq a t* *s*), which tests that the agent subject *a* is alive and active in system state *s*, and that the target of the capability at index *t* is also *alive* in *s*. All operations except allocate share *preReq* as a condition on their success. Additionally, the target capability must have the appropriate access right as previously defined. The exception to this is the (*create_preReq a n s*) which does not test a target capability, but checks that the object *n* is unborn. The remainder of the library contains the theorems for the decidability and equivalence of each of these predicates and their support functions.

Valid state transitions and information flow are defined in the **Semantics** library. Each state transition for operations where preconditions are satisfied is given in Figure 5.8. The theorems describing the trivial case of no state transition when preconditions are not satisfied have been omitted. The *read* and *write* operations do not cause any state transition in either case and are defined as a single theorem. The *do_op* function concretely unifies all of these specific cases together and is specified in Figure 5.9.

For brevity, certain definitions have been omitted from Figure 5.8. *optionMap1* and *optionMap2* are specialized *Option* structures that combine both mapping and reducing functionality. Each is applied to a collection of functions of arity zero through one or two, respectively. Once applied to either one or two *Option* types, they apply the number of *Some* results to the function of matching arity. The system state convenience library also includes the *SC.copyCap* and *SC.weakCopyCap* functions to perform a standard or weakening capability transfer. The *SC.copyCapList* func-

Figure 5.8 State transitions with preconditions satisfied. From **Semantics**.

```
Theorem read_spec: ∀ a t s, Sys.eq (do_read a t s) s.
Theorem write_spec: ∀ a t s, Sys.eq (do_write a t s) s.
Theorem fetch_read: ∀ a t c i s, SemDefns.fetch_preReq a t s →
  SemDefns.option_hasRight (SC.getCap t a s) rd →
  Sys.eq (do_fetch a t c i s)
        (option_map1 (fun tgt ⇒ SC.copyCap c tgt i a s) s
            (SemDefns.option_target (SC.getCap t a s))).
Theorem fetch_weak: ∀ a t c i s, SemDefns.fetch_preReq a t s →
  ¬ SemDefns.option_hasRight (SC.getCap t a s) rd →
  SemDefns.option_hasRight (SC.getCap t a s) wk →
  Sys.eq (do_fetch a t c i s)
        (option_map1 (fun tgt ⇒ SC.weakCopyCap c tgt i a s) s
            (SemDefns.option_target (SC.getCap t a s))).
Theorem store_valid: ∀ a t c i s, SemDefns.store_preReq a t s →
  Sys.eq (do_store a t c i s)
        (option_map1 (fun tgt ⇒ SC.copyCap c a i tgt s) s
            (SemDefns.option_target (SC.getCap t a s))).
Theorem revoke_valid : ∀ a t c s, SemDefns.revoke_preReq a t s →
  Sys.eq (do_revoke a t c s)
        (option_map1 (fun tgt ⇒ SC.rmCap c tgt s) s
            (SemDefns.option_target (SC.getCap t a s))).
Theorem send_valid : ∀ a t cil op_i s, SemDefns.send_preReq a t s →
  Sys.eq (do_send a t cil op_i s)
        (option_map1
          (fun tgt ⇒ SC.copyCapList a tgt cil
            (option_map1
              (fun i ⇒ SC.addCap i (Cap.mkCap a (ARSet.singleton tx)) tgt s)
              s op_i))
          s (SemDefns.option_target (SC.getCap t a s))).
Theorem allocate_valid: ∀ a n i cil s, SemDefns.allocate_preReq a n s →
  Sys.eq (do_allocate a n i cil s)
        (SC.addCap i (Cap.mkCap n all_rights) a
          (SC.copyCapList a n cil
            (SC.set_alive n (SC.updateObj n (Obj.MapS.empty _)
              (SC.rmCapsByTarget n s))))).
Theorem destroy_valid: ∀ a t s, SemDefns.destroy_preReq a t s →
  Sys.eq (do_destroy a t s)
        (option_map1 (fun tgt ⇒ SC.set_dead tgt s)
          s (SemDefns.option_target (SC.getCap t a s))).
```

Figure 5.9 Specification for *do_op*.

```
Inductive do_op_spec : operation → (Sys.t → Sys.t) → Prop :=
  | do_op_spec_read: ∀ a t,
     do_op_spec (read a t) (do_read a t)
  | do_op_spec_write: ∀ a t,
     do_op_spec (write a t) (do_write a t)
  | do_op_spec_fetch: ∀ a t c i,
     do_op_spec (fetch a t c i) (do_fetch a t c i)
  | do_op_spec_store: ∀ a t c i,
     do_op_spec (store a t c i) (do_store a t c i)
  | do_op_spec_revoke: ∀ a t c,
     do_op_spec (revoke a t c) (do_revoke a t c)
  | do_op_spec_send: ∀ a t cil op_i,
     do_op_spec (send a t cil op_i) (do_send a t cil op_i)
  | do_op_spec_allocate: ∀ a t i cil,
     do_op_spec (allocate a t i cil) (do_allocate a t i cil)
  | do_op_spec_destroy: ∀ a t,
     do_op_spec (destroy a t) (do_destroy a t).
Theorem do_op_spec_do_op: ∀ op,    do_op_spec op (do_op op).
```

tion performs many such copies using an index pair list as a map. All capabilities with a specific target are removed from the system using *SC.rmCapsByTarget*. *SC.updateObj* updates only the object at a reference and *SC.setAlive* or *setDead* updated the object label appropriately.

The potential for data motion is captured by the *readFrom* and *wroteTo* definitions. During an operation, data may flow from any object in the *readFrom* set to any object in the *wroteTo* set. The specification of these functions is given in Figures 5.11 and 5.12. However, Figure 3.6 summarizes the conditions simply, and it is reproduced in Figure 5.10. Self-mutation is not modeled by any operations and SDM presumes all active objects may alter their own state.

Performing an operation sequence is modeled by the *execute* function applied to an

Figure 5.10 Reproduction of information flow from Figure 3.6.

$$\begin{aligned}
readFrom(read(a,t),S) &\equiv \{a, objTarget(a,t,S)\}\\
readFrom(write(a,t),S) &\equiv \{a\}\\
readFrom(fetch(a,t,c,i),S) &\equiv \{a, objTarget(a,t,S)\}\\
readFrom(store(a,t,c,i),S) &\equiv \{a\}\\
readFrom(revoke(a,t,c),S) &\equiv \{a\}\\
readFrom(destroy(a,t),S) &\equiv \{a\}\\
readFrom(allocate(a,n,m,typ),S) &\equiv \{a\}\\
readFrom(send(a,t,m,x),S) &\equiv \{a\}
\end{aligned}$$

$$\begin{aligned}
wroteTo(read(a,t),S) &\equiv \{a\}\\
wroteTo(write(a,t),S) &\equiv \{objTarget(a,t,S)\}\\
wroteTo(fetch(a,t,c,i),S) &\equiv \{a\}\\
wroteTo(store(a,t,c,i),S) &\equiv \{objTarget(a,t,S)\}\\
wroteTo(revoke(a,t,c),S) &\equiv \{objTarget(a,t,S)\}\\
wroteTo(destroy(a,t),S) &\equiv \{a\}\\
wroteTo(allocate(a,n,m,typ),S) &\equiv \{a,n\}\\
wroteTo(send(a,t,m,x),S) &\equiv \{objTarget(a,t,S)\}
\end{aligned}$$

when preconditions are satisfied, otherwise

$$\begin{aligned}
readFrom(op,S) &\equiv \{\}\\
wroteTo(op,S) &\equiv \{\}
\end{aligned}$$

Figure 5.11 Definition of *readFrom*.

```
Theorem read_from_spec : ∀ s op ob_list,
  read_from_def s op ob_list ↔
  RefSet.Equal (read_from s op) ob_list.
Inductive read_from_def s: operation → RefSet.t → Prop :=
| read_from_read_valid : ∀ a t x, SemDefns.read_preReq a t s →
  RefSet.Equal (add_option_target s a t (RefSet.singleton a)) x →
  read_from_def s (read a t) x
| read_from_read_invalid : ∀ a t x, ¬ SemDefns.read_preReq a t s →
  RefSet.Empty x → read_from_def s (read a t) x
| read_from_write_valid: ∀ a t x, SemDefns.write_preReq a t s →
  RefSet.Equal (RefSet.singleton a) x → read_from_def s (write a t) x
| read_from_write_invalid: ∀ a t x, ¬ SemDefns.write_preReq a t s →
  RefSet.Empty x → read_from_def s (write a t) x
| read_from_fetch_valid : ∀ a t c i x SemDefns.fetch_preReq a t s →
  RefSet.Equal (add_option_target s a t (RefSet.singleton a)) x →
  read_from_def s (fetch a t c i) x
| read_from_fetch_invalid : ∀ a t c i x, ¬ SemDefns.fetch_preReq a t s →
  RefSet.Empty x → read_from_def s (fetch a t c i) x
| read_from_store_valid: ∀ a t c i x,SemDefns.store_preReq a t s →
  RefSet.Equal (RefSet.singleton a) x → read_from_def s (store a t c i) x
| read_from_store_invalid: ∀ a t c i x, ¬ SemDefns.store_preReq a t s →
  RefSet.Empty x → read_from_def s (store a t c i) x
| read_from_revoke_valid: ∀ a t c x, SemDefns.revoke_preReq a t s →
  RefSet.Equal (RefSet.singleton a) x → read_from_def s (revoke a t c) x
| read_from_revoke_invalid: ∀ a t c x, ¬ SemDefns.revoke_preReq a t s →
  RefSet.Empty x → read_from_def s (revoke a t c) x
| read_from_send_valid: ∀ a t cil op_i x, SemDefns.send_preReq a t s →
  RefSet.Equal (RefSet.singleton a) x → read_from_def s (send a t cil op_i) x
| read_from_send_invalid: ∀ a t cil op_i x, ¬ SemDefns.send_preReq a t s →
  RefSet.Empty x → read_from_def s (send a t cil op_i) x
| read_from_allocate_valid: ∀ a n i cil x, SemDefns.allocate_preReq a n s →
  RefSet.Equal (RefSet.singleton a) x → read_from_def s (allocate a n i cil) x
| read_from_allocate_invalid: ∀ a n i cil x, ¬ SemDefns.allocate_preReq a n s →
  RefSet.Empty x → read_from_def s (allocate a n i cil) x
| read_from_destroy_valid: ∀ a t x, SemDefns.destroy_preReq a t s →
  RefSet.Equal (RefSet.singleton a) x → read_from_def s (destroy a t) x
| read_from_destroy_invalid: ∀ a t x, ¬ SemDefns.destroy_preReq a t s →
  RefSet.Empty x → read_from_def s (destroy a t) x.
.
```

Figure 5.12 Definition of *wroteTo*.

```
Theorem wrote_to_spec : ∀ s op ob_list,
  wrote_to_def s op ob_list ↔
  RefSet.Equal (wrote_to s op) ob_list.
Inductive wrote_to_def s: operation → RefSet.t → Prop :=
| wrote_to_read_valid : ∀ a t x, SemDefns.read_preReq a t s →
  RefSet.Equal (RefSet.singleton a) x → wrote_to_def s (read a t) x
| wrote_to_read_invalid : ∀ a t x, ¬ SemDefns.read_preReq a t s →
  RefSet.Empty x → wrote_to_def s (read a t) x
| wrote_to_write_valid: ∀ a t x, SemDefns.write_preReq a t s →
  RefSet.Equal (add_option_target s a t RefSet.empty) x →
  wrote_to_def s (write a t) x
| wrote_to_write_invalid: ∀ a t x, ¬ SemDefns.write_preReq a t s →
  RefSet.Empty x → wrote_to_def s (write a t) x
| wrote_to_fetch_valid : ∀ a t c i x, SemDefns.fetch_preReq a t s →
  RefSet.Equal (RefSet.singleton a) x → wrote_to_def s (fetch a t c i) x
| wrote_to_fetch_invalid : ∀ a t c i x, ¬ SemDefns.fetch_preReq a t s →
  RefSet.Empty x → wrote_to_def s (fetch a t c i) x
| wrote_to_store_valid: ∀ a t c i x, SemDefns.store_preReq a t s →
  RefSet.Equal (add_option_target s a t RefSet.empty) x →
  wrote_to_def s (store a t c i) x
| wrote_to_store_invalid: ∀ a t c i x, ¬ SemDefns.store_preReq a t s →
  RefSet.Empty x → wrote_to_def s (store a t c i) x
| wrote_to_revoke_valid: ∀ a t c x, SemDefns.revoke_preReq a t s →
  RefSet.Equal (add_option_target s a t RefSet.empty) x →
  wrote_to_def s (revoke a t c) x
| wrote_to_revoke_invalid: ∀ a t c x, ¬ SemDefns.revoke_preReq a t s →
  RefSet.Empty x → wrote_to_def s (revoke a t c) x
| wrote_to_send_valid: ∀ a t cil op_i x, SemDefns.send_preReq a t s →
  RefSet.Equal (add_option_target s a t RefSet.empty) x →
  wrote_to_def s (send a t cil op_i) x
| wrote_to_send_invalid: ∀ a t cil op_i x, ¬ SemDefns.send_preReq a t s →
  RefSet.Empty x → wrote_to_def s (send a t cil op_i) x
| wrote_to_allocate_valid: ∀ a n i cil x, SemDefns.allocate_preReq a n s →
  RefSet.Equal (RefSet.add n (RefSet.singleton a)) x →
  wrote_to_def s (allocate a n i cil) x
| wrote_to_allocate_invalid: ∀ a n i cil x, ¬ SemDefns.allocate_preReq a n s →
  RefSet.Empty x → wrote_to_def s (allocate a n i cil) x
| wrote_to_destroy_valid: ∀ a t x, SemDefns.destroy_preReq a t s →
  RefSet.Equal (RefSet.singleton a) x → wrote_to_def s (destroy a t) x
| wrote_to_destroy_invalid: ∀ a t x, ¬ SemDefns.destroy_preReq a t s →
  RefSet.Empty x → wrote_to_def s (destroy a t) x.
```

```
Theorem execute_spec : ∀ s s' op_list,
  execute_def s op_list s' ↔ Sys.eq (execute s op_list) s'.
Inductive execute_def s: list Sem.operation → Sys.t → Prop :=
| execute_nil: ∀ s', Sys.eq s s' → execute_def s nil s'
| execute_cons : ∀ op tail s',
  execute_def s tail s' →
  ∀ s'', Sys.eq (Sem.do_op op s') s'' →
  execute_def s (op :: tail) s''.
```

operation list, provided by the **Execution** module. Because the semantics will ignore any invalid operation, it is defined as simply performing the operations sequentially. The only curious property of the *execute* function is that it performs operations on a list in reverse. This is done to align list induction with execution induction. An empty list performs no operations and, given a system state that is the result of executing an operation list, performing another operation is the result of the *cons* constructor and not an *append* function.

Chapter 6

Access Graphs and Potential Access

This chapter presents access graphs as the abstract structure for describing authority about multiple system states There are many different types of access graphs used by SDM. The direct access relation translates a system state to an access graph with the same authority. The complete access graph for a set of objects defines an access graph where each object has total authority to every other object. *Potential access* is the judgment defining the worst-case access that a system may ever come to have.

Potential access is used extensively throughout SDM. It is built from the *transfer* relation: the smallest possible access right transfer in an access graph. It defines a potential transfer as a sequence of *transfer*s. It then presents the definition of a maximal access graph along with the proof that for each initial access graph there is only one access graph that is both maximal and reachable by potential transfer, called the potential access graph. The potential access of a system state is the potential access graph of that system's direct access graph. Finally, this chapter demonstrates this by verifying that potential transfer forms a complete partial order when rooted at some initial access graph and computing potential access as a function in `Set`.

6.1 Access Graph Structure

Access graphs are the structure used by SDM to perform nearly all permission and information flow reasoning. Access graphs express system states abstractly, representing many system states simultaneously. They condense permission information encoded in system state to precisely the permissions held between objects, rather than their capabilities. An access graph is a finite set of *access edges*, each a triple in $ReferenceType * ReferenceType * AccessRight$. The edge $src \xrightarrow{ar} tgt$ indicates that *src* holds an *ar* access right to *tgt*. As a collection of edges, the access graph makes no assertions about the nature of objects within it. If any additional constraints are required, they must be carried alongside the access graph.

As a Coq module, the access graph is constructed as an **FSetList** with an *AccessEdge* element type. To satisfy the *UsualOrderedType* interface, the *AccessEdge* module is constructed as two *ProductType*s. The entire functor is parameterized over

Figure 6.1 Access graph interface definition.

```
Module Type AccessEdgeType (R: ReferenceType) .
  Module AccessArrow := !PairOrderedType R AccessRight.
  Module AccessEdge := !PairOrderedType R AccessArrow.
  Definition mkEdge (src tgt:R.t) (rgt:accessRight): AccessEdge.t :=
    (pair src (pair tgt rgt)).
  Definition source (edge : AccessEdge.t) := fst edge.
  Definition target (edge : AccessEdge.t) := fst (snd edge).
  Definition right (edge : AccessEdge.t) := snd (snd edge).
  ...
End AccessEdgeType.
Module Type AccessGraphType (R: ReferenceType) (Edges: AccessEdgeType R) .
  Module Edge := Edges.AccessEdge.
  Module AG := !FSetList.Make Edge.
  ...
End AccessGraphType.
```

the *ReferenceType* module interface. The concrete implementation uses the **FSetList** functor as it provides an equivalence relation based on identical data structures.

6.2 Direct Access

The *Direct access graph* of a system state is the access graph containing edges for every capability held by alive objects whose target is also alive. This forms a reduction of the system state, as many capabilities may justify the existence of a single edge. Capabilities held by dead or unborn objects are ignored as they may not be invoked and cannot be transferred. Capabilities targeting unborn objects are elided as they are removed before allocation. Access rights that appear within

Figure 6.2 Inductive definition of direct access graph.

```
Definition dirAcc_spec s ag : Prop := ∀ edge, AG.In edge ag ↔
  ∃ s', Sys.eq s s' ∧
  ∃ src_ref, ∃ src, ∃ lbl, ∃ srcType, ∃ srcSched,
    Sys.MapS.MapsTo src_ref (src, lbl, srcType, srcSched) s' ∧
  ∃ src', ∃ lbl', ∃ srcType', ∃ srcSched',
    Sys.P.eq (src, lbl, srcType, srcSched) (src', lbl', srcType', srcSched') ∧
    ObjectLabel.eq ObjectLabels.alive lbl' ∧
  ∃ ind, ∃ cap, Obj.MapS.MapsTo ind cap src' ∧
  ∃ cap_obj, ∃ cap_lbl, ∃ cap_type, ∃ cap_sched,
    Sys.MapS.MapsTo (Cap.target cap) (cap_obj, cap_lbl, cap_type, cap_sched) s' ∧
  ∃ cap_obj', ∃ cap_lbl', ∃ cap_type', ∃ cap_sched',
    Sys.P.eq (cap_obj, cap_lbl, cap_type, cap_sched)
             (cap_obj', cap_lbl', cap_type', cap_sched') ∧
    ObjectLabel.eq ObjectLabels.alive cap_lbl' ∧
  ∃ rgt, ARSet.In rgt (Cap.rights cap) ∧
    Edge.eq (Edges.mkEdge src_ref (Cap.target cap) rgt) edge.
```

Figure 6.3 Simplified, but not complete definition of direct access graph.

```
  Theorem dirAcc_simpl : ∀ s ag, dirAcc_spec s ag →
    ∀ src, SC.is_alive src s →
    ∀ t cap, SC.getCap t src s = Some cap →
    ∀ tgt, Ref.eq tgt (Cap.target cap) → SC.is_alive tgt s →
    ∀ rgt, CC.hasRight cap rgt →
      AG.In (Edges.mkEdge src tgt rgt) ag.
```

Figure 6.4 *dirAcc* function and lemmas.

```
Definition dirAcc_inner src_ref src_obj s ag :=
  Obj.MapS.fold (fun index cap acc_ag ⇒
    ag_add_cap_valid src_ref cap s (fun c⇒c) ag_add_cap_valid_std acc_ag)
  src_obj ag.
Definition dirAcc_outer s ag :=
  Sys.MapS.fold (fun src_ref src_tuple acc_ag ⇒
    dirAcc_inner src_ref (SC.tupleGetObj src_tuple) s acc_ag)
  s ag.
Definition dirAcc s := dirAcc_outer s AG.empty.
Theorem dirAcc_spec_dirAcc: ∀ s, dirAcc_spec s (dirAcc s).
```

multiple capabilities naming the same object are represented by a single edge, further abstracting the system state. Therefore, any analysis of access graphs necessarily considers how access rights within a capability could operate were they independent.

The libraries **DirectAccess** and **DirectAccessImpl** contain the interface and implementation of direct access, respectively. The *dirAcc_spec* predicate definition shown in Figure 6.2 defines direct access using set comprehension. It captures the equivalences needed to reason about different system states and access graphs. As such, it is a bit cumbersome, and the verification usually relies upon *dirAcc_simpl* for most requirements.

The remainder of the **DirectAccessImpl** module defines the *dirAcc* function in `Set` and demonstrates that it is equivalent to the *dirAcc_spec* proposition. Various map fold functions are used to construct the appropriate access graph from an *AG.empty* accumulator. The definition of *dirAcc* in `Set` and proof of equivalence together produce both decidability and existence proofs for *dirAcc_spec*.

SDM defines a number of reusable functions and lemmas as part of the *dirAcc*

Figure 6.5 Definition of *ag_add_cap* and supplemental lemmas.

```
Definition ag_add_cap src cap ag:=
  ARSet.fold (fun rgt acc ⇒ AG.add (Edges.mkEdge src (Cap.target cap) rgt) acc)
             (Cap.rights cap) ag.
Theorem ag_add_cap_equiv: ∀ src src' cap cap' ag ag',
  Ref.eq src src' → Cap.eq cap cap' → AG.eq ag ag' →
  AG.eq (ag_add_cap src cap ag) (ag_add_cap src' cap' ag').
Theorem ag_add_cap_nondecr : ∀ src cap ag ag',
  AG.Subset ag ag' → AG.Subset ag (ag_add_cap src cap ag').
Theorem ag_add_cap_add_commute: ∀ src cap,
  Seq.ag_add_commute (ag_add_cap src cap).
Definition ag_add_commute F := ∀ ag ag' x,
  AGProps.Add x ag ag' → AGProps.Add x (F ag) (F ag').
```

definition. Each of the functions named *ag_add_cap** adds capabilities to an access graph using the base *ag_add_cap* function. The *ag_add_cap* function adds all edges represented by a single capability into an access graph. The *ag_add_cap* function, and similar functions based thereon, all preserve equivalence, are non-decreasing, and are commutative with set addition. Figure 6.5 contains the definition of *ag_add_cap* and supplemental lemmas. The other definitions will be presented as needed, but theorem definitions will not accompany them. For the full implementation, please review the proof.

6.3 Potential Access

The *potential access graph* is the access graph representing the greatest potential permission state of a initial access graph. The definition of a potential access graph is the closure of the *transfer* relation, a micro-operation of permission transfer. The

Figure 6.6 Definition of *transfer*.

```
Inductive transfer (a b : AG.t) : Prop :=
| transfer_self_src : ∀ (rgt rgt': accessRight) (src tgt:Ref.t),
  AG.In (Edges.mkEdge src tgt rgt) a →
  AGProps.Add (Edges.mkEdge src src rgt') a b →
  transfer a b
| transfer_self_tgt : ∀ (rgt rgt': accessRight) (src tgt:Ref.t),
  AG.In (Edges.mkEdge src tgt rgt) a →
  AGProps.Add (Edges.mkEdge tgt tgt rgt') a b →
  transfer a b
| transfer_read : ∀ (rgt: accessRight) (src tgt tgt':Ref.t),
  AG.In (Edges.mkEdge src tgt rd) a →
  AG.In (Edges.mkEdge tgt tgt' rgt) a →
  AGProps.Add (Edges.mkEdge src tgt' rgt) a b →
  transfer a b
| transfer_write : ∀ (rgt: accessRight) (src tgt tgt':Ref.t),
  AG.In (Edges.mkEdge src tgt wr) a →
  AG.In (Edges.mkEdge src tgt' rgt) a →
  AGProps.Add (Edges.mkEdge tgt tgt' rgt) a b →
  transfer a b
| transfer_send : ∀ (rgt: accessRight) (src tgt tgt':Ref.t),
  AG.In (Edges.mkEdge src tgt tx) a →
  AG.In (Edges.mkEdge src tgt' rgt) a →
  AGProps.Add (Edges.mkEdge tgt tgt' rgt) a b →
  transfer a b
| transfer_send_reply : ∀ (src tgt:Ref.t),
  AG.In (Edges.mkEdge src tgt tx) a →
  AGProps.Add (Edges.mkEdge tgt src tx) a b →
  transfer a b
| transfer_weak : ∀ (rgt: accessRight) (src tgt tgt':Ref.t),
  AG.In (Edges.mkEdge src tgt wk) a →
  AG.In (Edges.mkEdge tgt tgt' rgt) a →
  (eq rgt wk ∨ eq rgt rd) →
  AGProps.Add (Edges.mkEdge src tgt' wk) a b →
  transfer a b.
```

Figure 6.7 *transfer*is monotonic and has a least upper bound.

`Theorem` $transfer\_monotonic$: $\forall\ (i\ a\ b\ c : AG.t)\ (edge\text{: } Edge.t)$,
$AG.Subset\ i\ b \to AGProps.Add\ edge\ i\ a \to transfer\ i\ a \to$
$AGProps.Add\ edge\ b\ c \to transfer\ b\ c.$
`Theorem` $transfer\_lub$: $\forall\ (i\ a\ b{:}AG.t)$,
$transfer\ i\ a \to transfer\ i\ b \to \exists\ c{:}AG.t,\ transfer\ a\ c \land transfer\ b\ c.$

constructors for a transfer describe the seven methods by which new edges may appear in an access graph.

Each *transfer* constructor relates two access graphs by the inclusion of a single edge justified by an access right. The *transfer_read* constructor describes how an edges with the *rd* permission may add a new edge and is similar to the *fetch* operation. A *wr* permission authorizes any permission to be transferred in the other direction as performed by the *transfer_write* constructor. The constructor for the *wk* case, *transfer_weak*, handles how *wk* permissions are transferred. *transfer_send_reply* and *transfer_send* cover the *tx* permission authorizing a reply or authorizing a transfer through inter-process communication. There are also two constructors admitting self-targeting edges. To keep the number of edges finite, some other edge in the access graph is required to identify an object before adding its self-targeting edges. The cases *transfer_self_src* and *transfer_self_tgt* admit self-targeting edges for objects named by the source or target of an existing edge, respectively.

A transfer forms a restriction of the subset partial order which continues to be a partial order. As transfer always relates two access graphs by a single access edge, it must be monotonic. Because a transfer performs judgments based solely upon the

Figure 6.8 Definition of *potTransfer*.

```
Inductive potTransfer (a c:AG.t) : Prop :=
| potTransfer_base : AG.Equal a c → potTransfer a c
| potTransfer_trans : ∀ (b:AG.t), potTransfer a b → transfer b c → potTransfer a c.
```

Figure 6.9 Theorems about *potTransfer*.

```
Theorem potTransfer_lub:
  ∀ (i a b: AG.t), potTransfer i a → potTransfer i b →
    ∃ c : AG.t, potTransfer b c ∧ potTransfer a c.
```

existence of edges and not their absence, any edge added by a transfer will continue to be valid regardless of what new edges exist, including other transfers. Therefore, there is always a least upper bound for any two transfers on the same initial graph. The definition of *transfer_lub* captures the specific case for transfer.

A potential transfer is any sequence of transfers, even empty ones, and is defined by *potTransfer*. The *transfer* least upper bound can be extended to *potTransfer* such that any two potential transfers rooted sharing the same base have a least upper bound. Potential transfer forms a partial order over all possible transfers starting with a base access graph. Potential transfer is reflexive by inspection and transitive by simple induction. It must also be anti-symmetric since, as a sequence of transfers, it is non-decreasing. This least upper bound on potential transfer is used heavily in this verification as it permits transfers sharing an initial access graph to be reordered.

The supremum of potential transfer is *potential access* as it represents the most permissive state after a sequence of transfers. It is defined as the access graph that is both maximal and reachable by potential transfer. The usual definition of *maximal* applies: an access graph is maximal precisely when all potential transfers are to

Figure 6.10 Both definitions of maximal and *potAcc*.

```
Definition maxTransfer (i:AG.t) : Prop :=
  ∀ a:AG.t, transfer i a → AG.Equal i a.
Definition maxPotTransfer (i:AG.t) : Prop :=
  ∀ a:AG.t, potTransfer i a → AG.Equal i a.
Theorem maxTransfer_maxPotTransfer: ∀ (i:AG.t), maxTransfer i ↔ maxPotTrans-
fer i.
Definition potAcc (i max:AG.t) : Prop :=
  potTransfer i max ∧ maxPotTransfer max.
```

Figure 6.11 Definition for *ag_objs_spec* and *complete_ag_spec*.

```
Definition ag_objs_spec (i:AG.t) (objs: RefSet.t) :=
  ∀ x, RefSet.In x objs ↔
    ∃ obj, ∃ rgt, AG.In (Edges.mkEdge x obj rgt) i ∨
                  AG.In (Edges.mkEdge obj x rgt) i.
Definition complete_ag_spec refs full := ∀ edge,
  AG.In edge full ↔
    RefSet.In (Edges.source edge) refs ∧ RefSet.In (Edges.target edge) refs.
```

equivalent graphs. To reduce case analysis, we often use the equivalent definition of maximal claiming that all transfers are to equivalent graphs. Because potential transfer always has a least upper bound, every maximal access graph is also a potential access graph.

6.4 Computing Potential Access

The remainder of **SequentialAccess** contains proofs that *transfer* is decidable and *potAcc* is computable in its first input. To accomplish this, both properties are computed as functions in `Set`. Verifying the *transfer* judgment is a Boolean is substantial case analysis, but is obvious by inspection. The definition of potential

Figure 6.12 Theorem demonstrating constancy of Access Graph Objects through transfer.

```
Definition AG_all_objs (i:AG.t) (objs: RefSet.t) :=
  ∀ (src tgt:Ref.t) (rgt:accessRight),
     AG.In (Edges.mkEdge src tgt rgt) i →
     RefSet.In src objs ∧ RefSet.In tgt objs.
Theorem ag_objs_spec_AG_all_objs: ∀ i n, ag_objs_spec i n → AG_all_objs i n.
Theorem ag_all_objs_transfer: ∀ A N B,
  AG_all_objs A N → transfer A B → AG_all_objs B N.
```

Figure 6.13 *transfer* as a Boolean decision.

```
Theorem transfer_dec : ∀ A B:AG.t, {transfer A B} + {¬ transfer A B}.
```

transfer cannot be used to compute the potential access graph as it is far too general and does not guarantee progress.

The function *potAcc_fun* computes potential access by finding a single sequence of transfers that always adds a novel edge at each step. For novelty to produce a sound measure, there must be a theoretical limit to the number of new edges. By inspection, transfer does not alter which object references are in an access graph, it only adds edges between existing object references. The *ag_objs_spec* judgment extracts the object references from an access graph and this value remains unchanged by the transfer and potential transfer relations. The complete access graph for these object references, the access graph containing an edge with every access right and pair of object references available, is used as this upper limit. The measure conjecture *dist_from_complete* computes the cardinality of the set difference of the complete access graph and the current access graph.

The method used to potential access is not efficient, but demonstrably yields a

Figure 6.14 Definition of *potAcc_fun* and proof satisfying *potAcc*.

```
Function potAcc_fun (i:AG.t) {measure dist_from_complete i}: AG.t :=
  match findTransferEdge i with
    | None => i
    | Some edge => potAcc_fun (AG.add edge i)   end.                    Theorem
potAcc_potAcc_fun: ∀ i, potAcc i (potAcc_fun i).
```

correct result by exhaustion. *potAcc_fun* iterates over the complete access graph until it finds an edge satisfying the *transfer* relation when added to the initial access graph. If it finds such an edge, it adds the edge to the initial access graph and recurses. If no such edge exists in the complete access graph the current access graph must be maximal. Consequently it must also be the potential access graph.

The majority of SDM is concerned with the relationships surrounding potential access. *Potential access* is central to both the safety property and statically bounding data motion. Although there are other access graphs used by SDM, they will be defined in Chapter 9 when used.

Chapter 7

Safety

This chapter presents a proof of the safety property for capability-based systems using SDM. The safety property asks whether, from a given initial configuration, an object will come to hold an access right to some other object in the future. Because operations in the model are only authorized by the presence of access rights, the safety problem can be approximated using an upper bound on access graphs. The proof begins by defining the concept of functions which "conservatively approximate" the direct access graphs of system states between operations. It also provides the concrete approximating function *dirAcc_op*. The second step of the proof follows the same form as the first, defining how functions "conservatively approximate" potential access graphs between direct access functions and providing the concrete approximating function *potAcc_op*. With the exception of the *allocate* operation, all *dirAcc_op* functions are potential transfers and are therefore approximated by the identity func-

Figure 7.1 Definition and visualization of *dirAcc_approx_dep*.

$$\begin{tikzcd}
S \arrow[r, "dirAcc"] \arrow[d, "(Fs\ S)"'] & ag \arrow[r, dash, "\subseteq"] & ag' \arrow[d, "(Fsa\ S)"'] \\
\mathrm{S2} \arrow[r, "dirAcc"] & ag2 \arrow[r, dash, "\subseteq"] & ag2'
\end{tikzcd}$$

```
Definition dirAcc_approx_dep Fs Fsa := ∀ s s' ag ag' ag2,
  dirAcc_spec s ag → dirAcc_spec (Fs s) ag2 →
    AG.Subset ag ag' → Sys.eq s s' →
    AG.Subset ag2 (Fsa s' ag').
```

tion between potential access graphs. The remaining *allocate* case is approximated by equating the allocator and fresh object. By observing all new access edges must name the fresh object, the proof verifies that all potential access relationships between preexisting objects may only decrease.

7.1 Direct Access Approximations

A solution to the safety problem requires the decidability of whether one object can ever come to hold an access right to another. Many operations in SDM have the ability to remove or overwrite capabilities, making direct comparison difficult. Therefore, the SDM verification first defines the structure of functions that "conservatively approximate" changes to the direct access graph across semantic operations and then defines functions satisfying this requirement.

Figure 7.2 Definition of *dirAcc_approx*.

```
Definition dirAcc_approx Fs Fa := ∀ s ag ag' ag2,
  dirAcc_spec s ag → dirAcc_spec (Fs s) ag2 → AG.Subset ag ag' →
    AG.Subset ag2 (Fa ag').
Theorem dirAcc_approx_dirAcc_approx_dep: ∀ Fs Fa,
  dirAcc_approx Fs Fa → dirAcc_approx_dep Fs (fun s ⇒ Fa).
```

Figure 7.3 Composing sequences of direct access operations.

```
Theorem dirAcc_dep_compose : ∀ Fs, Proper (Sys.eq ⟹ Sys.eq) Fs →
  ∀ Fsa, dirAcc_approx_dep Fs Fsa → ∀ Fs' Fsa', dirAcc_approx_dep Fs' Fsa' →
    dirAcc_approx_dep (compose Fs' Fs)
                      (fun s ⇒ (compose (Fsa' (Fs s)) (Fsa s))).
```

The definition of conservatively approximating functions for direct access are defined by *dirAcc_approx_dep* in Figure 7.1. A visual representation of the relationships is included to more intuitively illustrate the definition. *dirAcc_approx_dep* describes the relationship between a function on system state, such as an operation, and an approximating function between direct access graphs. An approximating function may be dependent on the system state to determine which edges are added. An approximating function must not only produce a superset of the direct access graph, but must continue to do so on other supersets.

Earlier work first attempted to use approximating functions that did not include dependency on a system state. However, determining which approximating function to select for an operation requires knowledge of the system state, leading to the dependent form. The definition of *dirAcc_approx* is identical to *dirAcc_approx_dep* with this omission and consequently entails a *dirAcc_approx_dep* result.

The definition of *dirAcc_approx_dep* permits operations and approximating func-

Figure 7.4 Trivial approximating functions.

```
Definition id_ag (ag:AG.t) := ag.
Definition read_ag := id_ag.
Definition write_ag := id_ag.
Definition revoke_ag := id_ag.
Definition destroy_ag := id_ag.
Theorem dirAcc_approx_read: ∀ a c,
    dirAcc_approx (Sem.do_read a c) read_ag.
Theorem dirAcc_approx_write:∀ a c,
    dirAcc_approx (Sem.do_write a c) write_ag.
Theorem dirAcc_approx_revoke:∀ a t c,
    dirAcc_approx (Sem.do_revoke a t c) revoke_ag.
Theorem dirAcc_approx_destroy:∀ a t,
    dirAcc_approx (Sem.do_destroy a t) destroy_ag.
```

tions to preserve approximation when composed in sequence. Figure 7.3 proves this property as *dirAcc_dep_compose*. The only additional constraint is that functions over system state must respect system states equivalence, a property valid for all operations in SDM.

SDM approximates each operation with a direct access approximating function. These definitions and theorems are provided three modules: **DirectAccess** contains useful functions and lemmas, **DirectAccessSemantics** verifies properites necessary to approximate each operation, and the functions performing approximations are located in **DirectAccessApprox**. Four operations are approximated by the identity function as demonstrated by the theorems in Figure 7.4. The theorems for *dirAcc_read* and *dirAcc_write* follow trivially as the read and write operations do not modify the system state. The proofs of *dirAcc_revoke* and *dirAcc_destroy* demonstrate that no new access edges could be added by observing that the revoke and de-

Figure 7.5 Fetch and store approximating functions.

```
Definition fetch_dep_ag a t c s :=
  ag_add_cap_by_indirect_index a t c s
    (if SemDefns.option_hasRight_dec (SC.getCap t a s) AccessRights.rd
     then (fun c ⇒ c)
     else Cap.weaken) ag_add_cap_valid_std.
Definition store_dep_ag a t c s :=
  ag_push_cap_by_indices a t a c s (fun c⇒c) ag_add_cap_valid_std.
Definition ag_add_cap_by_indirect_index src t i s Fc Fv ag:=
  option_map1
    (fun cap ⇒ ag_add_cap_by_obj_index src (Cap.target cap) i s Fc Fv ag)
    ag (SC.getCap t src s).
Definition ag_push_cap_by_indices o i o' i' s Fc Fv ag:=
  option_map1
    (fun src ⇒ ag_add_cap_by_obj_index src o' i' s Fc Fv ag)
    ag (SemDefns.option_target (SC.getCap i o s)).
Theorem dirAcc_approx_dep_fetch: ∀ a t c i,
  dirAcc_approx_dep (Sem.do_fetch a t c i) (fetch_dep_ag a t c).
Theorem dirAcc_approx_dep_store: ∀ a t c i,
  dirAcc_approx_dep (Sem.do_store a t c i) (store_dep_ag a t c).
```

stroy operations add no new capabilities. Selectively removing edges is not generally a safe operation as it does not generally satisfy the requirements for an approximating function.

The *fetch* and *store* operations add edges for a single capability and are therefore approximated by slightly different surface functions. *Store* is approximated by *ag_push_cap_by_indices* while *ag_add_cap_by_indirect_index* approximates the *fetch* operation. Both functions invoke *ag_add_cap_by_obj_index* which runs *ag_add_cap* to add a capability to the access graph only when it finds a capability at the correct index and the supplied validity test holds. The standard validity test used by most operations is defined by *ag_add_cap_valid_std*. It requires both the source and target of the new access edge to be alive in the system state. If all tests pass, the capability added to the access graph by *ag_add_cap* is first modified by the supplied capability transformation function. The distinction between *ag_add_cap_by_indirect_index* and *ag_push_cap_by_indices* is how the source of the new access edge is named. *ag_add_cap_by_indirect_index* names the source indirectly via another capability and *ag_push_cap_by_indices* names the source of the new edge directly. Additionally, a fetch operation using only a *wk* access right must use the modifying function *weaken*, which returns a *wk*-only capability when *wk* or *rd* are present and an empty access right set otherwise.

The *ag_add_caps_send* and *ag_add_caps_create* functions approxiamte the send and allocate operations. Both functions call *ag_add_caps_by_index_pair_list* to add multiple capabilities via *ag_add_cap_by_obj_index*. The *ag_add_caps_send* function includes a reply capability when specified and uses the standard validity check, while *ag_add_caps_create* always includes a capability with all access rights to the new object and only requires that the source be alive. The case where an object performs a send operation to itself is handled as a simplified special case adding only a reply capability.

All of the presented proofs and definitions are case of a much larger single function approximating an entire operation. The proof unifying all pieces is contained in the **Attenuation** module. Both definitions are presented in Figure 7.7.

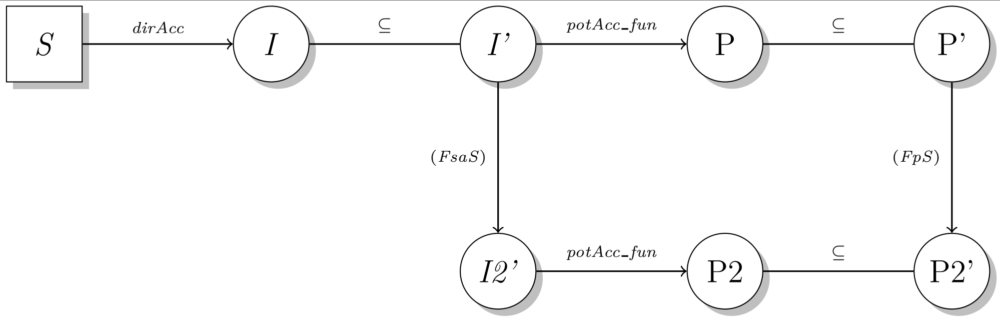
Figure 7.6 Direct access approximations for send and allocate.

```
Definition allocate_dep_ag a n ixi_list s ag:=
  if SemDefns.allocate_preReq_dec a n s
  then ag_add_caps_allocate a n ixi_list s ag
  else ag.
Definition send_dep_ag a t ixi_list s:=
  if (option_map1_eq_tgt_dec t a s)
  then (ag_add_caps_reply a t s)
  else (ag_add_caps_send a t ixi_list s).
Definition ag_add_caps_send a t ixi_list s ag:=
  let ag' := ag_add_caps_reply a t s ag in
    option_map1
      (fun cap ⇒ ag_add_caps_by_index_pair_list
        (Cap.target cap) a ixi_list s (fun c⇒c) ag_add_cap_valid_std ag')
      ag' (SC.getCap t a s).
Definition ag_add_caps_allocate a n ixi_list s ag :=
  (ag_add_cap a (Cap.mkCap n all_rights)
    (ag_add_caps_by_index_pair_list n a ixi_list s (fun c⇒c)
      ag_add_cap_valid_allocate ag)).
Definition ag_add_caps_by_index_pair_list src o (ixi_list:
  list (Ind.t × Ind.t)) s Fc Fv ag :=
    fold_right (fun ixi ag' ⇒ ag_add_cap_by_obj_index src o (fst ixi) s Fc Fv ag')
      ag ixi_list.
Theorem dirAcc_approx_dep_allocate: ∀ a n i ixi_list,
  dirAcc_approx_dep (Sem.do_allocate a n i ixi_list) (allocate_dep_ag a n ixi_list).
Theorem dirAcc_approx_dep_send: ∀ a t ixi_list opt_i,
  dirAcc_approx_dep (Sem.do_send a t ixi_list opt_i) (send_dep_ag a t ixi_list).
```

Figure 7.7 Approximation of all operations as *dirAcc_op*.

```
Definition $dirAcc\_op\ op\ s$ :=
  match $op$ with
  | $Sem.read\ a\ t \Rightarrow read\_ag$
  | $Sem.write\ a\ t \Rightarrow write\_ag$
  | $Sem.fetch\ a\ t\ c\ i \Rightarrow$
    if $SemDefns.fetch\_preReq\_dec\ a\ t\ s$
    then $fetch\_dep\_ag\ a\ t\ c\ s$
    else $id\_ag$
  | $Sem.store\ a\ t\ c\ i \Rightarrow$
    if $SemDefns.store\_preReq\_dec\ a\ t\ s$
    then $store\_dep\_ag\ a\ t\ c\ s$
    else $id\_ag$
  | $Sem.revoke\ a\ t\ c \Rightarrow revoke\_ag$
  | $Sem.send\ a\ t\ ixi\_list\ opt\_i \Rightarrow$
    if $SemDefns.send\_preReq\_dec\ a\ t\ s$
    then $send\_dep\_ag\ a\ t\ ixi\_list\ s$
    else $id\_ag$
  | $Sem.allocate\ a\ n\ i\ ixi\_list \Rightarrow$
    if $SemDefns.allocate\_preReq\_dec\ a\ n\ s$
    then $allocate\_dep\_ag\ a\ n\ ixi\_list\ s$
    else $id\_ag$
  | $Sem.destroy\ a\ t \Rightarrow destroy\_ag$
  end.
Theorem $dirAcc\_approx\_dep\_op$ : $\forall\ op$,
$dirAcc\_approx\_dep\ (Sem.do\_op\ op)\ (dirAcc\_op\ op)$.
```

Figure 7.8 Definition and Visualization of *potAcc_approx_dirAcc_dep*.



```
Definition potAcc_approx_dirAcc_dep Fsa Fp := ∀ i i' p p' p2 s s' s",
  Sys.eq s s' → Sys.eq s' s" →
  dirAcc_spec s i → AG.Subset i i' →
  Seq.potAcc i' p → Seq.potAcc (Fsa s' i') p2 →
  AG.Subset p p' → AG.Subset p2 (Fp s" p').
```

Figure 7.9 Composing sequences of potential access operations.

```
Theorem potAcc_approx_dirAcc_dep_compose:
  ∀ Fs, Proper (Sys.eq ==> Sys.eq) Fs →
  ∀ Fsa, dirAcc_approx_dep Fs Fsa →
  Proper (Sys.eq ==> AG.eq ==> AG.eq) Fsa →
  ∀ Fp, potAcc_approx_dirAcc_dep Fsa Fp →
  ∀ (Fs':Sys.t→Sys.t) Fsa', Proper (Sys.eq ==> AG.eq ==> AG.eq) Fsa' →
  ∀ Fp', potAcc_approx_dirAcc_dep Fsa' Fp' →
  potAcc_approx_dirAcc_dep (fun s ⇒ (compose (Fsa' (Fs s)) (Fsa s)))
                              (fun s ⇒ (compose (Fp' (Fs s)) (Fp s))).
```

7.2 Potential Access Approximations

The structure of conservatively approximating functions between potential access graphs is defined in Figure 7.8. The structure of *potAcc_approx_dirAcc_dep* is still dependent upon the system state, and must therefore retain its relationship through direct access. However, it is only approximating the direct access-approximating function, and does not ensure *Fsa Sa* is approximating direct access. The complete approximation occurs by joining the two definitions. The proof

Figure 7.10 Recomputing potential access preserves *potAcc_approx_dirAcc_dep*.

```
Definition potAcc_approx Fi Fp := ∀ i p p' p2,
  Seq.potAcc i p → Seq.potAcc (Fi i) p2 → AG.Subset p p' →
  AG.Subset p2 (Fp p').
Theorem potAcc_approx_potAcc_fun : ∀ Fa, Seq.ag_potTransfer_fn_req Fa →
  potAcc_approx Fa (fun ag ⇒ Seq.potAcc_fun (Fa ag)).
Theorem potAcc_approx_potAcc_approx_dirAcc_dep: ∀ Fa Fp ,
  potAcc_approx Fa Fp → potAcc_approx_dirAcc_dep (fun s ⇒ Fa) (fun s ⇒ Fp).
```

that *potAcc_approx_dirAcc_dep* composes in Figure 7.9 follows the same form as *dirAcc_approx_dep*.

Composing *potAcc_fun* with another function is often potential access approximating, as demonstrated in Figure 7.10. The *ag_potTransfer_fn_req* judgment requires that the function in question be commutative with set addition and equivalence preserving. This also guarantees that the function is non-decreasing, though not all non-decreasing functions necessarily have these properties[1]. Given these fairly simple requirements of most non-decreasing functions, recomputing potential access after their application always approximates potential access.

With the exception of *allocate*, each direct access-approximating function describes a *potTransfer* between access graphs. Each of these functions is commutative with set addition and equivalence preserving and may be approximated by composing them with *potAcc_fun*. Because these functions form potential transfers on the original potential access graph, this composition is equivalent to the identity function. This cannot hold for allocate, as new objects must be granted new access rights. The only

[1]The definition of *ag_potTransfer_fn_req* contains all three requirements as the non-decreasing proof was constructed later in the work.

Figure 7.11 Direct access functions produce potential transfers.

```
Theorem potTransfer_send_dep_ag: ∀ a t ixi_list s ag ag',
  dirAcc_spec s ag →
  SemDefns.send_preReq a t s →
  Seq.potTransfer ag ag' →
  Seq.potTransfer ag (send_dep_ag a t ixi_list s ag').
Theorem potTransfer_store_dep_ag: ∀ a t c s ag ag',
  dirAcc_spec s ag →
  SemDefns.store_preReq a t s →
  Seq.potTransfer ag ag' →
  Seq.potTransfer ag (store_dep_ag a t c s ag').
Theorem potTransfer_fetch_dep_ag: ∀ a t ixi_list s ag ag',
  dirAcc_spec s ag →
  SemDefns.fetch_preReq a t s →
  Seq.potTransfer ag ag' →
  Seq.potTransfer ag (fetch_dep_ag a t ixi_list s ag').
```

Figure 7.12 Approximating *create_dep_ag* with *endow_dep*.

```
Definition insert a n ag :=
  (ag_add_cap n (Cap.mkCap a all_rights)
    (ag_add_cap a (Cap.mkCap n all_rights) ag)).
Definition endow a n ag := (Seq.potAcc_fun (insert a n ag)).
Definition endow_dep a n s :=
  if SemDefns.allocate_preReq_dec a n s
  then endow a n
  else fun ag ⇒ ag.
Theorem potAcc_approx_allocate : ∀ a n ixi_list,
 potAcc_approx_dirAcc_dep (allocate_dep_ag a n ixi_list) (endow_dep a n).
```

remaining question for the safety property is whether these new capabilities increase the permissions between existing objects.

The allocate operation is approximated by *endow_dep* as demonstrated by Figure 7.12. The *endow_dep* function first checks if the allocate operation is performed before executing the *endow* function. The *endow* function approximates allocate by fully connecting the allocator and fresh objects using the *insert* function and then

Figure 7.13 Potential access approximation of *dirAcc_op* by *potAcc_op*.

```
Definition potAcc_op op s :=
match op with
| Sem.read a t ⇒ id_ag
| Sem.write a t ⇒ id_ag
| Sem.fetch a t c i ⇒ id_ag
| Sem.store a t c i ⇒ id_ag
| Sem.revoke a t c ⇒ id_ag
| Sem.send a t ixi_list opt_i ⇒ id_ag
| Sem.allocate a n i ixi_list ⇒ endow_dep a n s
| Sem.destroy a t ⇒ id_ag
end.
Theorem potAcc_approx_dirAcc_dep_op :
  ∀ op, potAcc_approx_dirAcc_dep (dirAcc_op op) (potAcc_op op).
```

recomputing potential access. Because all capability transfers during allocate are captured as potential transfers after the initial capability is created, *endow_dep* must approximate the allocate operation. Either capability added by the *insert* function entails the other in potential access. However, the *insert* function adds both capabilities to facilitate simpler inference in future proofs.

Each of these approximations has been a special case of a larger approximating function: *potAcc_op*. Like *dirAcc_op*, *potAcc_op* approximates the potential access of every operation. Potential access is approximated by the identity function for all operations except allocate, where it is approximated by *endow_dep*. This proof is constructed by case analysis given the previous approximations.

Figure 7.14 Definition of *AG_attenuating*.

```
Definition AG_attenuating N p p' :=
  ∀ src tgt, RefSet.In src N → RefSet.In tgt N →
  ∀ rgt, ¬ AG.In (Edges.mkEdge src tgt rgt) p →
    Ref.eq src tgt ∨ ¬ AG.In (Edges.mkEdge src tgt rgt) p'.
```

Figure 7.15 Properties of *AG_attenuating*.

```
Theorem AG_attenuating_trans_subset: ∀ N p p' N' p'',
  AG_attenuating N p p' → AG_attenuating N' p' p'' →
  RefSet.Subset N N' → AG_attenuating N p p''.
Theorem AG_attenuating_trans : ∀ N p p' p'',
  AG_attenuating N p p' → AG_attenuating N p' p'' → AG_attenuating N p p''.
Theorem AG_attenuating_subset_ag: ∀ N p p',
  AG.Subset p' p → AG_attenuating N p p'.
```

7.3 Attenuating Permissions

The next goal of this chapter is to demonstrate how potential access is attenuating for each operation. The definition of attenuating is given by *AG_attenuating* of Figure 7.14. The access graph *p'* is attenuating from *p* with respect to a set of objects *N* when all relationships in p' between elements in N can be no worse than the relationships in p.

Because new self-targeting access edges may arise via allocation, a simple subset relation will not suffice to capture attenuation. This occurs because all objects are assumed to have total self-authority, but potential transfer must remain finite. An isolated object will not appear in the direct access graph if it has no capabilities, but the allocate operation adds new access edges via capabilities that justify its inclusion. This case is largely uninteresting, but must be handled for precision.

The fact that attenuations are transitive will be used extensively to verify the

Figure 7.16 insert and endow are attenuating.

```
Theorem AG_attenuating_insert: ∀ ag p a n ag' p' objs N,
  Seq.potAcc ag p →
  Seq.ag_objs_spec p N →
  ¬ RefSet.In n N →
  ¬ RefSet.In n objs →
  AG.Equal (insert a n p) ag' →
  Seq.potAcc ag' p' →
  AG_attenuating objs p p'.
Theorem AG_attenuating_endow : ∀ ag p a n objs Np,
  Seq.ag_objs_spec p Np →
  ¬ RefSet.In n objs → ¬ RefSet.In n Np →Seq.potAcc ag p →
    AG_attenuating objs p (endow a n p).
```

safety property. Additionally, any subset is trivially attenuating, causing almost all cases of *potAcc_op* to be attenuating. Discussing an attenuation between objects that are unborn is generally nonsensical. Moreover, the set of objects examined for safety will be all those alive or dead in the system state. Therefore, during the allocate operation, the set of objects under consideration should not name the fresh object. To show that authority is attenuating over *endow*, it is sufficient to demonstrate that all new edges have the fresh object as a source or target.

The *insert* function is attenuating by simple case analysis. All edges between the allocator and fresh object have the fresh object as a source or target. Because attenuations are transitive, adding them in any order produces an attenuation for the whole function.

The proof that the *endow* function is attenuating requires substantial case analysis. The following is a simple sketch by contradiction. If *endow* is applied to a maximal access graph, then all edges added by transfer must have the fresh object as a source

Figure 7.17 *AG_project* and *endow*.

```
Definition AG_project a n p p' := ∀ src tgt rgt,
   AG.In (Edges.mkEdge src tgt rgt) p' ↔
    (AG.In (Edges.mkEdge src tgt rgt) p ∨
     AG.In (Edges.mkEdge src a rgt) p ∧ Ref.eq tgt n ∨
     AG.In (Edges.mkEdge a tgt rgt) p ∧ Ref.eq src n ∨
     Ref.eq src n ∧ Ref.eq tgt a ∨
     Ref.eq src a ∧ Ref.eq tgt n ∨
     Ref.eq src n ∧ Ref.eq tgt n ∨
     Ref.eq src a ∧ Ref.eq tgt a).
Theorem AG_project_endow : ∀ ag p, Seq.potAcc ag p →
  ∀ N, Seq.ag_objs_spec p N → ∀ n, ¬ RefSet.In n N →
  ∀ a ag', AG.Equal (insert a n p) ag' → ∀ p', Seq.potAcc ag' p' →
    AG_project a n p p'.
```

Figure 7.18 *potAcc_op* is attenuating.

```
Theorem AG_attenuating_potAcc_op: ∀ s op ag p,
  dirAcc_spec s ag → Seq.potAcc ag p →
  ∀ Np, Seq.ag_objs_spec p Np → ∀ objs, objs_not_unborn objs s →
    AG_attenuating objs p (potAcc_op op s p).
```

or target. This property is called a projection and is captured by the *AG_project* judgment in Figure 7.17. Recall that any transfer on a maximal access graph produces an identical access graph. Were this not the case, the edges required to perform such a transfer must also not identify the fresh object, by inspection. However, if these edges did not identify the fresh object, the transfer must have been valid in the maximal graph, so such an edge cannot exist. As all cases of *potAcc_op* must be attenuating, *potAcc_op* is also attenuating.

Figure 7.19 Approximating sequences of direct access operations.

```
Fixpoint dirAcc_execute op_list s : (AG.t → AG.t) :=
match op_list with
  | nil ⇒ id_ag
  | cons op tail ⇒
    compose (dirAcc_op op (Exe.execute s tail)) (dirAcc_execute tail s)
end.
Inductive dirAcc_execute_spec :
  list Sem.operation → (Sys.t → AG.t → AG.t) → Prop :=
| dirAcc_execute_spec_nil : dirAcc_execute_spec nil (fun (s:Sys.t) (a:AG.t) ⇒ a)
| dirAcc_execute_spec_cons : ∀ op op_list Fp, dirAcc_execute_spec op_list Fp →
  dirAcc_execute_spec (cons op op_list)
    (fun s ⇒ compose (dirAcc_op op (Exe.execute s op_list)) (Fp s)).
Theorem dirAcc_execute_spec_eq_iff: ∀ opList Fsa,
  dirAcc_execute_spec opList Fsa ↔ Fsa = (dirAcc_execute opList).
Theorem dirAcc_execute_spec_dirAcc_execute: ∀ op_list,
  dirAcc_execute_spec op_list (dirAcc_execute op_list).
```

Figure 7.20 Approximating sequences of potential access operations.

```
Fixpoint potAcc_execute op_list s : (AG.t → AG.t) :=
  match op_list with
    | nil ⇒ id_ag
    | cons op tail ⇒
    compose (potAcc_op op (Exe.execute s tail)) (potAcc_execute tail s)
  end.
Inductive potAcc_execute_spec :
  list Sem.operation → (Sys.t → AG.t → AG.t) → Prop :=
| potAcc_execute_spec_nil : potAcc_execute_spec nil (fun (s:Sys.t) (a:AG.t) ⇒ a)
| potAcc_execute_spec_cons : ∀ op op_list Fp, potAcc_execute_spec op_list Fp →
  potAcc_execute_spec (cons op op_list)
    (fun s ⇒ compose (potAcc_op op (Exe.execute s op_list)) (Fp s)).
Theorem potAcc_execute_spec_eq_iff: ∀ opList Fsa,
  potAcc_execute_spec opList Fsa ↔ Fsa = (potAcc_execute opList).
Theorem potAcc_execute_spec_potAcc_execute: ∀ op_list,
  potAcc_execute_spec op_list (potAcc_execute op_list).
```

Figure 7.21 Visualization of approximating execution.

$$\begin{tikzcd}
S_0 \arrow[r, "dirAcc"] \arrow[d, "do\_op_1"'] & I_0 \arrow[r, dash, "\subseteq"] & I'_0 \arrow[r, "potAcc\_fun"] \arrow[d, "Fsa_1"'] & P_0 \arrow[r, dash, "\subseteq"] & P'_0 \arrow[d, "Fp_1"'] \\
S_1 \arrow[r, "dirAcc"] \arrow[d, "do\_op_{N-1}"'] & I_1 \arrow[r, dash, "\subseteq"] & I'_1 \arrow[r, "potAcc\_fun"] \arrow[d, "Fsa_2"'] & P_1 \arrow[r, dash, "\subseteq"] & P'_1 \arrow[d, "Fp_2"'] \\
\ldots \arrow[d, "do\_op_N"'] & \ldots & \ldots \arrow[d, "Fsa_N"'] & \ldots & \ldots \arrow[d, "Fp_N"'] \\
S_N \arrow[r, "dirAcc"] & I_N \arrow[r, dash, "\subseteq"] & I'_N \arrow[r, "potAcc\_fun"] & P_N \arrow[r, dash, "\subseteq"] & P'_N
\end{tikzcd}$$

7.4 Safety

Having demonstrated how the potential access of each operation can be conservatively approximated by attenuating functions, the remainder of this chapter illustrates how these functions are composed to produce safety. Each fully instantiated attenuating function can be composed to produce attenuation transitively. However, each is dependent on a system state which must be computed after each operation. The functions *dirAcc_execute* and *potAcc_execute* approximate a sequence of operations given an initial system state. Both functional and inductive specifications are included.

The approximating functions correctly approximate the access graphs of each

Figure 7.22 Folding approximating operations for executions.

```
Theorem dirAcc_execute_approx :
  ∀ op_list Fsa, dirAcc_execute_spec op_list Fsa →
    dirAcc_approx_dep (fun s ⇒ (Exe.execute s op_list)) Fsa.
Theorem potAcc_execute_approx :
  ∀ op_list Fsa, dirAcc_execute_spec op_list Fsa →
  ∀ Fp, potAcc_execute_spec op_list Fp →
    potAcc_approx_dirAcc_dep Fsa Fp.
```

Figure 7.23 Nodes that are not unborn remain not unborn.

```
Definition objs_not_unborn objs s :=
  ∀ x : RefSet.elt, RefSet.In x objs → ¬ SC.is_unborn x s.
Theorem objs_not_unborn_oplist : ∀ n s, objs_not_unborn n s →
  ∀ opL s', Exe.execute_def s opL s' → objs_not_unborn n s'.
```

operation sequence. Composing direct access operations approximates the direct access of an operation sequence. Having performed that approximation, the potential access of an operation sequence is approximated by folding over *potAcc_op*. These properties are defined in Figure 7.22 and visualized in Figure 7.21.

As previously stated, it is nonsensical to discuss unborn objects in the attenuating judgment. The predicate *objs_not_unborn* determines that all object references of a set do not name unborn objects in the given system state. As access graphs do not track this information, it will be added as a precondition to the safety problem.

Verifying that no access edges added during endowment identify any existing objects is not sufficient to verify the safety property. The change in objects present in

Figure 7.24 How the objects in an access graph change through endowment.

```
Theorem ag_objs_spec_endow: ∀ p p_objs, Seq.ag_objs_spec p p_objs →
  ∀ a n p'_objs, Seq.ag_objs_spec (endow a n p) p'_objs →
    RefSet.eq p'_objs (RefSet.add a (RefSet.add n p_objs)).
```

Figure 7.25 Attenuations are composable

Theorem *AG_attenuating_compose*: ∀ *objs p Fp1*, *AG_attenuating objs p* (*Fp1 p*) →
  ∀ *Fp2*, *AG_attenuating objs* (*Fp1 p*) (*compose Fp2 Fp1 p*) →
  AG_attenuating objs p (*compose Fp2 Fp1 p*).

Figure 7.26 Executing a sequence of potential access operations produces a maximal access graph.

Theorem *potAcc_execute_spec_potAcc*: ∀ *p*, *Seq.maxTransfer p* →
  ∀ *op_list Fp'*, *potAcc_execute_spec op_list Fp'* →
  ∀ *s*, *Seq.maxTransfer* (*Fp' s p*).

an access graph must be known precisely. The only new objects which may arise from the endow function are the newly allocated object and the parent, if it was absent. This is verified by *ag_objs_spec_endow* in Figure 7.24.

There are two helper functions to assist the proof that access graphs are attenuating over a sequence of operations. The first uses the proof that attenuations are transitive to demonstrate how functions producing attenuations can be composed. The other helper theorem proves that *potAcc_execute* is maximal, provided its initial access graph is maximal. This ensures that *potAcc_execute* is a potential access graph producing function, in addition to being potential access approximating.

The proof that *potAcc_execute* is attenuating over any objects that are not unborn in the initial system state is given in Figure 7.27. Each operation is approximated by

Figure 7.27 Executing a potential access operation is attenuating.

Theorem *execute_potAcc_attenuating*:
    ∀ *op_list Fsa*, *dirAcc_execute_spec op_list Fsa* →
    ∀ *Fp*, *potAcc_execute_spec op_list Fp* →
    ∀ *s i*, *dirAcc_spec s i* → ∀ *p*, *Seq.potAcc i p* →
    ∀ *objs*, *objs_not_unborn objs s* →
    AG_attenuating objs p (*Fp s p*).

Figure 7.28 Execution is attenuating: Safety for Capability systems.

`Theorem` *execute_attenuating* : ∀ *s i*, *dirAcc_spec s i* → ∀ *p*, *Seq.potAcc i p* →
  ∀ *op_list s'*, *Exe.execute_def s op_list s'* → ∀ *i'*, *dirAcc_spec s' i'* →
  ∀ *p'*, *Seq.potAcc i' p'* → ∀ *objs*, *objs_not_unborn objs s* →
    AG_attenuating objs p p'.

an attenuating function for both direct access and potential access. The attenuating *potAcc_execute* function produces a maximal access graph from one already maximal, requiring no additional analysis when applied to an approximating potential access graph. By induction, that each step in *potAcc_execute* composes to produce another attenuating function. Therefore *potAcc_execute* must be attenuating.

The final theorem of this chapter is the verification of the safety property for capability-based systems. As the various approximating functions can be constructed as needed, the proof that the potential access is attenuating simply forgets them. Because any attenuations are preserved by subset, the potential access of any future system state must be an attenuation because each approximation is a attenuation. Since the permissions between any objects that are not unborn must be attenuating with each operation, the potential access initially produced must be preserved through the life of the system, satisfying the safety property.

Chapter 8

Information Flow

This chapter focuses on analyzing how the flow of data in SDM is related to potential access. As previously mentioned, the model does not directly capture application data; the transfer of application data is abstracted using the *readFrom* and *wroteTo* definitions. From these definitions, SDM defines what is *mutated* through a sequence of operations. It also supplies the *mutable* judgment: a simple, permission-based definition of mutability by directly examining an access graph. The remainder of the chapter examines how what is mutated through a sequence of operations is a subset of what is considered potentially mutable. Naively, this is demonstrated by observing that all information flow occurs by the existence of a capability and that, for each operation, what is mutated is a subset of what is mutable in the direct access. By capturing all potential capabilities, the computation of potential access has also captured all potential information flow.

8.1 Mutation

The semantics of SDM do not incorporate data values directly. Instead, they define how data may flow during each operation through the use of the *readFrom_spec* and *wroteTo_spec* relations. As their names imply, the *readFrom_spec* judgment indicates the potential information flow sources during an operation, and *wroteTo_spec* indicates the potential destinations. A complete system implementation must demonstrate that its operations do not violate these judgments, providing implementations of *readFrom* and *wroteTo* satisfying those specifications.

For review, the definition of *readFrom_spec* and *wroteTo_spec* capture potential flows of both data and capabilities. The simple *read* and *write* operations that have no impact on the system state indicate data-only data motion. Because *it is possible to encode data using capabilities*, their potential transfers are considered as information flow. While necessary for correctness, this is not the case for all models and many incorrectly label such flow as *covert.* Thus, the fetch and store operations admit the same mutability as *read* and *write.* Sending a message may contain both data and capabilities, and has the same mutation as *write.* Like the *send* operation, allocate is able to pass multiple capabilities and arbitrary data along for object instantiation. The *revoke* operation has the same flow potential as *write.* In many systems the revoke command is performed by a *write* of a trivially non-mutating capability. Though it alters the state of an object, the *destroy* operation is not modeled as effecting mutation upon that object. When an object is destroyed, subsequent operations on that object

Figure 8.1 Definition of *mutated*.

```
Inductive mutated_op_def n s op : RefSet.t → Prop :=
| mutated_op_not_in : ∀ rf n', RefSet.eq n n' → Sem.read_from_def s op rf →
   ¬ RefSet.Exists (fun x ⇒ RefSet.In x rf) n → mutated_op_def n s op n'
| mutated_op_valid_in : ∀ rf n', RefSet.eq n n' → Sem.read_from_def s op rf →
   RefSet.Exists (fun x ⇒ RefSet.In x rf) n →
   ∀ wt, Sem.wrote_to_def s op wt →
     ∀ n2, RefSet.eq (RefSet.union n' wt) n2 →
     mutated_op_def n s op n2.
Inductive mutated_def n (s:Sys.t) : (list Sem.operation) → RefSet.t → Prop :=
| mutated_nil : ∀ n', RefSet.eq n n' → mutated_def n s nil n'
| mutated_cons : ∀ opList n2, mutated_def n s opList n2 →
  ∀ s', Exe.execute_def s opList s' →
     ∀ op n3, mutated_op_def n2 s' op n3 →
        mutated_def n s (cons op opList) n3.
```

do not admit any outward information flow, not even the state of the object.[1]

The *mutated_def* inductive models where information might flow during a sequence of operations. Specifically, *mutated_def* considers the objects reachable by an initial subsystem in an initial system state. As a precondition, the initial subsystem must contain objects that are alive or dead. The algorithm is initialized with the initial subsystem being the reachable subsystem, as all objects are considered self-mutating. Each time an operation is performed the reachable subsystem is expanded to include the *wroteTo* set if an element of the subsystem is present in the *readFrom* set. After all operations have been executed, the resulting reachable subsystem contains all locations information in the initial subsystem could have reached as a result of that operation sequence.

[1]This is hiding a subtle issue and is addressed in Chapter 11.

Figure 8.2 Definition of mutable.

```
Definition mutable_spec p objs mut := ∀ x, RefSet.In x mut ↔
  RefSet.In x objs ∨
  (∃ e, RefSet.In e objs ∧
    (AG.In (Edges.mkEdge e x tx) p ∨
      AG.In (Edges.mkEdge e x wr) p ∨
      AG.In (Edges.mkEdge x e wk) p ∨
      AG.In (Edges.mkEdge x e rd) p)).
```

Figure 8.3 Properties of mutable.

```
Theorem mutable_spec_subset:
  ∀ n n', RefSet.Subset n' n → ∀ p p', AG.Subset p' p →
  ∀ m, mutable_spec p n m → ∀ m', mutable_spec p' n' m' →
    RefSet.Subset m' m.
Theorem Proper_mutable_spec:
  Proper (AG.eq ==> RefSet.eq ==> RefSet.eq ==> impl) mutable_spec.
```

8.2 Mutability

In contrast to the flows that occur over operations, the abstract concept of possible mutation as authorized by access rights is captured by the *mutable* judgment. Intuitively, most developers expect the potential for information transfer to be expressed using system permissions. A definition of *mutable* that relies on *mutated* would defeat such intuitions as they have been captured by potential access and the safety property. Therefore, *mutable* is defined simply by inspecting the permissions of an access graph, without considering possible operations. If any object in the initial subsystem is the source of a *wr* or *tx* access edge, the target is mutable by the subsystem. Additionally, if the target of a *rd* or *wk* access edge is in the initial subsystem, then source is mutable by the subsystem.

This definition of mutable capturing the instantaneous information flow of each access right within any access graph. Mutable is monotonic over the access graph, following from the intuition that increasing access rights increases mutability. This allows all proofs describing approximations over access graphs to be lifted to approximate mutability. The fact that a subsystem can self-mutate is captured directly. *Mutable* is also monotonic over the initial subsystem, causing it to be non-decreasing. Applying mutable to a direct access graph is called direct mutability while applying it to potential access graph is called potential mutability. Because direct access is a subset of potential access, direct mutability is a subset of potential mutability.

8.3 Operational Mutability

The hypothesis that all direct mutation is captured by direct mutability and all potential mutation by potential mutability does not match the definition of mutable because it does not capture any transitivity of information flow. For access graph approximations to successfully capture mutation, the opportunity to capture the transitivity of flow must occur when approximating potential access. Naively applying *mutable* after each potential access approximating operation produces an induction strategy based on potential transfer. However, the induction strategy of potential transfer is insufficient to describe how mutability grows from the initial subsystem. Because *mutable* is a static definition and does not describe how mutability changes over operations, it cannot approximate *mutated*. A definition of mutability that fol-

Figure 8.4 Figure and definition for *general_approx_dirAcc_dep*.

$$\begin{tikzcd}
\boxed{S} \arrow[r, "dirAcc"] & D \arrow[r, dash, "\subseteq"] & I \arrow[r, "Fg"] \arrow[d, "(Fsa\ S)"'] & I' \arrow[r, dash, "\subseteq"] & P \arrow[d, "(Fsa'\ S)"'] \\
 & & I_2 \arrow[r, "Fg"] & I_2' \arrow[r, dash, "\subseteq"] & P_2
\end{tikzcd}$$

```
Definition general_approx_dirAcc_dep
  (Fg: AG.t → AG.t) (Fsa Fsa': Sys.t → AG.t → AG.t) :=
    ∀ s s', Sys.eq s s' →
    ∀ s", Sys.eq s' s" →
    ∀ i, dirAcc_spec s i →
    ∀ i', AG.Subset i i' →
    ∀ p', AG.Subset (Fg i') p' →
      AG.Subset (Fg (Fsa s' i')) (Fsa' s" p').
Definition approx_dirAcc_dep Fg Fsa Fsa' :=
  general_approx_dirAcc_dep Fg Fsa Fsa' ∧
  ag_nondecr Fg ∧
  Seq.ag_equiv Fg ∧
  Proper (Sys.eq ⟹ AG.eq ⟹ AG.eq) Fsa ∧
  Proper (Sys.eq ⟹ AG.eq ⟹ AG.eq) Fsa'.
```

lows the induction strategy of *mutated* is needed.

Operational mutability is the concept of folding *mutable* over an operation sequence in the same manner as *mutated*. However, because operational mutability must be general enough to extend to all partial potential transfers approximating *dirAcc_execute*, it has a slightly complicated structure. *general_approx_dirAcc_dep* is constructed as a generalized form of *potAcc_approx_dirAcc_dep* that relates two access graph transformations by the function *Fg*. *potAcc_approx_dirAcc_dep* becomes the special case where *Fg* is *potAcc_fun* and the related transformations are *dirAcc_execute* and *potAcc_execute*. *approx_dirAcc_dep* places additional constraints

Figure 8.5 Operational mutability.

```
Definition indexed {A B:Type} (R: relation A) (R': relation B)
  (ind: A → B → Prop) :=
    ∀ (a a':A), (R a a') → ∀ (b:B), ind a b → ∀ b', ind a' b' → (R' b b').
Inductive mutable_execute Fg
   (exe_spec: list Sem.operation → (Sys.t → AG.t → AG.t) → Prop) :
   list Sem.operation → Sys.t → RefSet.t → RefSet.t → Prop :=
| mutable_execute_nil: ∀ Fdx, dirAcc_execute_spec nil Fdx →
  indexed eq eq exe_spec →
  ∀ Ftx, exe_spec nil Ftx → approx_dirAcc_dep Fg Fdx Ftx →
  ∀ s i, dirAcc_spec s i →
  ∀ E M, mutable_spec (Ftx s (Fg i)) E M →
    mutable_execute Fg exe_spec nil s E M
| mutable_execute_cons: ∀ opList Fdx, dirAcc_execute_spec opList Fdx →
  indexed eq eq exe_spec →
  ∀ Ftx, exe_spec opList Ftx → approx_dirAcc_dep Fg Fdx Ftx →
  ∀ s i, dirAcc_spec s i →
  ∀ E M, mutable_execute Fg exe_spec opList s E M →
  ∀ op Fdx', dirAcc_execute_spec (cons op opList) Fdx' →
  ∀ Ftx', exe_spec (cons op opList) Ftx' → approx_dirAcc_dep Fg Fdx' Ftx' →
  ∀ M', mutable_spec (Ftx' s (Fg i)) M M' →
    mutable_execute Fg exe_spec (cons op opList) s E M'.
```

on *general_approx_dirAcc_dep*. It requires all functions to be equivalence preserving and restricts *Fg* to also be non-decreasing; these conditions are satisfied by all potential transfers and approximating functions.

Operational mutability, defined by *mutable_execute* in Figure 8.5, folds *mutable* over each operation to aligns with the induction strategy of mutated. Unlike the static definition of mutable, operational mutability is defined inductively over a list of operations. Like *mutated*, it starts with an initial system state and subsystem, and grows these based on what is statically mutable after each step. However, rather than determining what was mutated, it is parameterized over a specification for approx-

Figure 8.6 Properties of operational mutability.

```
Theorem Proper_mutable_execute :
  Proper (eq ==> eq ==> eq ==> Sys.eq ==> RefSet.eq ==> RefSet.eq ==> iff)
    mutable_execute.
Theorem mutable_execute_subset:
  ∀ Fg Fg' exe_spec exe_spec',
  ∀ opList s E m, mutable_execute Fg exe_spec opList s E m →
  ∀ E', RefSet.Subset E E' →
  ∀ m', mutable_execute Fg' exe_spec' opList s E' m' →
  (∀ opl FXa, exe_spec opl FXa → ∀ FXb, exe_spec' opl FXb →
      ∀ s s', Sys.eq s s' → ∀ i, dirAcc_spec s i →
      ∀ a, AG.Subset i a → ∀ a', AG.Subset a a' →
      AG.Subset (FXa s (Fg a)) (FXb s' (Fg' a'))) →
    RefSet.Subset m m'.
```

imating *dirAcc_execute* and a function relating this specification to *dirAcc_execute*, using *approx_dirAcc_dep*. Because the executable specification is a relation between operation lists and functions, the *indexed* constraint ensures that all equivalent lists are related to equivalent functions. As will all other general properties, this is a fairly trivial property for all relevant functions.

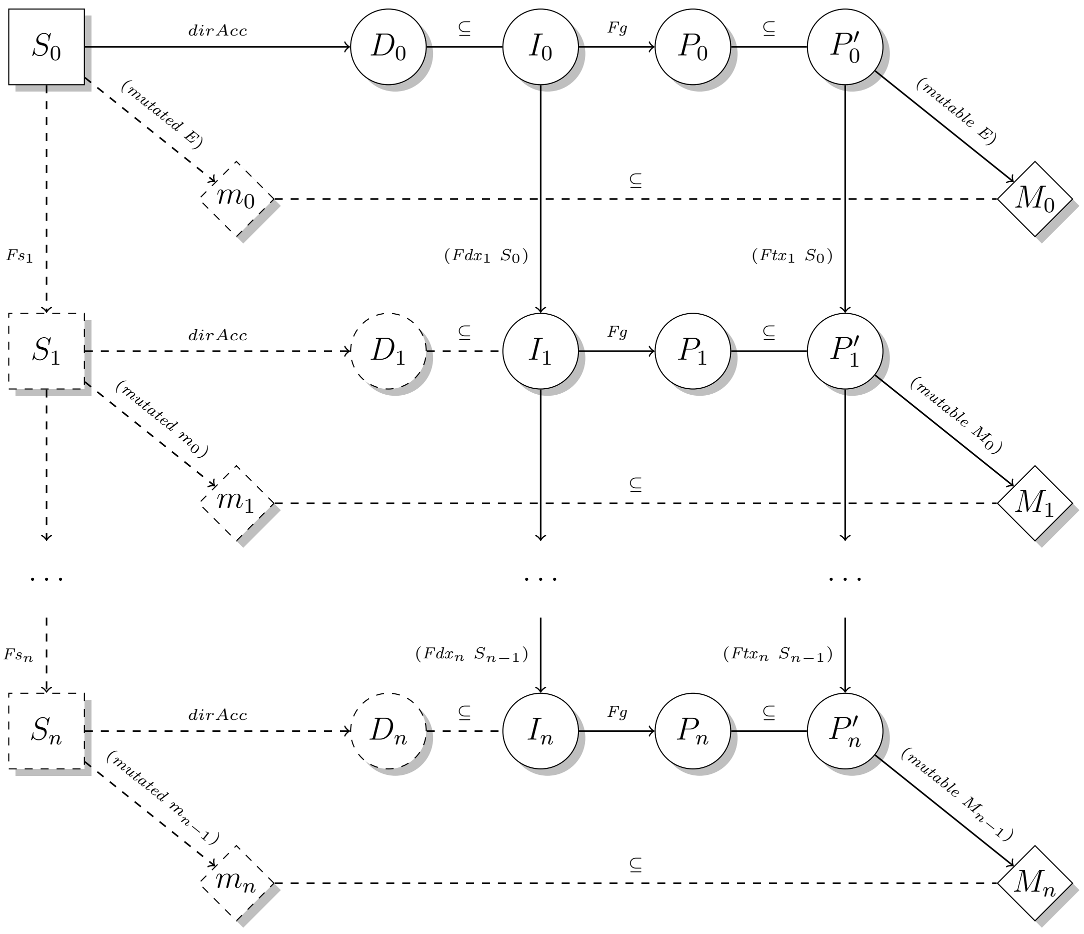
Operational Mutability has two very useful properties: it preserves access graph equivalence and subset relationships. This generalization applies to any execution sequence in a step-wise approximating relationship with *dirAcc_execute*, including all potential transfer approximations including *potAcc_execute*. The proof that the operational mutability of direct access approximations must be bounded by the operational mutability of potential access approximations involves specializing these theorems correctly. This result forms the foundation of information flow analysis for the remainder of this chapter.

Figure 8.7 Specialized forms of *mutable_execute*.

```
Definition mutable_dirAcc_execute :=
  mutable_execute (fun a ⇒ a) dirAcc_execute_spec.
Definition mutable_potAcc_execute :=
  mutable_execute Seq.potAcc_fun potAcc_execute_spec.
Theorem exists_mutable_dirAcc_execute:
  ∀ opList s E, ∃ m, mutable_dirAcc_execute opList s E m.
Theorem exists_mutable_potAcc_execute:
  ∀ opList s E, ∃ m, mutable_potAcc_execute opList s E m.
Theorem mutable_execute_dirAcc_subset_potAcc:
  ∀ s s', Sys.eq s s' →
  ∀ E E', RefSet.Subset E E' →
  ∀ opList m, mutable_dirAcc_execute opList s E m →
  ∀ m', mutable_potAcc_execute opList s' E' m' →
    RefSet.Subset m m'.
```

Operational mutability is specialized for direct access approximations using the identity function and for potential access approximations using *potAcc_fun*. Specialized forms are constructed by simple instantiation and checking completeness. For completeness, the proof that the mutability of *potAcc_execute* approximates the mutability of *dirAcc_execute* is also instantiated.

Figure 8.8 Relationships between mutated and operational mutability.



8.4 Mutation is always Mutable

Figure 8.8 illustrates the relationship between mutated and operational mutability as one induction strategy. This diagram introduces some additional notation. Sets of object references are depicted with drop-shadow diamonds for both mutated and mutable sets. Also, the figure contains dashed lines to indicate portions of operational mutability that are not part of its definition, but are intended to connect to the definition. The diagram has been flattened to appear sequential when this is not

Figure 8.9 Approximations of Mutated

```
Theorem mutable_dirAcc_execute_approx_mutated :
  ∀ s s', Sys.eq s s' →
  ∀ E E', RefSet.eq E E' →
  ∀ opList m, mutated_def E s opList m →
  ∀ m', mutable_dirAcc_execute opList s' E' m' →
     RefSet.Subset m m'.
Theorem mutable_potAcc_execute_approx_mutated:
  ∀ s s', Sys.eq s s' →
  ∀ E E', RefSet.Subset E E' →
  ∀ opList m, mutated_def E s opList m →
  ∀ m', mutable_potAcc_execute opList s' E' m' →
     RefSet.Subset m m'.
```

actually the case. All functions are actually parameterized over the initial system state and direct access graph and capture all possible functions approximating the whole operation sequence. Though slightly overspecialized, the resulting diagram is more intuitive and more closely aligns with other diagrams in this document.

Direct operational mutability approximates potential operational mutability. To show that direct operational mutability approximates what is mutated, it is sufficient to show that potential operational mutability approximates what is mutated.

It must be the case that potential operational mutability approximates mutated by induction on operations. Presume the subsystem analyzed for potential operational mutability is a superset of that analyzed for mutation. The prerequisite capability for each operation is represented as a collection of access edges in the direct access graph. By case analysis, the mutability of these access edges directly capture the information flow possible for this operation. However, potential operational mutability grows its subsystem by all possible mutations, producing a superset of what was mutated. In the base case, nothing is mutated and the potential operational mutability is simply direct mutability, which is a superset of the initial subsystem. Therefore, potential operational mutability approximates mutated, and this result must extend to direct operational mutability using previous results.

The remainder of this section describes how initial potential mutability and direct operational mutability are related. Specifically, the goal is to demonstrate that initial potential mutability for existing objects will never change and subsumes all direct operational mutability. This analysis begins by examining the relationships of potential access approximations, as shown in Figure 8.10. First, *mutable* is a maximal result when applied to a maximal access graph, as shown in *mutable_maximal*. That is, each potential access graph has captured all potential information flow for existing objects and operational mutability does not grow for operations that do not alter potential access. Since *mutable* preserves subset, it also can't be any smaller and must therefore be equal. Second, operational mutability grows precisely with each projection, which occurs with each allocation. After a projection, operational mutability grows only by the allocated object exactly when the allocator is in the previous mutable set, otherwise it is unchanged. That is, in the case of *allocate*, operational mutability must only grow by precisely the allocated object when the allocator is mutable.

These definitions rely upon the *obj_existed* judgment that identifies those objects either alive or dead in a system state. Objects that existed must be kept disjoint from objects that are valid in a projection, as projecting to a previously existing object

Figure 8.10 Relationships of mutable in potential access approximations.

```
Theorem AG_project_maximal:
  ∀ p, Seq.maxTransfer p →
  ∀ objs, Seq.ag_objs_spec p objs →
  ∀ N, RefSet.Subset objs N →
  ∀ o, ¬ RefSet.In o N →
  ∀ a p', AG_project a o p p' →
     Seq.maxTransfer p'.
Theorem mutable_maximal:
  ∀ p, Seq.maxTransfer p →
  ∀ E m, mutable_spec p E m →
  ∀ m', mutable_spec p m m' →
     RefSet.Subset m' m.
Theorem mutable_project_not_in_eq:
  ∀ p, Seq.maxTransfer p →
  ∀ objs, Seq.ag_objs_spec p objs →
  ∀ N, RefSet.Subset objs N →
  ∀ E, RefSet.Subset E N →
  ∀ a, RefSet.In a N →
  ∀ o, ¬ RefSet.In o N →
  ∀ p', AG_project a o p p' →
  ∀ m, mutable_spec p E m → ¬ RefSet.In a m →
  ∀ m', mutable_spec p' m m' →
     RefSet.eq m' m.
Theorem mutable_project_in_eq:
  ∀ p, Seq.maxTransfer p →
  ∀ objs, Seq.ag_objs_spec p objs →
  ∀ N, RefSet.Subset objs N →
  ∀ E, RefSet.Subset E N →
  ∀ a, RefSet.In a N →
  ∀ o, ¬ RefSet.In o N →
  ∀ p', AG_project a o p p' →
  ∀ m, mutable_spec p E m → RefSet.In a m →
  ∀ m', mutable_spec p' m m' →
  ∀ E', RefSetProps.Add o m E' →
     RefSet.eq m' E'.
```

Figure 8.11 Initial potential mutability and Initial mutability are upper bounds on mutation for existing objects.

```
Theorem mutable_potAcc_execute_approx:
  ∀ opList S E Mx, mutable_potAcc_execute opList S E Mx →
  ∀ D, dirAcc_spec S D →
  ∀ P, Seq.potAcc D P →
  ∀ Ex, obj_existed Ex S →
  RefSet.Subset E Ex →
  ∀ M, mutable_spec P E M →
    (RefSet.Subset (RefSet.inter Mx Ex) M).
Theorem mutable_approx_mutated:
  ∀ opList S E m, mutated_def E S opList m →
  ∀ D, dirAcc_spec S D →
  ∀ P, Seq.potAcc D P →
  ∀ Ex, obj_existed Ex S →
  RefSet.Subset E Ex →
  ∀ M, mutable_spec P E M →
    (RefSet.Subset (RefSet.inter m Ex) M).
```

violates existing information flows. In theorems not involving a system state, the set of objects that existed is a free variable and projections are stated targeting references outside this set. Although this pattern obliges the reader to remember hypotheses not present in the theorem definition, the theorems are more general and easier to apply.

Both potential mutability, and direct operational mutability only place an upper bound on the information flow between existing objects, and cannot describe what information flows will come to exist to new objects. Direct operational mutability can only grow by the allocated object, such with the case of all potential access approximating operations. Therefore, the direct operational mutability between new objects cannot escalate the existing direct operational mutability, when restricted to existing objects, because these sets are disjoint. At each step, the direct operational mutability restricted to the existing objects must remain constant. In the base case, the direct operational mutability is identical to the potential mutability. Finally, by transitivity, what is initially potentially mutable is an upper bound on all mutation between existing objects.

This result enables analysis of a system state to statically determine an upper bound on the future mutability of all subsystems present by simply examining the initial potential mutability. Not only is potential access relevant to solving the safety problem, but it also directly approximates potential mutation through the *mutable* judgment. Both definitions are computable and decidable leading to a fully computable and decidable result that can be applied to reason about security policies.

The ability to successfully implement a security policy is dependent on the ability to model how changes in behavior impact the potential for information flow within a system. The definition of *mutable* is simple and follows directly from permission-based reasoning, yet its results form a persistent upper-bound on mutation through the life of the system. By examining how restricting capabilities can restrict potential mutability, it is possible to model the behavior of application-based security policies such as confinement.

Chapter 9

Confinement

This chapter presents the confinement proof in SDM. It begins with a review of the confinement test in capability-based systems and then translates these concepts for an embedding into SDM as a post-condition. Next, it introduces a concept of *fully authorized access graph* to define what is authorized by a set of capabilities in relation to the confinement post-condition. It presents the proof that any subsystem satisfying the confinement post-condition of SDM must be initially confined. It concludes by arguing that, from previous results, confinement must persist through the life of the system and can be used to verify constructor implementations in the future.

9.1 Review

The purpose of the confinement test is to allow applications to restrict the potential information flow of subsystems they construct or are constructed on their behalf. The test is usually performed by a *constructor* on behalf of an application, most likely before requesting the constructor to yield one or more new subsystems. An application trusts the constructor to execute the test faithfully and to produce a subsystem according to specification. Relying on a mutually trusted constructor permits mutually suspicious applications to interact without being required to engage in further trust.

As a constructive test, the confinement test may be performed by any application in the system via the mechanism they use to initialize new subsystems. The confinement test is parameterized over a set of capabilities that authorize all outward information flow. When the confinement test is successful, all outward information flow is authorized by a capability in the authorized set, and no information flow may occur if this set is empty. Viewed from the perspective of a constructor with a subsystem image definition, the confinement test requires all capabilities in the subsystem image to be 1) in the authorized set, 2) trivially non-mutating, 3) weak-only capabilities, or 4) name a constructor confined under the same authorized set. If this is the case, then the resulting subsystem is confined.

Figure 9.1 Definition of confinement.

```
Definition authorized_confined_subsystem C E S :=
  novel_capabilities C E ∧
  extant_capabilities C S ∧
  Sub.confined_subsystem C E S.
Definition confined_subsystem C E S :=
  RefSet.For_all (fun e ⇒ extant_test S e ∧
                          constructive_test E S e ∧
                          confinement_test C S E e) E.
```

9.2 Embedding Confinement

Confinement as embedded into SDM contains a few alterations to the motivating use case. SDM does not contain any notion of a constructor. Capabilities naming constructors appear as very permissive send capabilities. Without a trusted constructor implementation, SDM assumes they could misbehave as any other entity. Modeling and verifying the programs implementing a constructor is outside the scope of this proof. Rather than model confinement as a pre-condition and subsequent guarantee, SDM embeds the confinement test as a post-condition on the yield of the constructor within the system. This confinement test may be applied to any system state, subsystem, and authorized set, making it a more widely applicable result. However, this definition requires greater care when describing the nature of the confinement test and meaning of the confinement lemma, as there are many system states unreachable by the constructor pattern which had been implicitly pruned.

The definition of confinement is given by *authorized_confined_subsystem* and begins by restricting the authorized set of capabilities as part of analysis. During the

Figure 9.2 Extant test.

`Definition` $extant\_test\ S\ e := SC.is\_alive\ e\ S \lor SC.is\_dead\ e\ S.$

constructor pattern, an application asks for a confinement test before requesting the yield of a constructor. In any system, all capabilities must name *alive* or *dead* or objects according to *extant_capabilities*. As the subsystem has not yet been allocated, it is impossible for any capabilities to meaningfully name any element of the new subsystem. Therefore, the capabilities forming the authorized set are not permitted to target elements of the subsystem being confined and this is captured by the *novel_capabilities* judgment. When comparing subsystems composed of different objects, the meaning of the authorized set must be constant over all fresh subsystems. While this could be accomplished by reasoning about object state transitions, it does not translate well to access graph reasoning. Requiring all authorized capabilities to be external ensures that their meaning is preserved across access graph instances, provided the rest of the system remains identical.

An *extant subsystem* is one which consists entirely of alive or dead objects. It is impossible for any application to produce a subsystem containing *unborn* objects. Further, admitting a query about a subsystem containing any unborn objects is almost entirely nonsensical as their parent, and relationship in the system, is currently undefined. The confinement test requires an extant subsystem.

In addition to building a confined subsystem upon request, the constructor also has obligations to its yield. The constructor guarantees that no external capabilities

Figure 9.3 Constructive test.

```
Definition constructive_test E S e :=
  ∀ o, ¬ RefSet.In o E → SC.is_alive o S →
  ∀ i, option_map1 (fun cap ⇒ ¬ Ref.eq e (Cap.target cap))
    True (SC.getCap i o S).
```

Figure 9.4 Confinement test.

```
Definition confinement_pred C S E (cap:Cap.t) :=
  CapSet.In cap C ∨
  RefSet.In (Cap.target cap) E ∨
  ARSet.Empty (Cap.rights cap) ∨
  ¬ SC.is_alive (Cap.target cap) S ∨
  (ARSet.eq (Cap.rights cap) (ARSet.singleton wk)
    ∧ ¬ RefSet.In (Cap.target cap) E).
  Definition confinement_test C S E e:=
    ∀ i, option_map1 (confinement_pred C S E) True (SC.getCap i e S).
```

naming the yield exist when the yield starts executing, implemented by the careful deletion of capabilities. The parent cannot even know if its request was successful until the yield invokes the return capability[1]. At this point, no external influence can alter the nature of the yield without its prior consent. We call such subsystems *constructive*, and while the proof of these concepts is beyond the scope of this document, the structural property is an essential part of confinement. Were any such capability permitted to exist, it would fully undermine the result of the confinement test, as the holder of such a capability could wield it to modify the information flow of the new subsystem.

The confinement test is defined by *confinement_test* in Figure 9.4. Each element of a confined subsystem may only contain one of the following capabilities: 1) authorized

[1]Constructors do not abide by the standard call-return pattern. See Section 2.7

capabilities 2) trivially non-mutating capabilities 3) weak capabilities naming external objects 4) capabilities naming an internal object. This test differs from the canonical confinement test in two fundamental ways. First, it permits any internal structure to exist with any internal access. Intuitively, the confinement test only examines the constructed subsystem at its perimeter, and therefore is not concerned with how the subsystem is constructed internally. Further, ignoring these capabilities would require the subsystem to consist of fully disjoint objects, a property that will not be preserved through analysis of the confinement property. It is therefore critical that these capabilities are included as part of the confinement post-condition.

The case admitting a recursive constructor capability is missing from the SDM confinement test. As previously mentioned, SDM does not have an internal notion of constructors nor does it trust them to behave correctly. Therefore, the goal is to describe the pattern of confinement while seeking to have as few assumptions as possible. The recursively confined constructor is a structural guarantee that subsequently constructed subsystems are also confined. When verifying implementations are faithful in future efforts, the fixpoint of the constructor operation over the present confinement lemma should satisfy the inductive requirement to verify the constructor behavior.

Figure 9.5 Fully authorized subsystems

```
Definition exists_cap_edge tgt rgt C:=
  (CapSet.Exists (cap_edge tgt rgt) C).
Definition ag_remainder I E :=
  AG.filter (fun edge ⇒ true_bool_of_sumbool (excluded_edge_dec E edge)) I.
Definition ag_authorized_src C src acc := CapSet.fold (ag_add_cap src) C acc.
Definition ag_authorized E C := RefSet.fold (ag_authorized_src C) E AG.empty.
Definition ag_fully_authorized I E C :=
  AG.union
  (AG.union (Seq.complete_ag E) (ag_authorized E C))
  (ag_remainder I E).
```

9.3 Fully Authorized Subsystems

To verify that no subsystem satisfying the confinement test can ever exceed the mutability provided by the authorized set, SDM must first provide a specification for precisely what is authorized by a set of capabilities. Such a judgment must consider all possible subsystem configurations authorized by this set of capabilities. Rather than quantifying over all fully authorized subsystems directly, the definition uses the system's direct access graph to generalize the problem. The *fully authorized access graph* is composed by the union of three disjoint access graphs. It contains the complete access graph of the confined subsystem to encompass any possible internal structure. Additionally, each object in the subsystem is given all edges defined by the capabilities in the authorized set. Finally, all edges not mentioning elements in the confined subsystem are preserved, also called the *access graph remainder*.

Note that the definition of *ag_authorized* does not inspect any system state for object liveness. This should not be taken to mean that discussing access graphs of nonliving objects is pertinent. Capabilities within a subsystem passing the confinement test that name *dead* objects will be pruned in the direct access graph and are not analyzed here. However, because the fully authorized access graph is an upper bound on authority, their inclusion does not impact the confinement result. This is covered in greater detail as future work in Section 11.2.3.

9.4 The Confinement Proof

To validate the confinement test, SDM demonstrates that a subsystem passing the confinement test will remain confined for the life of the system. Chapter 8 verified that permissions and mutation are attenuating over the life of the system; all that remains is to demonstrate that the confinement test produces an *initially* confined subsystem. This proof proceeds in two phases. The first proof phase demonstrates that mutability of any subsystem passing the confinement test has a subset of the mutability of the *fully authorized subsystem* of the same *shape*. The *shape* of a subsystem is the set of object references of which it is composed. The second stage of this proof generalizes this specific result to include all subsystem shapes composed of free references.

Figure 9.6 illustrates the relationships used to describe how the first phase of this proof is verified. S is the System State in which the subsystem named by set E passes the confinement test over authorized set C. The top row contains the familiar direct access and potential access relationships. D is the direct access graph of S,

Figure 9.6 Visualization of the confinement lemma

Given (*ag_fully_authorized_spec I C E A*) and (*confined_subsystem S C E*),

$$\begin{tikzcd}
S \arrow[r, "dirAcc"] & D \arrow[rrr, "potAcc"] \arrow[dr, dash, "\subseteq"'] & & & Pd \arrow[rr, "(mutable\ E)"] \arrow[dr, dash, "\subseteq"'] & & Md \arrow[dr, dash, "\subseteq"'] & \\
& & I \arrow[r, "potTransfer"] \arrow[d, dash, "(ag\_simply\_confined\ E)"'] & I' \arrow[rr, "potAcc"] \arrow[d, dash, "(ag\_confined\ E)"'] & & Pi \arrow[rr, "(mutable\ E)"] & & Mi \arrow[d, dash, "="'] \\
& & A \arrow[r, "potAcc"] & Pa \arrow[rrrr, "(mutable\ E)"] \arrow[urr, dash, "(ag\_confined\ E)"] & & & & Ma
\end{tikzcd}$$

Pd the Potential Access of *D*, and *Md* is what is mutable by *E* in *Pd*. The bottom row contains the same relationships without an initial system state. *A* is the fully authorized access graph, *Pa* is its potential access, and *Ma* is the similarly computed mutable set.

The fully authorized access graph is defined in terms of the direct access graph of the system state of subsystem *E*. This preserves all relationships defined in the system state which are external to subsystem *E*. Therefore, no access edges mentioning subsystem *E* are used in the construction of the fully authorized access graph, as they are discarded by the remainder function. This will be crucial in second phase of the confinement proof where subsystem *E* can vary.

The middle row relates these sets of access graphs and their mutability. All of the relations in this row of the diagram have the same structure defined by *subset_eq_pred*. For access graphs *A* and *B* and proposition *P*, (*subset_eq_pred P A B*) requires *A* to be a subset of *B* and *B* to be a subset of *A* for all elements satisfying (*P B A*).

Figure 9.7 Simple confinement for access graphs.

```
Definition subset_eq_ag_simply_confined E A B :=
  subset_eq_pred (fun _ _ ⇒ ag_simply_confined E) A B.
Definition ag_simply_confined E x :=
  ¬ (Ref.eq (Edges.source x) (Edges.target x)) ∧
  ¬ (RefSet.In (Edges.source x) E ∧
    ¬ RefSet.In (Edges.target x) E ∧
    AccessRight.eq (Edges.right x) wk).
Definition subset_eq_pred P A B :=
  AG.Subset A B ∧ subset_pred (fun B A ⇒ P A B) B A.
Definition subset_pred P A B := ∀ x, P A B x → AG.In x A → AG.In x B.
```

Figure 9.8 Confinement for access graphs.

```
Definition subset_eq_ag_confined E A B:= subset_eq_pred (fun P _ ⇒ (ag_confined
E P)) A B.
Definition ag_confined E P x :=
  ¬ (Ref.eq (Edges.source x) (Edges.target x)) ∧
  ¬ (AccessRight.eq (Edges.right x) wk ∧
    ( ag_ex_flow E P (Edges.source x) ∨ ¬ ag_ex_flow E P (Edges.target x))).
Definition ag_ex_flow E A o := RefSet.Exists ((fun e ⇒ ag_flow A e o)) E.
Inductive ag_flow P a b : Prop :=
  | ag_flow_refl : Ref.eq a b → ag_flow P a b
  | ag_flow_tx : AG.In (Edges.mkEdge a b tx) P → ag_flow P a b
  | ag_flow_wr : AG.In (Edges.mkEdge a b wr) P → ag_flow P a b
  | ag_flow_wk : AG.In (Edges.mkEdge b a wk) P → ag_flow P a b
  | ag_flow_rd : AG.In (Edges.mkEdge b a rd) P → ag_flow P a b.
Theorem subset_eq_ag_simply_confined_ag_confined : ∀ E A B,
  subset_eq_ag_simply_confined E A B → subset_eq_ag_confined E A B.
```

Figure 9.9 Confined access graphs have the same mutability.

```
Theorem subset_eq_ag_confined_mutable:
  ∀ P P' E, subset_eq_ag_confined E P' P →
  ∀ M, mutable_spec P E M →
  ∀ M', mutable_spec P' E M' →
    RefSet.Equal M M'.
```

This latter relationship is defined by *subset_pred*. The two predicates used are *simple confinement for access graphs* and *confinement for access graphs*. *Simple confinement for access graphs*, defined by *ag_simply_confined*, admits the additional edges in the confinement test that are not in the authorized set of capabilities. It selects all edges that do not have identical source and target and are not external weak edges, allowing these edges to exist in the middle row of Figure 9.6. The predicate *ag_confined* defines *confinement for access graphs*. This relation is similar to *ag_simply_confined* but examines an access graph, presumed to be maximal, that will be used to query potential information flow. It relies on *ag_ex_flow* to provide a point-wise existence test on the mutability of subsystem E in access graph P, and then extend simple confinement for access graphs for full confinement by altering the external weak case to be weak flows into the subsystem. Simple confinement for access graphs is used as the initial relation when no potential access graph is available to query while confinement for access graphs is used to compare the remaining intermediate access graphs.

Given these properties, it is easiest to work from right to left. Figure 9.9 demonstrates that when subsystem E is confined to Pa in Pi, E has the same mutability in

Figure 9.10 Confinement is preserved from a maximal access graph.

```
Theorem subset_eq_ag_confined_potTransfer_max:
  ∀ B D, Seq.potTransfer B D →
  ∀ P, Seq.maxPotTransfer P →
  ∀ E, subset_eq_ag_confined E P B →
    subset_eq_ag_confined E P D.
```

Pi and *Pa*. This is proved by induction on the set difference of *Pi* and *Pa*. Consider the nature of any edge x in *Pi* that is not in *Pa*. Either the source and target of x are equivalent and cannot possibly alter the mutability predicate, or x admits a new weak edge. To satisfy confinement for access graphs, the source of x is mutable by E in *Pa* but the target is not; information in E is only authorized to flow to the source of x and not its target. Since a weak edge only allows allows information to flow *in the other direction*, from the target of x to its source, it must be the case that the target is also not mutable by E in *Pi*.

The proof that a subsystem remains confined over any maximal access graph for all access graphs reachable via potential transfer is shown in Figure 9.10. It is verified by induction over the potential transfer relation. Confinement for access graphs admits cyclic edges, or weak edges with a mutable source and a non-mutable target. When all of these conditions hold for all edges except the one under examination by induction, it becomes impossible to add any other type of edge. Weak flows can only beget other weak flows. Having restricted them to appropriate targets, they are only allowed to expand flows into E, but never outward. Cyclic edges require a pre-existing edge to cause an external flow. But the only pre-existing edges are either impotently cyclic

Figure 9.11 Initial simple confinement produces potential confinement.

`Theorem` *dirAcc_confined* :
  ∀ *E D D'*, *subset_eq_ag_simply_confined E D D'* →
  ∀ *P*, *Seq.potAcc D P* →
  ∀ *P'*, *Seq.potAcc D' P'* →
  ∀ *M*, *mutable_spec P E M* →
  ∀ *M'*, *mutable_spec P' E M'* →
  RefSet.eq M M'.

Figure 9.12 Confinement for subsystems of the same shape.

`Theorem` *confined_subsystem_mutable*:
  ∀ *C E S*, *Sub.confined_subsystem C E S* →
  ∀ *D*, *dirAcc_spec S D* →
  ∀ *A*, *ag_fully_authorized_spec D E C A* →
  ∀ *P*, *Seq.potAcc D P* →
  ∀ *P'*, *Seq.potAcc A P'* →
  ∀ *M*, *mutable_spec P E M* →
  ∀ *M'*, *mutable_spec P' E M'* →
  RefSet.Subset M M'.

or are weak flows into E. All other edges must have been pre-existing.

Figure 9.11 verifies that given Pa and Pi as the potential access of A and I, respectively, if E is simply confined to A in I, then E is confined to Pa in Pi. By definition A is a subset of I, so the least upper bound of potential transfers will find the smallest access graph I' that is a superset of Pa and is reachable by potential transfers from I. Adding any the same access edges to both I and A preserves simple confinement for access graphs, and consequently preserves simple confinement for access graphs between Pa and I'. Because simple confinement for access graphs always satisfies confinement for access graphs, it must also be the case that E is confined to Pa in I'. All that remains is to choose an I.

The simple value used for I is $D \cup A$, where A is the fully authorized access graph.

Figure 9.13 Final confinement proof.

```
Theorem confined_subsystem_mutability_subset_any_fully_authroized_mutability:
  ∀ E, ¬ RefSet.Empty E →
  ∀ C S, authorized_confined_subsystem C E S →
  ∀ D, dirAcc_spec S D →
  ∀ Ex, obj_existed Ex S →
  ∀ E', ¬ RefSet.Empty E' →
  novel_capabilities C E' →
  RefSet.Empty (RefSet.inter E' (RefSet.diff Ex E)) →
  ∀ R, ag_remainder_spec D E R →
  ∀ A', ag_fully_authorized_spec R E' C A' →
  ∀ P, Seq.potAcc D P →
  ∀ P', Seq.potAcc A' P' →
  ∀ M, mutable_spec P E M →
  ∀ M', mutable_spec P' E' M' →
    RefSet.Subset (RefSet.diff M E) (RefSet.diff M' E').
```

This trivially fulfills the subset requirement and also satisfies simple confinement for access graphs as well. Any access edge that could pass the confinement test is either in the fully authorized access graph or is excluded from comparison by simple confinement for access graphs.

Placing these components together in Figure 9.12 demonstrates that a subsystem passing the confinement test has less mutability than is expressed by the fully authorized access graph for subsystems of the same shape. The mutability of a subsystem in a fully authorized access graph does not vary with the elements of that subsystem, provided the new subsystem consists only of non-extant objects or elements of the previous subsystem. Any two disjoint subsystems with novel elements have the same mutability in their fully authorized access graph, with respect to the rest of the system. Extending a subsystem with fresh object references preserves mutability between fully authorized access graph, also with respect to the rest of the system. By case analysis and transitivity, these results are combined for the final confinement proof in Figure 9.13 demonstrating that all confined subsystems of fresh elements have the same mutability.

This result validates the confinement test of the constructor pattern. When a subsystem is confined in SDM, the initial potential mutability of the yield must be confined by the authorized set. Because initial potential mutability is an upper bound on future mutability, the subsystem must be confined for the life of the system. To verify the correctness of a constructor implementation, it suffices to demonstrate that when the constructor declares a subsystem image confined, each yield will pass the confinement test in SDM. If no constructor capabilities are present in the subsystem image, this should be case analysis on the constructor precondition producing a confined yield. If constructor capabilities are present and recursively pass the constructor precondition, then the yield of this constructor produces a confined yield by induction. Therefore, a correctly implemented constructor produces confined subsystems

Chapter 10

Applications of SDM

This chapter discusses various applications of SDM. It begins by outlining some specific systems that satisfy SDM including KeyKOS, EROS, Coyotos, and seL4. Next, it approaches the general domains applicable to SDM and then how to evaluate them. Evaluation is broken into four parts: correspondence with the operational semantics, correspondence for constructors, issues arising from deletion, and working in other logics.

10.1 Systems Satisfying SDM

SDM models the behavior of many different capability-based systems. While specifically targeting the behavior of Coyotos, SDM is also applicable to its predecessors, KeyKOS and EROS, and the seL4 microkernel. This section discusses some of the specific systems to which SDM applies. It also discusses the potential for formally connecting the model to Coyotos and seL4.

10.1.1 KeyKOS and EROS

As previously mentioned, SDM can be applied to KeyKOS and EROS. The system states and atomic actions of KeyKOS and EROS can be modeled by the system state and operational semantics of SDM with little alteration. Some aspects of these systems can be modeled using slightly more permissive embeddings in SDM, but these will not impact the confinement proof.

Both KeyKOS and EROS maintain the separation of capabilities and data using a Harvard-style partition. Main memory is divided into *Nodes* to hold capabilities and *Pages* to hold regular data. Instead of using access rights, the permissions of both KeyKOS and EROS use access restrictions. Using the set difference from a maximally permissive set of access rights in SDM will capture these permissions.

KeyKOS and EROS are atomic-action microkernels and this facilitates correspondence with SDM. In an atomic-action kernel, a thread that has entered the kernel does not wait with resources held. Instead, every kernel operation either acquires resources necessary for completion or unwinds its transaction completely to permit other operations to succeed. SDM only models observable system state and permits internal state to be hidden by equivalence relations. Consequently, transactions that are rolled back do not alter the observable state of the system and do not need to be modeled. At the moment an operation succeeds, it is possible to examine the state of the system and determine a corresponding sequence of SDM operations, many with only a single operation. System actions involving multiple SDM operations include the address space traversal and MMU updates.

Address spaces in KeyKOS and EROS are constructed hierarchically using Nodes for page directories. When processing an address fault, both systems walk these structures to update the hardware mapping structures. A traversal operation can not be modeled by a single operation in SDM, but can be modeled by a sequence of operations. The subject accessing memory can be modeled by simply performing the address walk of its own accord. This embedding considers a process to have sufficient access to its address space to perform the traversal, which may cause its access to appear escalated in SDM. However, this escalation occurs when mapping these permissions to SDM access rights and preserves safety and confinement.

Atomic actions in KeyKOS and EROS have the property that they are non-interfering. That is, two atomic actions executing simultaneously result in a state reachable by some serialization of actions. This serializability simplifies reasoning about these operations in SDM, as SDM does not contain support for concurrent execution.

10.1.2 Coyotos

Coyotos shares many similarities to its predecessors and applying SDM to Coyotos follows the same paradigm as with KeyKOS and EROS. One notable exception is how Coyotos partitions capabilities and data. The development of Coyotos includes considerations for embedding a high-level specification, and transcribing its executable software, into a proof assistant. As such, the potential to formally apply SDM to Coyotos also exists.

Unlike KeyKOS and EROS, Coyotos employs type-based separation of capabilities and data. Data-only *Pages* and Capability-only *CapPages* are mapped into a single process address space. This facilitates better software design and has a positive influence on cache locality. Some system objects contain both capabilities and data. Objects containing any capabilities are never mapped in such a way that an application may access or modify them without supervisor mediation. By mediating requests to these objects, Coyotos preserves a partition between data and capabilities.

One of the design goals of Coyotos was the possibility of transcription into a safe systems programming language. Once transcribed, this implementation could be embedded into a proof assistant and verified against a high-level specification of the Coyotos interface. Satisfying the operational semantics of SDM could then be achieved from such a high-level specification.

BitC is a functional, type and memory safe language with precise operational semantics for managing systems problems. [SDS08] The precision offered by BitC is intended to permit the language to be embedded for mechanical verification and to allow BitC compilers to assist software verification engines to discharge proofs about BitC programs. From early in its development, Coyotos has been developed to be transcribed into BitC to enable the construction and verification of a high-level specification. Such a high-level specification should correspond to the operational semantics of SDM.

Verifying the Coyotos Constructor should follow a similar approach to that used with EROS. The Coyotos constructor follows the same model as the EROS constructor in that it admits recursively confined constructors but does not parameterize confinement with an authorized set. The major issue impacting the recursive confinement case is to ensure constructors are authentic. Verifying this property may be difficult if cryptographic identifiers are used, as is the case in Coyotos, as it is not possible to totally prohibit collision. Provided attestation is established, the existing post-condition should serve as the foundation of an inductive step to verify a recursive constructor implementation.

10.1.3 seL4

The goal of the L4.verified [Kle10] project is to produce a trustworthy, mechanically verified L4 microkernel [Lie96]. The seL4 microkernel [EKK06] is the implementation modeled and inspected by L4.verified.

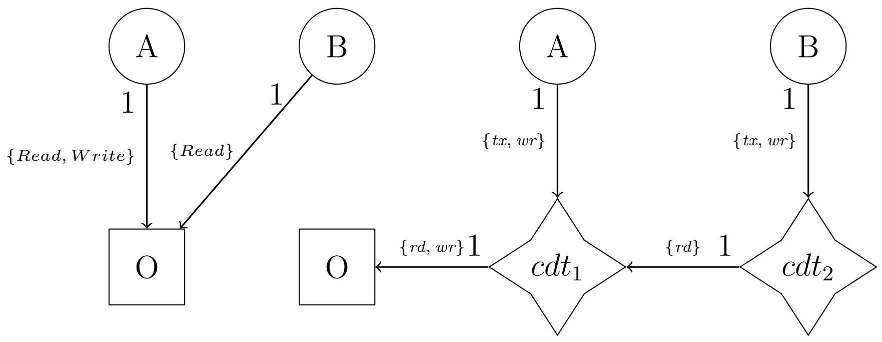
The underlying seL4 microkernel is a capability-based system. Capabilities in seL4 follow a strict partitioning similar to EROS and KeyKOS from Chapter 2. Not only are capabilities stored in protected structures, but they are named by a separate address space. The thread control block for each thread in seL4 contains separate capabilities naming a virtual data address space and a capability address space. In contrast, Coyotos *Pages* contain only data and *CapPages* contain only capabilities, though the two are intermingled in a single address space. Attempts to access *Cap-Pages* as data or *Pages* as capabilities will result in a fault. SDM does not distinguish these semantics and can be used to describe either structure.

The mechanics of memory management via capabilities in seL4 distinguishes it from the capability-based systems discussed in Chapter 2. All memory management interfaces are implemented by the seL4 kernel. [TST14] When seL4 starts, all unallocated memory is represented by a collection of *Untyped Memory* objects. Invoking a capability to an *Untyped Memory* object permits it to be be *retyped* into smaller *Untyped Memory* objects or into other kernel objects. Both the original untyped memory object and the new objects are valid after a *retype* operation along with their capabilities.

The kernel maintains a *capability derivation tree*, or CDT, to track the parent-child relationships. Reclaiming memory involves invoking a *revoke* operation on the original untyped memory object, invalidating all child capabilities. This structure is similar to hierarchical bank structure provided by the Space Bank domain. As a practical matter, the CDT is implemented as a linked-list structure represented

Figure 10.1 Embedding the CDT into SDM.

If the capability held by B is the child of the capability held by A:



within the capabilities themselves. This places an implementation-specific limit on the depth of the CDT, which is presently 128.

The seL4 kernel extends the CDT past the memory-management interface. Whenever a capability is copied, it can be placed in either a sibling relationship or in a child relationship with the original. The *copy* operation (sometimes called *imitate*) creates a sibling capability while the *mint* operation (sometimes called *grant*) creates a child capability. Whenever a capability is copied, it is possible to restrict the set of permissions on it. Child capabilities created in this way are recursively revoked in the same manner as untyped memory.

Modeling seL4 in SDM can be accomplished by modeling the nodes of the CDT as subjects with known behavior. In the present model, CDT nodes would appear as very permissive subjects connected by *tx* capabilities. To actually model seL4, SDM will need some refinement.

Refining SDM to faithfully model seL4 requires embedding a CDT node as a type of object. As discussed in Chapter 11, this can be performed by adding additional labels and permissions to the system state or by directly modeling *send* behavior. Because SDM embeds the memory manager into the model, the only portions of the CDT which must be modeled do not involve allocation or deletion. A CDT node is a proxy object and will respond to 3 different messages each requiring the *tx* permission: *move*, *mint*, and *invoke*. The *invoke* message requests that the CDT node invoke its internal capability in some manner. The *move* message requests that the CDT node return a capability to a new CDT node with its present internal state. The *mint* message requests that the CDT node create a new CDT node whose target is this CDT node, with possible restrictions.

These alterations to SDM capture the basic capability operations of seL4. All seL4 capabilities are represented by a capability to a CDT node. When a new object is allocated in seL4, it is allocated alongside a CDT node that directly points to it. The *move* and *mint* operations are performed as previously described. Note that the *wr* permission is used to describe the authority to delete the CDT node, but does not permit a store operation on a CDT object.

The constructor model remains largely unchanged. This test is slightly more involved as the constructor does not need to consider local CDT nodes as part of the confinement test, only the authority they confer. The ability to delete CDT nodes presently in the constructor will not be passed to the yield and their deletion only causes data to move into the yield excluding them from the constraints for confinement. With this more complex test, verifying a correct implementation will prove more difficult, but should be manageable.

10.2 Generally Applying SDM

This section presents the general domains where SDM is applicable and how to evaluate them. The evaluation discussion begins examining how to find correspondence between a system and the SDM operational semantics. Evaluating the confinement test of a constructor is presented next with an emphasis on the recursive confinement case. The third portion presents information flow issues arising from deletion and how to reason about them in SDM. This section concludes with how to evaluate SDM in other logics.

10.2.1 Applicable Domains

SDM can be applied to most protected capability-based operating systems. There are three critical requirements for satisfying SDM. Every system operation must be authorized by capability or modeled as though it were. No system operation may provide ambient authority; all authority must be capability-protected. There must not be any rights-amplifying operations that can not be simulated as a sequence of non-amplifying operations.

SDM incorporates specific features for reasoning about operating systems and their security-enforcing applications. The system model consists of small-step semantic operations intended to closely correspond with system-calls. They are presented simply in three parts: a pre-condition, a state transition, and an upper-bound on information flow. The use of high-order abstract syntax and monadic transformations has been purposely avoided in favor of clarity. The system model does not rely on a definition of confinement for correctness. Instead, confinement is a non-primitive property constructed from the semantics of the system. Finally, SDM provides indices to facilitate future proofs to accurately correspond model operations with the behavior of security-enforcing applications.

SDM can also model capability-based systems where part or all of the system protection mechanism is implemented in hardware. IBM's System/38 and AS/400 architectures were commercially available systems that supported capability-based addressing. [IBM81] The i432 processor implemented *access descriptors* (ADs) which simultaneously named a memory segment and contained permissions to control access. [Int83] Because no segment could be accessed without an access descriptor, ADs were hardware-implemented capabilities. The BiiN CPU architecture refined access descriptors as tagged pointers with permissions to system objects. [Bii88] The access control behavior of BiiN system objects could be defined via a hardware descriptor allowing the architecture to provide intra-application capabilities with fast function calls. The CHERI processor provides capability-based protection via fine-grain segment descriptors compatible with C language pointers. [WWN^{+}15] When coupled with a capability-aware compiler, CHERI minimizes the changes to application code that are necessary to implement a capability-aware application.

Architectures containing explicit support for capabilities have been used to produce operating systems that are not capability-based. The OS/400 operating system ran on the System/38 and AS/400, but does not provide capability-based protection to applications. [Sol96] Unix variants have also been ported and run on many of these processors and offer capability-based protection to varying degrees. The BiiN architecture supported its own Unix variant and CHERI includes a hybrid FreeBSD implementation that can compose traditional MMU-protected and capability-protected software libraries. Hardware support for capability-based systems is not sufficient to satisfy SDM. All actions performed by the operating system must correspond to the operational semantics of SDM. Therefore, SDM is only applicable when these architectures are supporting a pure capability-based operating system.

Applying SDM to language run-times with type and memory safety is also possible. In this arena, function closures and threads correspond to active objects in SDM while records and cells correspond to the passive objects. Capability representations vary in these systems and range from those similar to Coyotos, presented in Chapter 2, to memory references with safe type information. When using typed memory references, these capabilities correspond either to *rd*, *wr* capabilities for cells, or *tx* capabilities for closures.

A problem facing many type and memory safe language run-times is that they often encode ambient authority. Access to globally shared state remains typical in many otherwise safe languages. Reflection APIs are also detrimental because they can be used to escalate authority by accessing resources without a safe function pointer. These operations are difficult or impossible to model in SDM and eliminate confinement. Type and memory safe language run-times that do correspond to SDM include E [Mil06], Caja [caj], and W7 Scheme [Ree96].

10.2.2 Satisfying the Semantics

Correspondence with the operational semantics is the primary challenge when applying SDM to any system. For most systems, correspondence proofs with SDM will largely occur by demonstrating that the permissions of the examined system are subsumed by the access rights in SDM. Access rights in SDM are intended to assist with such subsumption proofs. The *rd* and *wr* access rights confer authority to read or write both data or capabilities. Taken together, these permissions confer universal authority over an object as *wk* access rights are a sub-type of *rd* and the mechanics of *tx* can be simulated. Systems with separate permissions or object types distinguishing capabilities and data operations can be fit to SDM by using the more powerful permissions, and consequently operations. The *tx* access right subsumes most known IPC and RPC mechanisms by optionally fabricating a reply capability and authorizing message transfers containing both capabilities and data.

When simple correspondence with access rights is insufficient to capture all system operations, the next strategy employed should be to find correspondence between system operations and a sequence of SDM operations. Developers are not obliged to demonstrate that all SDM operations can be produced by the examined system, but only that those sequences produced are subsumed by SDM operations. Because SDM assures any sequence of model operations preserves safety and confinement, correspondence with examined systems may interleave model operation sequences corresponding to system operations. This permits SDM to model long-running, concurrent operations composed of atomic actions that each correspond to a SDM operation. All such correspondence can be performed by predicating valid operation sequences for the examined system.

The safety and confinement theorems verify upper-bounds on permissions and information flow and remain so for systems satisfying SDM. Because *potAcc* analyzes a system using access rights instead of operations, the interpretation of potential access and safety change with how an examined system corresponds with SDM. For potential access to remain an upper-bound on future permissions, the permissions present in an examined system must confer less authority than the access rights of SDM. This is sufficient even when refinements subsumed by access rights are not recoverable.

10.2.3 Constructors and Recursive Confinement

Evaluating constructors for direct correspondence with SDM requires checking a modest set of properties. To preserve the integrity of all subsystem image tests before instantiating its yield, a constructor should not permit alterations to its subsystem image. Given this constraint, the constructor should implement the confinement test as a pre-condition that produces a yield satisfying the confinement test of SDM when affirmative. Ideally, verifying that the constructor correctly embeds the confinement test is simple iteration and case analysis over capabilities as described by the confinement test in SDM. The last constraint of a constructor is that it should delete the capabilities naming its yield after initialization. Whether these capabilities are deleted immediately after invoking its yield or lazily during the next yield request is irrelevant. A constructor with these properties successfully implements confinement as modeled by SDM.

The confinement test described in Chapter 2 admits recursively confined constructors as part of a confined subsystem image. The usefulness of this case is motivated by a few examples. A confined subsystem equipped with confined constructors can allocate and free different internal subsystems as needed and with further constrained authorized sets. Because the constructor also provides a trusted attestation mechanism, a confined subsystem may be required to provide its constructors to other subsystems. Without the recursive case, all confined subsystems would be required to reinstantiate internal subsystems without the assistance of constructors and attestation.

For a constructor to meaningfully query the confinement of a capability to another constructor, it must first establish that the capability *does* name another constructor. The constructor is also responsible for attesting its yield and all constructors are equipped with the system meta-constructor which yields and verifies constructors. A constructor performing the confinement test may safely rely on the result of a confinement test performed by an authentic constructor. Because constructor capabilities can not be placed into a subsystem image before it is sealed, these invocations are therefore free of cycles and the recursive confinement test is terminating. Verifying the recursive case requires verifying the attestation mechanism and checking the missing condition of the confinement test. As mentioned in Chapter 2, the KeyKOS Factory implements the confinement test with recursive constructors and an authorized set while EROS and Coyotos Constructors implement variant with recursive constructors and an empty authorized set.

The recursive constructor case is not directly implemented by SDM for a few reasons. One of the goals of SDM is to avoid embedding any notion of trusted subsystems in the model. However, embedding the recursive constructor case *requires* embedding which subsystems are trusted constructors. Establishing both the confinement and attestation interfaces for constructors and tracking their behavior sufficiently burdens an already complex model. Therefore, SDM eschews identifying constructors or managing subsystem behavior in favor of generalizing confinement as a post-condition on the yield. In future efforts, the confinement post-condition can be inductively invoked to describe how recursively confined subsystems evolve thereby handling the recursive case.

10.2.4 Destruction and Information Flow

Access-control meta-data is frequently not subject to a security policy and, consequently, may be used as a method for unconstrained information transfer. While many access-control mechanisms prevent unintentional data flow encoded in meta-data between *existing* domains, the creation (or allocation) and destruction of *new* domains is often overlooked. Applications with the authority to allocate or destroy a system resource effectively have the ability to transmit information to all applications capable of determining whether that resource exists. This section describes some of the problems and solutions to this issue and how they apply to SDM.

Section 2.2 discussed the problem of ambient authority in systems. The example therein illustrated how systems using access control lists provide ambient authority via the “owner” permission. A domain with the “owner” permission to an owned domain can create relationships between the owned domain and *any other domain,* even when the other domain has no pre-existing relationship to the owner. What the example does not describe is that many of these systems presume that the allocator of a new domain or system resource should have the “owner” permission by default. Coupled with the previous example, this behavior effectively grants total, system-wide authority to any domain with the ability to allocate a system resource.

A simple solution to prevent unforeseen information flow via allocation and destruction is to prohibit them entirely. Safety and information flow analysis become tractable in a static system with a fixed number of resources, especially when the "owner" permission is removed. While this strategy may work for fixed systems, it often fails when applied to general-purpose computing. All use-cases of a general-purpose system are not known in advance and these systems are therefore obligated to repurpose their resources for different tasks in different security contexts. Therefore, applying this strategy to SDM is not an interesting problem.

The solution presently supported by SDM is to prevent all applications from observing the state of other resources' existence. By requiring all operation preconditions to name alive objects, SDM provides no mechanism for witnessing the destruction of an object. The system must not generate any error messages and must produce valid responses for each system call. While this solution produces a reasonable theoretical approach, it is not practical for most systems.

Capability-based systems can safely generate error responses without violating the intended information flow constraints for their permissions. They accomplish this by ensuring that the information flow that may arise from observing the existence of an object is intentionally authorized by a capability. A simpler way to state this is that, absent specific permissions for creation, destruction, and observation, all information flow occurring from these acts could have occurred via some other operation that does not alter object existence. If this is the case, then leveraging a system error response is simply an inefficient encoding mechanism for approved transmissions.

For example, in systems where object existence is observable, consider the following three constraints. 1) No capabilities may name objects which have never existed, preventing their observation. 2) The ability to destroy an object is considered at least an outward information flow, or a “write,” to the destroyed object, though it may also authorize inward information flow, or “reading.” 3) Invoking capabilities authorizing only outward information flow, or “write-only” permission, must not be able to observe an error. Given these constraints, the system may safely generate error messages when a capability authorizing any inward information flow, or any “read” permission, is invoked. The only way for data to flow through destruction is to “write” a half-bit by destroying the object and then “read” it via error message later. Because the capabilities for doing so already authorize information flow in these directions, the destruction and error receipt are authorized.

This structure is more difficult to satisfy than it seems. The seL4 take-grant models [Elk10] [Boy09] do not require capabilities with any access rights to remove capabilities from other objects or destroy other objects. Consequently, these models only constrain information flow occurring via *SysRead* and *SysWrite* operations, analogous to the *read* and *write* operations in SDM. However, they do not constrain information flow using invokable capabilities as a transmission mechanism. Because object deletion was axiomatized very generally, the model places no upper bound on which objects are touched when an object is destroyed. While seL4 is not itself this unconstrained, the model admits the possibility that a deletion writes information to every object in the system. Section 13.4.1 covers this problem in more detail.

Chapter 11 proposes an update to SDM to align more closely with systems which fit the aforementioned pattern. The SDM operational semantics already capture the three constraints; SDM is only missing the ability for subjects to observe object state. The proposal adds an operation to explicitly observe object existence, *isAlive*, which requires the *rd* access right to perform. Exceptions can then be handled by electing to perform the *isAlive* operation, and all information flow is already authorized by the *rd* access right. Because SDM satisfies the necessary constraints and the information flow present in the *isAlive* operation is already present in the system, adding this operation to SDM will not impact the confinement result.

Systems like Coyotos do not have "write-only" capabilities; a capability authorizing "writing" it must also authorize some form of "reading." Because object destruction is a half-bit of information that occurs only once, error messages for invalid capabilities convey no additional information. Therefore, Coyotos may safely deliver error messages for all capability invocations. Simpler capability-based systems and language run-times do not distinguish between read-write and read-only capabilities, trivially satisfying this pattern.

10.2.5 Alternative Logics

SDM is constructed in Coq, a higher-order intuitionistic logic, to be as strong and widely applicable as possible. Intuitionistic logics are founded upon fewer axioms and consequently have stronger proofs than classical logics. Therefore, statements in higher-order intuitionistic logic are also statements in higher-order classical predicate logic. For example, re-stating and satisfying SDM in Isabelle/HOL [NWP02] should be sufficient to justify safety and confinement. However, to the author's knowledge, no automated tool exists to translate theorems or proofs between systems. Unlike Coq, some intuitionistic logics do not support impredicative propositions, making general translation difficult. Fortunately, SDM includes equivalence relations between all predicative boolean decision procedures and their impredicative definitions providing proofs that equivalent predicative theorems exist. Whether this constitutes sufficient information to automatically construct these proofs in an impredicative logic is a separate problem.

SDM is applicable to more systems than are presented here. Although this chapter gives specific consideration to KeyKOS, EROS, Coyotos, and seL4, it is intended to cover diverse capability-based systems and language run-times with type and memory safety. Applying SDM to these systems involves finding a correspondence for their operational semantics and their constructors while avoiding pitfalls regarding information flow via deletion. SDM is also built generally in an intuitive logic to be applicable in other proof assistants.

Chapter 11

Future Work

This chapter presents some opportunities for improvements and extensions to SDM. Performance has been a relevant issue throughout this effort and, although it is not relevant to confidence and therefore is not presently addressed, there are some enhancements that must be performed before SDM may be realistically extended. Like any software project, there are also some areas where the proof is poorly structured in ways which may undermine confidence. Although they do not impact the general result, they should be refactored to make theorems more readily understood. The last section discusses some of the extensions SDM is eventually intended to support.

11.1 Improve Performance

The SDM proof presently takes over a week to compile on a single-threaded processor and uses over 12 GB of RAM. The Make environment permits up to three jobs to run simultaneously, each requiring a core and 12 GB of RAM, allowing a machine with over 42 GB of RAM to compile much faster. As discussed in Section 4.3, SDM depends on the *FSet* libraries and therefore uses module functors and signatures as an abstraction mechanism. Unfortunately, as these module signatures grow in complexity, the obligations of the type checker grow exponentially. Any future effort will need to address this issue before extending the model. This section focuses on two improvements that will help with this problem.

11.1.1 From Modules to Typeclasses

The use of the Coq module system has the greatest impact on compilation performance in SDM. Modules in Coq are generative and use subtyping for abstraction. In generative module systems, two identical applications of a module functor to a module produce two distinct modules. This means that every module or signature must be unique and it appears that every module parameter must be checked independently of all other parameters. Using the pure functor paradigm causes an exponential overhead on early modules. While checking type signatures should be a terminating procedure for Coq modules, various single module instances of SDM were terminated after a week of run-time with a working-set over 80 GB of memory. Type-checking modules is a non-interactive process, which compounds the problem as the developer is unable to assist the verifier in constructing a solution.

Typeclasses offer many of the abstraction features provided by the module system, but do so by leveraging the core term language and automation features. Consequently, much of the Coq standard library is migrating away from modules in favor of typeclasses. A typeclass is conceptually a record type used as a signature, and instances of a typeclass are records satisfying that signature precisely. The remainder of naming conventions and type search proceed through a specialized typeclass "binder" look-up table and the automated hint database.

The primary benefit of typeclasses is that they put the developer back in control of how proof requirements are discharged. All proof obligations for a class or instance that are not solved by the automation system are presented to the developer. If a typeclass is not in scope, or multiple typeclass binders of the same name are in scope, the developer may manually specify which should be used in any context. Because the Gallina is applicative and does not admit subtyping, typeclasses avoid the exponential overhead in the module system. A typeclass instance provides precisely the proof satisfying an abstraction boundary, but does not lose the underlying structure as is the case with module subtypes. Therefore, the first step to any future effort using SDM is to migrate modules to type classes.

Unfortunately, migrating SDM to typeclasses will require refactoring the *FSet* and *FMap* libraries, along with other dependencies. These modules are among the largest productions in the COQ standard library and it is unclear how much effort this task will be. Although the *MSet* libraries are a somewhat modern update to the *FSet* libraries, the *FMap* structures have are not included and neither suite has been updated to use typeclasses.

11.1.2 Proof Enhancements

Due to the high cost of recompiling SDM, many general theorems about *FSet*s or system states appear in later modules where they are first used. Many of these theorems can replace previous theorems in the SDM definition and support libraries, and this could improve efficiency by reducing the total number of theorems to compile. Some of these are over-specialized for access graphs or other structures and could also improve efficiency and readability if refactored. While all of this effort would improve efficiency and increase confidence, it should not be attempted until after SDM has been converted to use typeclasses.

There are two specific alterations in this regard that would have a major impact on efficiency and confidence. First, the *ag_potTransfer_fn_req* definition should be altered by eliminating the *ag_nondecr* requirement, as it can be derived from the definition of *ag_add_commute* and *ag_equiv*. Second, the theorems describing *endow* should use *AG_project* for uniformity and *mutable* should be rephrased in terms of *ag_flow* and *ag_ex_flow*. This will eliminate a substantial amount of what is now duplicated verification effort by unifying a number of concepts that were not simultaneously envisioned.

11.2 Improve Confidence

SDM contains a number of issues that obfuscate the problem statement. While not exactly flaws in the model, adjusting these definitions to align with developer intuitions would improve confidence in the final result. These alterations are therefore highly recommended as future work.

11.2.1 Minor Complexities

The definition of *copyCapList* currently folds *copyCap* over the list of index pairs. This design decision was made because it is an easy induction to describe and follows the pattern of non-allocating systems. However, this pattern undermines confidence in the interpretation of an index map when an object updates itself. When an index appears as a destination in the list and later appears as a source, the intended meaning of such a map may not have been what was intended. In this case, the first invocation of *copyCap* will overwrite the destination location, causing this capability to be copied in subsequent invocations. While such lists can always be refactored into logically equivalent ones using temporary storage in the same object, requiring systems which provide intermediate allocations to frame their operations in this way is counter-productive. The *copyCapList* operation should preserve the map intuition by allocating temporary storage for all capabilities, transferring them to temporary storage, and then copying them to the destination. The *copyCap* function should then invoke *copyCapList*with a single element.

SDM does not contain a no-op operation, implicitly assuming that all active objects are self-mutating. The model captures this in the definitions of *mutated* and *mutable*. However, in the semantics, the definitions of *readFrom* and *wroteTo* do not capture this intuition. The *readFrom* judgment declares that all operations read from the invoking object in addition to other sources, but this is not the case for *wroteTo*. This undermines confidence as it does not appear to capture all flow during an operation, and therefore should be included as part of the *wroteTo* specification. All new information flows will evaporate during when examining *mutated*, leaving the remainder of the proof largely untouched.

In retrospect, the reflexive cases of *transfer* introduced more problems than they solved. Object references and access edges can spring into being justified by the underlying system state rather than by examining an access graph. This required additional predicates to be carried along with an access graph during every stage of the proof. One possible solution is to alter the direct access graph to add all reflexive edges for objects that are alive. This would cause the definition of *ag_objs_spec* to be identical to the live objects and eliminate many corner-cases during their analysis. This alteration may make *potTransfer* more amenable to analysis in future efforts that specialize the access graph structure to distinguish active and passive objects as not all objects should be considered to have total self-authority.

11.2.2 Flow for System Error Handling

When the invocation of a capability cannot be performed by the system, most capability-based systems offer some degree of error reporting. For example, attempting to invoke a capability identifying a destroyed object could result in an error message reply fabricated by the system. These errors are different from those returned by the recipient as they potentially reveal information about the recipient without its consent. Great care is needed when handling system error-reporting as it is possible to unintentionally introduce ambient authority. Therefore, SDM does not presently take a stance on the issue and does not permit any information flow due to error reporting.

Fitting existing capability-based systems with system-initiated error reporting into SDM can be done without modification. In systems where all invocations are synchronous, like EROS, the destruction of an object can be modeled as though the object remains alive but behaves according to the specification of the *void object*. As the behavior of the void object is specified by the system, the underlying object storage may be safely freed. In systems permitting asynchronous invocations, there must be a distinction between invoking a capability and invoking an object. This can be conceptually handled by modeling each system object with two objects in SDM: a subject which may potentially respond with an error and the object with the intended behavior.

Both of these strategies are cumbersome to envision when the capabilities present *already* capture the information flow that occurs during an error response. Altering the state of an object requires the *wr* access right, which permits information to flow from the invoking subject to the target. Any information that could be encoded by destroying the object could have been written directly. Similarly, a capability with *rd* authority permits information to flow from the target object to the invoking subject, allowing the object state to be read along with anything else. SDM can be extended to directly incorporate error reporting for capability invocations by permitting the object label to encode data.

Updating the object label to be accessible data can be accomplished with two changes. The first alteration modifies *wroteTo* for the *destroy* operation to include the destroyed object. This alteration adds no new authority because the *destroy* operation requires the *wr* permission, so the half-bit of data could just as easily have been written using the *write* operation. The second update to the model adds a new operation: *isAlive*. The *isAlive* operation prerequisite requires a *wk* or *rd* capability, but examines only *preReqCommon* removing the need for the target to be alive. The *readFrom* and *wroteTo* values for this operation are identical to *read*, and therefore the half-bit result is also authorized. Any error handling that might occur can be modeled as the system performing this operation instead of the error-free operations.

This extension to SDM has significant impact on the interpretation of access rights in systems that support error-reporting for asynchronous invocations. The only access right permitted to notice an error is *rd*; the *wr* permission does not authorize error reporting by itself. In systems where operations requiring *wr* may receive errors, it is necessary that the *wr* always entails *rd* to permit this operation. When this is the case, all errors can be modeled using the *isAlive* operation.

11.2.3 Improve Fully Authorized Access Graphs

The present definition of the fully authorized access graph does not eliminate nonsensical capabilities identifying dead objects. This is not a great concern because the confinement test may be posed without these capabilities producing a conceptually equivalent test. However, their inclusion erodes confidence in the proof as the theorem does not directly pattern-match with the constructor pattern.

Filtering the authorized set of capabilities before determining the fully authorized access graph will produce an identical access graph to the one formed by filtering these capabilities before the confinement test. Constructing this proof follows from case analysis. Each access edge is in one of three sets: the access graph remainder, the complete access graph of the subsystem, or the authorized access edges. Filtering these capabilities cannot impact the complete access graph of the subsystem, and no capability targeting a dead object can impact the direct access graph as it ignores them. Computing the filtered set of capabilities is the same whether it is specified before or after testing confinement, producing identical access graphs. Therefore, the mutability of an authorized set containing capabilities naming dead objects must be bounded by the fully authorized access graph that does not contain them.

11.3 Model Extensions

There are a number of extensions to SDM that would increase confidence in the result or facilitate future verification endeavors. Extending the model to permit object reclamation will permit the model to more closely resemble a real system implementation. While support for application verification exists, it can be enhanced by restructuring the system state. Finally, verifying the constructor mechanism directly as an implementation archetype would further increase confidence in the result.

11.3.1 Object Reclamation

The confinement proof does not rely on a finite number of unborn objects, which indicates that the finite system requirement could be relaxed to only include extant objects. However, as facilitating dead object reclamation is intended as future work, this change is not currently present. Recall that all unborn objects must have all capabilities naming them removed from the system state before they may be marked alive. Therefore, the only reason a dead object may not be safely marked unborn is because the name of the object no longer holds consistent meaning over the life of the system. Consequentially, enriching the structure of object references or capabilities to admit structural reclamation without logical reclamation will alleviate this constraint and admit object reclamation. Therefore, with a sufficiently rich naming structure, an unbounded number of objects can be managed from a finite system state in future endeavors.

11.3.2 Application Verification

Presently, SDM provides support for verifying future application behavior through the index structure. When embedding the behavior of applications, theorems may use indices to identify precisely which capabilities are invoked. In many capability-based systems, the ability to inspect the physical representation of a capability is not universally available to applications. As such, applications cannot inspect their capabilities to determine behavior and can only distinguish capabilities by index. By identifying which objects have known behavior, the ability to quantify the impact of invoking capabilities as part of an application configuration becomes possible.

Restructuring the system state will assist future verification of application behavior. Presently, the system state is a map onto an ordered tuple containing all object information. This arrangement is sub-optimal as object meta-data is not easily carried alongside an access graph. Splitting the system state into two maps, the object map and the meta-data map, would simplify constraint management across access graph.

This creates other avenues for application verification. The definition of *transfer* could be refined to distinguish between active and passive objects. While not relevant for confinement, the subject/object distinction can have significant impact on security policies that only rely on the inability of passive objects to invoke capabilities. The schedule structure is a placeholder originally intended to permit suspending and resuming processes to facilitate scheduler policies. Objects could be further refined to distingusih those that only hold data, from those that also may hold capabilities. Other behaviors in the system or for applications may be carried between both system states and access graphs with a partitioned map.

11.3.3 Recursive Constructor

The most obvious application verification to perform in the future is to model the constructor application in SDM. As a post-condition, the present confinement verification describes the obligation of any constructor in how it instantiates future subsystems. These constraints can be phrased inductively to describe how confined subsystems might instantiate other subsystems under the same obligations. Folding these obligations to build an inductive definition of confinement is the first step to verifying a constructor archetype.

The second part of verifying a constructor application model is to verify the confinement test as a precondition. There are two interesting parts of this verification that are readily identifiable. First, the constructive test will require the use of index management to ensure that the constructor destroys all capabilities naming its yield after calling into it. Second, the system will need to name those components of the system that comprise constructors so as to simulate their invocation via *tx* capabilities. Using the inductive definition of confinement, it should be possible to verify the recursive constructor case.

Chapter 12

The SW Model

SDM was initially conceived as a mechanical verification of the SW confinement proof [SW00] with some additional features. During the verification process, SDM diverged from SW in response to extensions, verification pressures, and discovered flaws. As SW informs much of SDM, this chapter presents a comparison of the two models and proposing corrections to SW based on the SDM proof.

This chapter begins by discussing some of the differences between SW and SDM and proposes some minor changes that are not crucial to the heart of the proof, but necessary for completeness. It then presents the two major flaws of SW identified by SDM: the base case of the main theorem and Lemma 4. It concludes with a proposal that fits the verified solution of SDM into SW to form a result that does not significantly alter the main theorem of SW.

Figure 12.1 SW Model : **create** operation.

If $S_n \xrightarrow{\mathbf{create}(p,a)} S_{n+1}$ and $a \subseteq {S_n}^{\mathbf{caps}}(p)$, then
let$o' \in \mathbf{Object} - {S_n}^{\mathbf{exist}} - {S_n}^{\mathbf{dead}}$ in

$$\begin{aligned} {S_{n+1}}^{\mathbf{exist}} &= {S_n}^{\mathbf{exist}} \cup \{o'\} \\ {S_{n+1}}^{\mathbf{caps}} &= {S_n}^{\mathbf{exist}}[o' \to a][p \to {S_n}^{\mathbf{caps}}(p) \cup \mathbf{ObCap}(o', \mathcal{R})] \end{aligned}$$

12.1 Differences Between Models

Although SDM and SW are very similar, they differ in a few critical ways. Their system states differ as SW represents the state of objects in disjoint sets instead of tagging each object with meta-data in the map. The sets are $S^{\mathbf{exist}}$ and $S^{\mathbf{dead}}$ with all remaining objects in S as being available, and correspond to *alive* and *unborn* object labels. The set of objects **Object** in SW is permitted to be infinite where it is presently finite in SDM, though this is addressed in Section 11.3.1. Additionally, SDM contains more object meta-data and uses an index structure in objects to refer to capabilities.

This chapter uses different font faces to represent concepts in each proof. Definitions from SW are written in **boldface**, while definitions from SDM are written in $regularface$. Unless otherwise specified, most definitions herein are referring to SW definitions or updated definitions from this chapter.

Most of the operations in SDM are identical to those in SW with the exception that they are described using indices instead of capabilities. The **create** operation has been renamed to *allocate*, the **invoke** operation has been renamed *send*, and the **exec**

Figure 12.2 SW Model : $\mathcal{R}_\perp$definition.

Let $\mathcal{R}_\perp$be the CPO where $\perp = \emptyset, \top = \mathcal{R}, \mathcal{R}_\perp = 2^{\mathcal{R}}$ and $\leq$ is defined by

$$x \subseteq y \Rightarrow x \leq y \; \forall x, y \; , \in \mathcal{R}$$

Figure 12.3 SW Model : **DirAcc** definition.

If $S \in \mathcal{S}$, then we define $\mathbf{DirAcc}_S : \mathbf{Object} \times \mathbf{Object} \rightarrow \mathcal{R}_\perp$ by

$$\mathbf{DirAcc}_S(x, y) = \mathbf{lub}(\{a | (x, y, a) \in \mathbf{DASet}\})$$

where

$$\begin{array}{ll} \mathbf{DASet} = & \\ \quad \{(o, \mathbf{target}(c), \mathbf{rights}(c)) & | o \in \mathbf{Object}, c \in S^{\mathbf{caps}}(o)\} \\ \cup \quad \{(\mathbf{target}(c), o, \top) & | o \in \mathbf{Object}, c \in S^{\mathbf{caps}}(o), \mathbf{exec} \in \mathbf{rights}(c)\} \end{array}$$

access right has been renamed to *tx* to avoid confusion in terminology. Additionally, the *send* operation includes an optional reply capability. The SW **create** operation will receive special attention in this chapter.

SDM does not define a complete partial order over access rights, but instead lifts this CPO to access graphs by subset inclusion and *transfer*. All of the relationships between access rights are captured in the *transfer* relation, similar to **transAccess** in SW The definition of $\mathcal{R}_\perp$ from SW is included in Figure 12.2 for reference.

The definition of *dirAcc* in SDM closely follows the definition of **DirAcc** in SW. The primary difference is that the **exec** case does not produce $\top$ authority in *dirAcc*. In SDM this is also managed as emergent behavior in the *transfer*.

Most of the differences between SDM and SW appear in how they structure potential access. In SW, **PotAcc** is defined directly in terms of **DirAcc** as the closure

Figure 12.4 SW Model : **PotAcc** definition.

If $S \in \mathcal{S}$, then the potential access relation, $\mathbf{PotAcc}_S$ is the limit of the series $T_0, T_1, T_2, \ldots$ where

$$T_0 = \mathbf{DirAcc}_S$$
$$\forall x, y \ , \ T_{i+1}(x, y) = \mathbf{lub}\{T_i(x, y), \mathbf{combine}(T_i)(x, y)\})$$

where

$$\mathbf{combine}(A)(x, z) = \mathbf{lub}(\{a | \exists y \in \mathbf{Object} \text{ such that } a = \mathbf{transAccess}(x, y, z)\})$$

and

$$\mathbf{transAccess}(x, y, z) = \begin{cases} \bot & \text{if } A(x, y) = \bot \\ A(y, z) & \text{if } \{\mathbf{rd}, \mathbf{exec}\} \cap A(x, y) \neq \emptyset\} \\ \{\mathbf{wk}\} & \text{if } \{\mathbf{wk}, \mathbf{rd}, \mathbf{exec}\} \cap A(x, y) = \{\mathbf{wk}\} \wedge \\ & \{\mathbf{wk}, \mathbf{rd}\} \cap A(y, z) \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$$

of **combine**: a very large combinator of the least upper bound of **transAccess**. SDM breaks these relations apart using access graphs as an intermediate structure. Therefore, the definition of *potAcc* is meaningful over many different access graphs, not only the direct access graph. This allows SDM to restructure **transAccess** into *transfer* as a micro-operation of access right transfer. This eases mechanical reasoning about potential access from any reachable access graph by allowing justifications in the closure to be simply reordered. Therefore, *transfer* subsumes the previously mentioned properties from **DirAcc** and $\mathcal{R}_\bot$.

Reflexivity in potential access is another major difference between SW and SDM. SW does not define **DirAcc** or **PotAcc** as reflexive closures, only transitive ones. SDM ensures that *potAcc* contains all reflexive access edges through *transfer* causing

Figure 12.5 SW Model : **mutable**.

If $S \in \mathcal{S}$, then

$$\mathbf{mutable}_S(E) = \{y | \exists x \in E, \{\mathbf{wr}, \mathbf{exec}\} \cap \mathbf{PotAcc}_S(x, y) \neq \emptyset\}$$

Figure 12.6 Correction to **mutable**.

If $S \in \mathcal{S}$, then

$$\mathbf{mutable}_S(E) = \begin{array}{ll} \{y | \exists x \in E, & \{\mathbf{wr}, \mathbf{exec}\} \cap \mathbf{PotAcc}_S(x, y) \neq \emptyset \vee \\ & \{\mathbf{rd}, \mathbf{wk}\} \cap \mathbf{PotAcc}_S(y, x) \neq \emptyset\} \end{array}$$

direct access to remain a simple translation. Reflexivity of access will become very important in the discussion below.

12.2 Minor Errata

There are a few minor errata in SW that, while their meaning may be inferred given context, must be corrected before addressing large changes. The four alterations are to **mutable** (resp. **readable**), $\mathcal{R}$, **transAccess**, and information flow via the **create** operation. Figure 12.5 presents **mutable** from SW, which is intended to capture all potential information flow. However, the **rd** and **wk** rights are omitted and **readable** is simply the inverse of **mutable**. Because **transAccess** in **PotAcc** propagates existing permissions, **rd** and **wk** access rights will never be considered. This is fixed by adding the missing condition to **mutable** using the definition in Figure 12.6. It is not necessary to alter **readable**, as the change to **mutable** corrects

Figure 12.7 Correction to SW Model : $\mathcal{R}_\perp$.

Let $\mathcal{R}_\perp$ be the CPO where $\perp = \emptyset, \top = \mathcal{R}, \mathcal{R}_\perp = 2^{\mathcal{R}}$ and $\leq$ is defined by

$$\forall x, y \in \mathcal{R}\ ,\ x \leq y = \begin{cases} x - \{\mathbf{wk}, \mathbf{rd}\} \subseteq y - \{\mathbf{wk}, \mathbf{rd}\} & \text{if } \mathbf{wk} \in x \wedge \mathbf{rd} \in y \\ x \subseteq y & \textit{otherwise} \end{cases}$$

Figure 12.8 Correction to **transAccess**.

$$\mathbf{transAccess}(x, y, z) = \begin{cases} A(y, z) & \text{if } \{\mathbf{rd}, \mathbf{exec}\} \cap A(x, y) \neq \emptyset\} \\ A(y, z) & \text{if } \{\mathbf{wr}, \mathbf{exec}\} \cap A(y, x) \neq \emptyset\} \\ \{\mathbf{wk}\} & \text{if } \{\mathbf{wk}, \mathbf{rd}, \mathbf{exec}\} \cap A(x, y) = \{\mathbf{wk}\} \wedge \\ & \{\mathbf{wk}, \mathbf{rd}\} \cap A(y, z) \neq \emptyset \\ \perp & \text{otherwise} \end{cases}$$

both definitions.

The **wk** permission is interpreted inconsistently in SW and is not always considered a positive permission. The **rd** permission conveys all of the authority that the **wk** permission does. Therefore, **rd** should be considered a proper subtype of **wk**. In practice, these two permissions do not appear together, however, the model does not exclude this possibility. By implementing this change in $\mathcal{R}_\perp$, it will trickle down to all other aspects of the proof including **lub** and **PotAcc** definitions.

In SW, the underlying transitive operation capturing potential permission transfers is defined by **transAccess**. Unfortunately, **transAccess** neglects a discussion of **wr** permissions being capable of transferring other permissions. The missing case should invert the transitivity rule handling the **rd** and **exec** case. The updated definition also omits first $\perp$ case, as it is fully subsumed by the last case.

The last minor correction involves the definition of information flow for the **create** operation. The **readfrom** and **wroteto** relations are used to define potential information flow for a given operation. It is important to note that the **create** operation may send capabilities from the parent to the child. *Capability flow is information flow* and it must be captured to avoid incorrect analysis. The only correction to the SW model is to alter the definition of **wroteto**(**create**(p, a)) to be $\{p, n\}$, where n is the newly created object.

12.3 Major Concerns

This section addresses two of the major issues in the SW verification. The first demonstrates that the main theorem's induction hypothesis is invalid in the base case and proposes a simple solution. The second illustrates a flaw in Lemma 4, required for the main theorem.

It is worth noting that the discovery of both flaws occurred as the direct outcome of interacting with the proof assistant. Attempts to verify similar theorems in SDM arrived at obviously erroneous goals which, with some careful examination, lead to the counterexamples presented below. Ultimately, these counterexamples informed changes in the structure of the proof execution leading to a nearly identical conclusion.

Figure 12.9 SW Model : Main Theorem.

$$\mathbf{mutated}_E(S_0 \xrightarrow{\alpha_1} S_1 \ldots \xrightarrow{\alpha_n} S_n) \cap {S_0}^{\mathbf{existed}} \subseteq \mathbf{mutable}_{S_0}(E)$$

Figure 12.10 SW Model : Induction Hypothesis.

Assume that for all sets F,

$$\mathbf{mutated}_F(S_0 \xrightarrow{\alpha_1} S_1 \ldots \xrightarrow{\alpha_{n-1}} S_{n-1}) \cap {S_0}^{\mathbf{existed}} \subseteq \mathbf{mutable}_{S_0}(F)$$

12.3.1 Invalid Induction Hypothesis

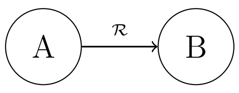
The main theorem SW attempts to prove the theorem in Figure 12.9 by induction on n. This relies on the induction hypothesis for n-1 as shown in Figure 12.10. SW claims that the base case is trivial, yet it is straightforward to violate. **mutated** is always growing the initial subsystem E, while **mutable** only considers what E can mutate via **PotAcc**. When no operations are performed, the base case reduces to the property of Figure 12.11: **mutable** is non-decreasing. However, Lemma 5 of the SW excludes the initial subsystem E from what is considered mutable, indicating that the base case does not hold.

To help visualize counterexamples, concrete instances are described visually. System states are represented as graphs in the same manner as SDM. As SW considers all objects active, objects are circles and capabilities are edges in the system state

Figure 12.11 Property: **mutable** is non-decreasing.

$$\forall E \ , \ E \subseteq \mathbf{mutable}_{S_0}(E)$$

Figure 12.12 Contradiction: System State S_0 and $\mathbf{PotAcc}_{S_0}$ which violates the induction hypothesis.



$$\begin{tikzcd} A \arrow[r, "\top"] & B \end{tikzcd}$$

$$\mathbf{mutated}_{\{A\}}(S_0) \cap \{A, B\} = \{A\} \not\subseteq \{B\} = \mathbf{mutable}_{S_0}(\{A\})$$

Figure 12.13 Correction to **DASet**.

$$\begin{array}{lll} \mathbf{DASet} = & & \\ & \{(o, \mathbf{target}(c), \mathbf{rights}(c)) & |o \in \mathbf{Object}, c \in S^{\mathbf{caps}}(o)\} \\ \cup & \{(\mathbf{target}(c), o, \top) & |o \in \mathbf{Object}, c \in S^{\mathbf{caps}}(o), \mathbf{exec} \in \mathbf{rights}(c)\} \\ \cup & \{(o, o, \top) & |o \in \mathbf{Object}\} \end{array}$$

diagram. As with SDM diagrams, capabilities are labeled with their access right set and the border and fill of objects indicate their label. **Dead** objects are gray, an object that **exists** has a solid border, and an **available** object has a dashed border. Access relations are also visually represented, but they consist only of relationships in $\mathcal{R}_\perp$ between objects which have no border.

The counterexample in Figure 12.12 illustrates that the present definition of **mutable** is not non-decreasing. When no operations are performed, $\mathbf{mutable}_E = E$. However, **DASet** does not capture this access, and therefore it must not be present in $\mathbf{DirAcc}_{S_0}$ or $\mathbf{PotAcc}_{S_0}$. Therefore, $\mathbf{mutable}_{S_0}(\{A\}) = \{B\}$ which is not a superset of $\{A\}$.

This problem can be avoided by altering the definition of **mutable** to conform to the property in Figure 12.11. There are two potential solutions: alter **mutable**

Figure 12.14 SW Model : Lemma 4

If $S_0 \overset{\alpha}{\rightarrow} S_1$, then for all E,

$$\mathbf{mutable}_{S_1}(E) \cap {S_0}^{\mathbf{existed}} \subseteq \mathbf{mutable}_{S_0}(E \cap {S_0}^{\mathbf{existed}})$$

to have this property directly, or alter the definition of **DASet** to cause **PotAcc** to be a transitive reflexive closure. In SDM, **mutable** was altered to be non-decreasing in addition to altering **PotAcc** to be a transitive reflexive closure without altering **DASet**. In hindsight, it is better to alter **DirAcc** to be reflexive, as in Figure 12.13, as this simplifies relations between access relations. For more information on this future work in SDM, see Section 11.2.1.

12.3.2 Violating Lemma 4

The main theorem also relies on Lemma 4, which also contains flaws that must be corrected to present a solution. Lemma 4 in the SW Model misquantifies the initial subsystem E and is consequentially too general to be correct. The subsystem E admits *any* subset of **Object**, including any *unborn* objects. This permits unborn objects that will be descended from subsystems *outside* of $E \cap S^{\mathbf{existed}}$. Because E is unrestricted in this way, it is possible to construct an example where elements of some initial E become children of an external subsystem, connecting E to a previously unconnected subsystem.

Figure 12.15 System State S_0 and $\mathbf{PotAcc}_{S_0}$ to violate Lemma 4.

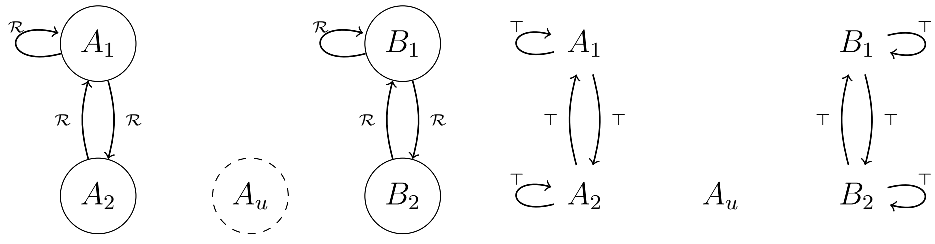
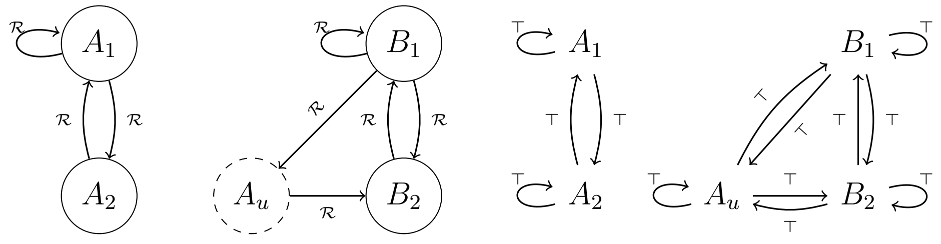


Figure 12.16 System State S_1 and $\mathbf{PotAcc}_{S_1}$ to violate Lemma 4.



Consider the result of a **create** operation on S_0 as in Figure 12.15. If the operation is:

$$S_0 \xrightarrow{\mathbf{create}(p,\{\mathbf{ObCap}(B_2,\mathcal{R})\})} S_1$$

then assuming that there exists an object $A_u \in$ **Object**, the result of this **create** operation will be S_1, shown in Figure 12.16. This example is constructed such that the error occurs using either the original SW definitions or the modifications to this point. With A_u as the unborn object to be created by B_2, examine Lemma 4 with $E = \{A_1, A_2, A_u\}$. It is now possible for $\{A_1, A_2, A_u\}$ to modify B_* where before it was not.

As this example uses $\top$ relationships, no alterations thus far could not prevent this problem. The reflexivity alteration for **DASet** is subsumed by including sufficient

Figure 12.17 Value of **mutable** for S_0 and S_1.

$$\begin{aligned}\mathbf{mutable}_{S_0}(\{A_1, A_2, A_u\} \cap S_0{}^{\mathbf{existed}}) &= \mathbf{mutable}_{S_0}(\{A_1, A_2, A_u\}) \cap \{A_1, A_2, B_1, B_2\}\\ &= \{A_1, A_2, A_u\} \cap \{A_1, A_2, B_1, B_2\}\\ &= \{A_1, A_2\}\end{aligned}$$

$$\begin{aligned}\mathbf{mutable}_{S_1}(\{A_1, A_2, A_u\}) \cap S_0{}^{\mathbf{existed}} &= \mathbf{mutable}_{S_1}(\{A_1, A_2, A_u\}) \cap \{A_1, A_2, B_1, B_2\}\\ &= \{A_1, A_2, A_u, B_1, B_2\} \cap \{A_1, A_2, B_1, B_2\}\\ &= \{A_1, A_2, B_1, B_2\}\end{aligned}$$

Figure 12.18 Contradiction of Lemma 4.

Lemma 4 states that If $S_0 \xrightarrow{\alpha} S_1$, then for all E,

$$\mathbf{mutable}_{S_1}(E) \cap S_0{}^{\mathbf{existed}} \subseteq \mathbf{mutable}_{S_0}(E \cap S_0{}^{\mathbf{existed}})$$

But for the example defined above, when $E = \{A_1, A_2, A_u\}$:

$$\mathbf{mutable}_{S_1}(E) \cap S_0{}^{\mathbf{existed}} = \{A_1, A_2, B_1, B_2\} \not\subseteq \{A_1, A_2\} = \mathbf{mutable}_{S_0}(E \cap S_0{}^{\mathbf{existed}})$$

Figure 12.19 Definition of **ProjPotAcc**.

If $\mathcal{A}$: **Object** $\times$ **Object** $\rightarrow$ $\mathcal{R}_\perp$ is a resource relationship, then **ProjPotAcc**($\mathcal{A}$) : **Object** $\times$ **Object** $\rightarrow \mathcal{A}$ is defined to be:

$$\mathbf{ProjPotAcc}(\mathcal{A})(p,u)(x,y) = \begin{cases} \top & \text{if } x = p \wedge y = u \\ \top & \text{if } x = u \wedge y = p \\ \top & \text{if } x = p \wedge y = p \\ \top & \text{if } x = u \wedge y = u \\ \mathcal{A}(p,y) & \text{if } x = u \\ \mathcal{A}(x,p) & \text{if } y = u \\ \mathcal{A}(x,y) & \text{otherwise} \end{cases}$$

self-authority to cause reflexivity. Corrections in $\mathcal{R}_\perp$ and **mutable** are uninteresting as all capabilities are fully permissive. The contradiction does not examine the definition of **mutated** and therefore does not involve the modification of **wroteto** from Section 12.3.1.

12.4 Solution

The absurdity of Lemma 4 arises from the ability to ask questions about unborn objects without knowing their lineage. The main theorem never quantifies Lemma 4 with such an absurd E, but the analysis skips this detail. The solution is similar to operational mutability from SDM.

Figure 12.20 Theorem: **ProjPotAcc** approximates all operations.

If $S_0 \xrightarrow{\mathbf{create}(p,a)} S_1$ then if u is selected as a newly created object,

$$\mathbf{PotAcc}_{S_1} \leq \mathbf{ProjPotAcc}(\mathbf{PotAcc}_{S_0})(p, u)$$

and in all other cases

$$\mathbf{PotAcc}_{S_1} \leq \mathbf{PotAcc}_{S_0}$$

where the definition of $\leq$ is overloaded on access relations to have the meaning

$$\mathcal{A} \leq \mathcal{B} \Leftrightarrow \forall x, y\ \mathcal{A}(x, y) \leq \mathcal{B}(x, y)$$

12.4.1 Relating Access Relations

Integrating solutions from SDM begins by describing the relationships between access relations over each operation. From the updated definitions, **PotAcc** is a transitive reflexive closure and should approximate all operations except **create**. It is impossible for **PotAcc** to approximate **create**, as there is no advance knowledge about the lineage of new objects. However, in the worst case, a newly created object is just as permissive as its creator. This relationship between access relations is given by **ProjPotAcc** in Figure 12.19. Given **ProjPotAcc**, it is possible to demonstrate that **PotAcc** approximates all operations, as shown in Figure 12.20.

This proof is solved by very exhaustive case analysis similar to the SDM theorem *AG_project_endow* in Figure 7.17. It should also be apparent that under a transitive reflexive closure, if $\mathcal{A}$ has converged over **combine**, then **ProjPotAcc**($\mathcal{A}$) has also converged. This proof is less obvious, but is similar to the SDM theorem *AG_project_maximal* in Figure 8.10. Such converged resource relationships will be

Figure 12.21 Definition of potential access relation sequences.

If $S_0 \xrightarrow{\alpha_1} S_1 \ldots \xrightarrow{\alpha_n} S_n$ is an execution, then $\mathcal{P}_0 \xrightarrow{\alpha_1} \mathcal{P}_1 \ldots \xrightarrow{\alpha_n} \mathcal{P}_n$ is defined such that

$$\forall i\ \mathcal{P}_i = \mathbf{PotAcc}_{S_i}$$

Figure 12.22 Definition of one maximal access relation sequence approximating another.

If $\mathcal{P}_0 \xrightarrow{\alpha_1} \mathcal{P}_1 \ldots \xrightarrow{\alpha_n} \mathcal{P}_n$
and $\mathcal{Q}_0 \xrightarrow{\alpha_1} \mathcal{Q}_1 \ldots \xrightarrow{\alpha_n} \mathcal{Q}_n$ are **maximal** access relation sequences,
then $\leq$is overloaded as

$$\mathcal{P}_0 \to \mathcal{P}_1 \cdots \to \mathcal{P}_n \leq \mathcal{Q}_0 \to \mathcal{Q}_1 \cdots \to \mathcal{Q}_n \Leftrightarrow \forall i\ \mathcal{P}_i \leq \mathcal{Q}_i$$

called **maximal** throughout the remainder of this chapter.

It is now possible to define notation relating sequences of access relations that are sourced from executions, as shown in Figures 12.21 and 12.22. For clarity, the $\leq$ operator is further overloaded for **maximal** relation sequences.

12.4.2 Mutable Lineage

With the introduction of transformations between access relationships, the definition of mutability is too specific to incorporate new relationships as it is still dependent on **PotAcc**. Therefore, **mutable** must be parameterized over any access relationship

Figure 12.23 Parameterized definition of **mutable**.

$$\mathbf{mutable}_{\mathcal{A}}(E) = \begin{array}{ll} \{y | \exists x \in E, & \{\mathbf{wr}, \mathbf{exec}\} \cap \mathcal{A}(x,y) \neq \emptyset \vee \\ & \{\mathbf{rd}, \mathbf{wk}\} \cap \mathcal{A}(y,x) \neq \emptyset\} \end{array}$$

Figure 12.24 Theorem: **mutable** respects ordering.

If $\mathcal{A}$ and $\mathcal{B}$ are access relations and E and F are object sets, then

$$\mathcal{A} \leq \mathcal{B} \wedge E \subseteq F \ \Rightarrow \mathbf{mutable}_{\mathcal{A}}(E) \subseteq \mathbf{mutable}_{\mathcal{B}}(F)$$

Figure 12.25 Definition of **mutableLin**.

If $S_0 \xrightarrow{\alpha_1} S_1 \ldots \xrightarrow{\alpha_n} S_n$ is an execution,
and $\mathcal{P}_0 \xrightarrow{\alpha_1} \mathcal{P}_1 \ldots \xrightarrow{\alpha_n} \mathcal{P}_n \leq \mathcal{Q}_0 \to \mathcal{Q}_1 \cdots \to \mathcal{Q}_n$,
and $E \subseteq S_0{}^{\mathbf{existed}}$, and $q_i = \mathcal{Q}_0 \to \mathcal{Q}_1 \cdots \to \mathcal{Q}_i$ are sub-sequences

$$\begin{aligned}
\mathbf{mutableLin}_E(\mathcal{Q}_0) &= \mathbf{mutable}_{\mathcal{Q}_0}(E)\\
\mathbf{mutableLin}_E(\mathcal{Q}_{i-1} \to \mathcal{Q}_i) &= \mathbf{mutable}_{\mathcal{Q}_i}(E)\\
\mathbf{mutableLin}_E(q_n) &= \mathbf{mutableLin}_{\mathbf{mutableLin}_E(q_{n-1})}(\mathcal{Q}_{n-1} \to \mathcal{Q}_n)
\end{aligned}$$

$\mathcal{A}$ as in Figure 12.23. Although this chapter has striven to preserve notation from SW, it is necessary to deviate slightly to accomplish this change.

This form of **mutable** respects the various ordering relations for all parameters. Therefore, as the **ProjPotAcc** relationship preserves these orderings, **mutable** must respect **ProjPotAcc** as well. The theorem in Figure 12.24 corresponds to the *Proper_mutable_spec* theorem from SDM show in Figure 8.3.

The definition of **mutableLin** in Figure 12.25 sets up a sequence of **mutable**s

Figure 12.26 Theorem: **mutated** is captured by of a **mutableLin**.

If $S_0 \xrightarrow{\alpha_1} S_1 \ldots \xrightarrow{\alpha_n} S_n$ is an execution,
and $\mathcal{P}_0 \xrightarrow{\alpha_1} \mathcal{P}_1 \ldots \xrightarrow{\alpha_n} \mathcal{P}_n \leq \mathcal{Q}_0 \to \mathcal{Q}_1 \cdots \to \mathcal{Q}_n$,
and $E \subseteq S_0{}^{\mathbf{existed}}$,
and $e_i = S_0 \xrightarrow{\alpha_1} S_1 \ldots \xrightarrow{\alpha_i} S_i$ are sub-executions,
and $q_i = \mathcal{Q}_0 \to \mathcal{Q}_1 \cdots \to \mathcal{Q}_i$ are sub-sequences

$$\mathbf{mutated}_E(e_n) \subseteq \mathbf{mutableLin}_E(q_n)$$

Figure 12.27 Theorem: **mutableLin** respects ordering.

If E and F are object sets,
and $\mathcal{P}_0 \rightarrow \mathcal{P}_1 \cdots \rightarrow \mathcal{P}_n \leq \mathcal{Q}_0 \rightarrow \mathcal{Q}_1 \cdots \rightarrow \mathcal{Q}_n$,
and $p_i = \mathcal{P}_0 \rightarrow \mathcal{P}_1 \cdots \rightarrow \mathcal{P}_i$ are sub-sequences,
and $q_i = \mathcal{Q}_0 \rightarrow \mathcal{Q}_1 \cdots \rightarrow \mathcal{Q}_i$ are sub-sequences,

$$E \subseteq F \ \Rightarrow \mathbf{mutableLin}_E(p_i) \subseteq \mathbf{mutableLin}_F(q_i)$$

to capture object lineage while approximating **mutated**, similar to *mutable_execute* in SDM shown in Figure 8.5. The induction strategy for **mutableLin** is identical to **mutated** except that it starts with what is **mutable** and cycles **mutable** at each step. However, as it will become important to abstract over approximations, the definition does not directly perform reasoning on $\mathbf{PotAcc}_{S_n}$. Therefore, the definition of **mutableLin** is quite abstract. If all **PotAcc** operations are approximated reflexively, the theorem begins look a bit more like **mutable** and it should become clear that **mutableLin** also approximates **mutated** as shown in Figure 12.26. In SDM, this theorem corresponds to *mutated_approx_dirAcc_execute_mutable* in Figure 8.9.

Since **mutableLin** is defined inductively over **mutable** – which respects approximations and subset as shown in Figure 12.24 – it too must respect approximations and subset. This is presented in Figure 12.27 and is similar to the SDM theorem *Proper_mutable_execute* in Figure 8.6.

Figure 12.28 Definition : $\mathbf{Lin}_R$.

If $\mathcal{C}$ is an access relation,
let $\mathbf{Lin}_\mathcal{C}$ be the inductively defined set rooted at a **maximal** $\mathcal{C}$:

$$\begin{array}{rl} \mathcal{C} \in \mathbf{Lin}_\mathcal{C} & \\ \mathbf{ProjPotAcc}(\mathcal{A})(p,u) \in \mathbf{Lin}_\mathcal{C} & \text{if } \mathcal{A} \in \mathbf{Lin}_\mathcal{C} \wedge p \in \mathcal{A} \wedge u \notin \mathcal{A} \end{array}$$

where definition of $\in$ is overloaded as $\forall o, \mathcal{A}\ ,\ o \in \mathcal{A} \Leftrightarrow \mathcal{A}(o,o) \neq \bot$
And let $\preccurlyeq$ be the CPO defined by:

$$\begin{array}{rcll} \mathcal{A} & \preccurlyeq & \mathcal{A} & \\ \mathcal{A} & \preccurlyeq & \mathbf{ProjPotAcc}(\mathcal{A})(p,u) & \text{if } p \in \mathcal{A} \wedge u \notin \mathcal{A} \end{array}$$

Note, $\bot = \mathcal{C}$ and $\preccurlyeq$ is anti-symmetric by the definition of subset and **ProjPotAcc**.

12.4.3 Lineage

The partial order $\mathbf{Lin}_\mathcal{R}$ using **ProjPotAcc** as in Figure 12.28 is defined to assist reasoning about **mutableLin**. The purpose of $\mathbf{Lin}_\mathcal{R}$ is to fully abstract all possible lineage hierarchies rooted at access relation $\mathcal{R}$. Primarily, this abstracts all series of create operations, as the preconditions of $\mathbf{Lin}_\mathcal{R}$ are always satisfied by any sequence of operations due to state transitions. Additionally, $\mathbf{Lin}_\mathcal{R}$ parameterizes theorems without needing to instantiate them through laborious notation. The notion of $\mathbf{Lin}_R$ as a partial order is not directly present in SDM as it would require restructuring many previous theorems.

For any sequence of executions, $\mathbf{PotAcc}_{S_n}$ can be approximated by member of $\mathbf{Lin}_{\mathbf{PotAcc}_{S_0}}$ using the theorem from Figure 12.20. This is justified for two reasons. First, all non-create operations are self-approximating. Second, because $\mathbf{PotAcc}_{S_0}$ is **maximal** and **ProjPotAcc** is **maximal** given a **maximal** input, this sequence must

Figure 12.29 Theorem: **Lin** approximates **PotAcc**.

If $S_0 \xrightarrow{\alpha_1} S_1 \dots \xrightarrow{\alpha_n} S_n$ is an execution, $E \subseteq {S_0}^{\textbf{existed}}$, then

$$\exists \mathcal{L} \in \mathbf{Lin}_{\mathbf{PotAcc}_{S_0}} ,\ \mathbf{PotAcc}_{S_n} \leq \mathcal{L}$$

or

$$\exists \mathcal{L}_0 \to \mathcal{L}_1 \dots \to \mathcal{L}_n \in \mathbf{Lin}_{\mathbf{PotAcc}_{S_0}} ,\ \mathcal{P}_0 \xrightarrow{\alpha_1} \mathcal{P}_1 \dots \xrightarrow{\alpha_n} \mathcal{P}_n \leq \mathcal{L}_0 \to \mathcal{L}_1 \dots \to \mathcal{L}_n$$

Figure 12.30 Theorem : **mutableLin** always **existed**.

If $S_0 \xrightarrow{\alpha_1} S_1 \dots \xrightarrow{\alpha_n} S_n$ is an execution, and $\mathcal{L}_0 \to \mathcal{L}_1 \dots \to \mathcal{L}_n \in \mathbf{Lin}_{\mathbf{PotAcc}_{S_0}}$ and $E \subseteq {S_0}^{\textbf{existed}}$, and $l_i = \mathcal{L}_0 \to \mathcal{L}_1 \dots \to \mathcal{L}_i$ are sub-sequences

$$\mathbf{mutableLin}_E(l_i) \subseteq {S_i}^{\textbf{existed}}$$

be **maximal** by induction. This notion is presented in Figure 12.29 and loosely corresponds to the SDM theorem *mutable_execute_dirAcc_subset_potAcc* in Figure 8.7.

$\mathbf{mutableLin}_E$ cannot exceed ${S_n}^{\textbf{existed}}$ in every step for sequences in $\mathbf{Lin}_{\mathbf{PotAcc}_{S_0}}$, as it relies on **PotAcc** and **ProjPotAcc**, both of which do not consider non-existent objects. Theorem 12.30 captures this invariant formally in Figure 7.24 and corresponds to *ag_objs_spec_endow* theorem in SDM. Presently, it is a useful diagnostic for determining if the erroneous cases of SW Lemma 4 have been fixed. It will be specialized for precision in the theorem in Figure 12.31.

Figure 12.31 Theorem: **mutableLin** grows only by new objects at each step.

If $S_0 \xrightarrow{\alpha_1} S_1 \dots \xrightarrow{\alpha_n} S_n$ is an execution,
and $\mathcal{P}_0 \xrightarrow{\alpha_1} \mathcal{P}_1 \dots \xrightarrow{\alpha_n} \mathcal{P}_n \leq \mathcal{L}_0 \to \mathcal{L}_1 \dots \to \mathcal{L}_n \in \mathbf{Lin}_{\mathbf{PotAcc}_{S_0}}$,
and $E \subseteq {S_{i-1}}^{\mathbf{existed}}$,

$$\mathbf{mutableLin}_E(\mathcal{L}_{i-1} \to \mathcal{L}_i) \subseteq \begin{cases} E \cup \{u\} & \text{if } \mathcal{L}_i = \mathbf{ProjPotAcc}(\mathcal{L}_{i-1})(p,u) \wedge p \in E \\ E & \text{if } \mathcal{L}_i = \mathbf{ProjPotAcc}(\mathcal{L}_{i-1})(p,u) \wedge p \notin E \\ E & \text{if } \mathcal{L}_i = \mathcal{L}_{i-1} \end{cases}$$

Figure 12.32 Theorem: **mutableLin** is **mutable**.

If $S_0 \xrightarrow{\alpha_1} S_1 \dots \xrightarrow{\alpha_n} S_n$ is an execution,
and $\mathcal{P}_0 \xrightarrow{\alpha_1} \mathcal{P}_1 \dots \xrightarrow{\alpha_n} \mathcal{P}_n \leq \mathcal{L}_0 \to \mathcal{L}_1 \dots \to \mathcal{L}_n \in \mathbf{Lin}_{\mathbf{PotAcc}_{S_0}}$,
and $E \subseteq {S_i}^{\mathbf{existed}}$,
and $l_i = \mathcal{L}_0 \to \mathcal{L}_1 \dots \to \mathcal{L}_i$ are sub-sequences

$$\mathbf{mutableLin}_E(l_n) \cap {S_0}^{\mathbf{existed}} \subseteq \mathbf{mutable}_{\mathbf{PotAcc}_{S_0}}(E)$$

12.4.4 Main Theorem

There are two other theorems required to successfully prove the main theorem. Consider how **mutableLin** changes over sequences in $\mathbf{Lin}_R$. Each stepin $\mathbf{Lin}_R$ is given by some $\mathbf{ProjPotAcc}(\mathcal{A})(p,u)$. If $p \in E$, then **mutableLin** can only grown by u, otherwise **mutableLin** is constant. Another way of stating this is that **ProjPotAcc** is surjective on set inclusion. As the only objects added to **mutableLin** are not members of ${S_0}^{\mathbf{existed}}$, they are excluded when intersecting the result with what initially existed. As the base case of **mutableLin** is **mutable**, this induction must hold. Although these properties are many individual theorems in SDM, they are presented as a single property in Figure 12.31.

Figure 12.33 Corrected final theorem and proof.

If $S_0 \xrightarrow{\alpha_1} S_1 \dots \xrightarrow{\alpha_n} S_n$ is an execution,
and $E \subseteq S_i^{\mathbf{existed}}$,
and $e_i = S_0 \xrightarrow{\alpha_1} S_1 \dots \xrightarrow{\alpha_i} S_i$ are sub-executions

$$\mathbf{mutated}_{\mathbf{PotAcc}_{S_0}}(e_n) \cap S_0{}^{\mathbf{existed}} \subseteq \mathbf{mutable}_{\mathbf{PotAcc}_{S_0}}(E)$$

Proof:

$$\begin{array}{rl}
\mathbf{mutated}_{\mathbf{PotAcc}_{S_0}}(e_n) \cap S_0^{existed} & \subseteq \mathbf{mutable}_{\mathbf{PotAcc}_{S_0}}(E) \\
\multicolumn{2}{l}{\text{let } p_i = \mathcal{P}_0 \xrightarrow{\alpha_1} \mathcal{P}_1 \dots \xrightarrow{\alpha_i} \mathcal{P}_i \text{in}} \\
\multicolumn{2}{r}{\text{by Definition}} \\
\mathbf{mutableLin}_E(p_n) \cap S_0{}^{\mathbf{existed}} & \subseteq \mathbf{mutable}_{\mathbf{PotAcc}_{S_0}}(E) \\
\multicolumn{2}{r}{\text{by Theorem 12.26 instantiated reflexively}} \\
\multicolumn{2}{l}{\text{let } \mathcal{P}_0 \xrightarrow{\alpha_1} \mathcal{P}_1 \dots \xrightarrow{\alpha_n} \mathcal{P}_n \leq \mathcal{L}_0 \to \mathcal{L}_1 \dots \to \mathcal{L}_n \in \mathbf{Lin}_{\mathbf{PotAcc}_{S_0}}} \\
\multicolumn{2}{l}{\text{and } l_i = \mathcal{L}_0 \to \mathcal{L}_1 \dots \to \mathcal{L}_i \text{ in}} \\
\multicolumn{2}{r}{\text{by Theorem 12.29}} \\
\mathbf{mutableLin}_E(l_n) \cap S_0{}^{\mathbf{existed}} & \subseteq \mathbf{mutable}_{\mathbf{PotAcc}_{S_0}}(E) \\
\multicolumn{2}{r}{\text{by Theorem 12.27}} \\
\multicolumn{2}{r}{\text{this is Theorem 12.32}}
\end{array}$$

The $E \subseteq S_i^{\mathbf{existed}}$ requirement is an invariant of **mutableLin** over **Lin**, so that constraint can be satisfied. The remaining cases are evident by examination of the theorem in Figure 12.20. Using the result in Figure 12.31 as an induction step, the theorem in Figure 12.32 is produced.

With the exception of a parameterized definition of **mutable**, the statement of the main theorem in Figure 12.33 is identical to the SW theorem. Discharging the main theorem proceeds by subset transitivity and previous definitions. From Figure 12.26, **mutableLin** approximates **mutated** and may be substituted by transitivity. There must exist a lineage in $\mathbf{Lin}_{\mathbf{PotAcc}_{S_0}}$ that approximates **PotAcc**. Because **mutableLin** preserves ordering in Figure 12.27, it must also approximate **PotAcc** and may be substituted by transitivity. The remaining relation is precisely that **mutableLin** is **mutable** from Figure 12.32. Therefore, the main theorem holds; what is **mutated**, when restricted to what **existed**, is a subset of what was determined **mutable**.

As SDM began as a mechanical verification attempt for SW, it can be used to correct the proof execution. The corrections in this chapter are altered from those found in SDM, but follow the spirit of that proof. They also contain some of the suggestions and simplifications found in Chapter 11 on future work. These alterations generally impact the proof execution, but do not significantly impact the statement of the main theorem in SW. Previous conclusions based on the SW result [Sha03] can be expected to hold without modification.

Chapter 13

Related Work

This chapter discusses other access-control proofs that are related to SDM. Constructing confinement from memoryless subsystems is an approach that differs from the constructor model. The Provably Secure Operating System, PSOS, was the first attempt to specify and build a capability-based operating system that supported confinement through memoryless subsystems. SCOLL is a language for describing the behavior of entities within capability-based systems, and to reason about different authority configurations the system may come to have in the future. seL4 is the first microkernel to be fully connected to its specification by automated verification. The take-grant models for seL4 are similar to SDM.

13.1 Memoryless Subsystems

Memoryless subsystems are a protection mechanism for mutually suspicious programs. In "Memoryless Subsystems," [Fen74] Fenton constructs a general-purpose machine with infinite registers that are statically tagged as privileged or unprivileged, with the exception of the program counter. The program counter has a variable tag which is upgraded to privileged whenever impacted by privileged data. The machine provides a call-return procedure that preserves and restores the value and protection-state of the program counter and the machine may only halt in the unprivileged state. From this, he defines a memoryless system as one that, when started in an unprivileged state, no unprivileged information may be impacted by privileged information. His solution to leaking information by non-termination requires both privileged and unprivileged input to provide counters for their respective operations.

The use of memoryless subsystems to produce security policies depends on their initial configuration and how resulting data is dispersed by the system. Memoryless subsystems can be used as part of a confinement model that differs from the constructor model. A memoryless subsystem where all output is considered privileged must also be confined, assuming the destination of this privileged data is authorized by the caller. Enforcing this constraint upon a memoryless subsystem can be accomplished by erasing all non-privileged registers after execution or by requiring all registers to be privileged.

While it is possible to model memoryless confinement in SDM, it is difficult because SDM does not consider state as part of the model. Therefore, each invocation of a memoryless subsystem must be modeled in SDM as a fresh subsystem over shared unprivileged state, if any. This violates the intuition that there is an injective map from objects in the system to objects in the model.

13.2 PSOS

The Provably Secure Operating System, PSOS, was the first attempt at a mechanically verifiable operating system built on sound design principles [NBF+80]. The PSOS specification and implementation is structured as a layered TCB. Each layer is clearly delineated and refines the system by adding services and features not present in previous layers. The lowest layers of the system are constructed from computational hardware and the minimal software to construct capabilities. Subsequent layers range from memory management and I/O up to users and applications. Many high-level structures in modern capability-based systems resemble those developed as part of the PSOS specification.

The PSOS system specification was not only rendered in English text, but also in the non-procedural specification language SPECIAL. SPECIAL is an assertion language designed to capture system requirements and to verify their composition as part of the refinement process. Many theorems about the different layers of PSOS were verified in SPECIAL. The SPECIAL project would eventually be succeeded by the PVS [ORS92] proof system.

In PSOS, the Confined System Manager, or CSM, is responsible for constructing confined subsystems. As with the constructor, the CSM tests the specification of an object before instantiating it. Confinement provided by the CSM requires all objects to be recursively memoryless so that they may be safely reused by otherwise independent subsystems. This permits the CSM to memoize the result of producing confined subsystems to aid in confinement attestation and to permit complex structures to be built through repeated invocation.

When an application desires to make a confined entity, it presents a specification of that entity as a vector of capabilities. The `make_segments_confined` interface inspects all elements for capabilities to known confined system functions, previously certified objects, or another element of the presented vector. If satisfied, the CSM retains a copy of all capabilities and returns a copy of the vector where all write or delete access has been removed. These capabilities include the "confined-delete" permission, which can be used in conjunction with the CSM to perform deletes only to certified segments. To make extended objects, the CSM essentially has the same behavior, though the capability vector upon which the object is based is not granted "confined-delete" permissions. This restriction is in place as a confined object must not be able to delete capabilities from another confined object as this could cause data transfer. Objects not so constrained may write data to a confined subsystem through deletion without violating confinement.

Many theorems of high-level PSOS properties remain unpublished or unverified. Unfortunately, this includes the confinement property for the Confined Systems Manager. As it is using a memoryless confinement mechanism, the PSOS CSM can be conceptually modeled by SDM using fresh subsystems.

13.3 SCOLL

SCOLL, the Safe Collaboration Language, is a system description language for statically reasoning about the authority relationships of subsystems in capability-based systems. [JSR05] SCOLL allows developers to define a system, its semantics, an initial configuration, Horn clauses defining the behavior of various subsystems, and goal conditions which are to be preserved. Static reasoning is performed by the SCOLLAR model checker, which computes constraints that preserve the goal conditions while attempting to be minimal.

Alfred Spiessens presents a proof of the safety property for capability-based systems as part of his work on SCOLL. [Spi07] SCOLLAR relies on the tractability of the safety property to produce solutions which preserve the goal conditions. Positive goal conditions are *liveness* constraints, while negative goal conditions are *safety* constraints. Although SCOLL and SCOLLAR are not formally verified, they do produce meaningful, and more importantly, understandable solutions to the safety problem.

SCOLLAR has been used to examine and validate a number of capability-based patterns. Revocable capabilities via the caretaker pattern has been examined along with the methods by which it can be circumvented. Inescapable interposition has also been examined. In this scenario, two fully untrusted subjects are prevented from escalating their authority to one another by the interposition of proxies with known behavior. SCOLLAR has determined that no further behavioral restrictions are necessary to preserve the interposition arrangement, indicating that escalating authority is impossible.

SCOLL is an extremely useful tool for reasoning about safety in capability-based systems. However, SCOLL has not been extended to reason about information flow and therefore cannot describe the confinement problem. Due to their legibility, solutions presented by SCOLLAR are easily inspected and understood. However, the safety property upon which SCOLLAR relies has not been mechanically checked.

13.4 seL4

Section 10.1.3 presents an brief overview of the seL4 system. As previously mentioned, the seL4 microkernel [EKK06] is the L4 microkernel [Lie96] implementation modeled and inspected as part of the L4.verified project [Kle10]. To achieve this goal, the seL4 mirokernel is a capability-based system.

In contrast to the systems described in Chapter 2, memory management interfaces in seL4 are handled directly by the kernel. seL4 provides *Untyped Memory* objects representing system memory. An Untyped Memory object may be subdivided into other Untyped Memory objects or into other primitive kernel objects. The kernel maintains a *Capability Deriviation Tree*, or CDT, to track the relationships between capabilities naming object derivations. When an Untyped Memory object is retyped, the system checks the CDT to determine which capabilities to invalidate. This interface is extended to all capabilities, and processes may choose to either *copy* capabilites, creating siblings in the CDT, or by *minting* new and restricted capabilities as children in the CDT. The terms *copy* and *mint* sometimes appear as *imitate* and *grant* in the seL4 literature.

13.4.1 seL4 models

Elkaduwe et. al. presented the first access control model for seL4. [EKE08] [Elk10] Like SW and SDM, it is a state-transition model over a capability graph as a system state. All operations require a *sane* system state in which all capabilities name objects that are part of the system. The operation preconditions require a sane state and appropriate capabilities authorizing the operation.

The Elkaduwe model partitions data and capability operations based on access rights following earlier take-grant models [LS77] [Sny77]. Data-only operations require *Read* or *Write* permissions, while all state update operations are modeled with *Grant* and *Create*. The ability for a thread to retrieve a capability from a storage object is modeled by inverting the relationship so it appears that the storage object *Grant*s capabilities to the thread. The ability to create new objects requires holding a *Create* capability to some other object, presumably representing untyped memory.

The operations of the system are straight-forward. *SysNoOp* represents any self-mutation not involved with a capability. *SysRead* and *SysWrite* model explicit data mutation and require capabilities with *Read* or *Write* permissions respectively. *SysGrant* requires two capabilities: the invoked capability with the *Grant* permission and a capability to transfer. The *SysGrant* operation may "diminish" the transferred capability by reducing the available access rights. *SysCreate* also requires two capabilities: the invoked capability with the *Create* permission, and a *Grant* capability to an object where a capability to a new object will be inserted. *SysRemove* removes a specific capability from an object using a capability, and *SysRevoke* revokes an unspecified collection of capabilities from the system by repeatedly invoking *SysRemove* based on the capability derivation tree.

The model defines *subsystems* as an emergent property of the arrangement of capabilities. First, the *leak* judgment is defined from the holder of a *Grant* capability to its target. Next, two objects are *connected* if there is a *leak* between them in either direction. Finally, two objects are members of the same *subsystem* if they are in the transitive, reflexive closure of *connected.* They then verify that, from this point, it is not possible for the access rights between members of subsystems to increase over the execution of the system. This must be the case as both operations to add capabilities to the state of the system require the *Grant* permission.

The ability for information to flow between objects is examined next. The *canAccess* judgment is defined between objects A to B if any capability containing a subset of $\{Grant, Read, Write\}$ access rights exists between them in either direction. All information flow between objects is defined as the transitive, reflexive closure of *canAccess*. As before, this initial property is verified to hold over the life of the system. Therefore, the potential information flows over the life of the system is statically determined.

There are two problems with the model. First, the definitions of subsystems and authorized flows are emergent rather than intentional. That is, subsystems emerge as those collections of objects which can share capabilities irrespective of any human labeling or intent in their construction. This is not the standard definition of a subsystem. A subsystem is any subset of the system objects to be considered for inspection. By restricting subsystems in this way, many desirable properties, including confinement, become impossible to express. The same problem arises with information flow. While information flow may not increase between emergent subsystems, there is no discussion of whether that information flow was expected or how to constrain it without complete partitions.

The other problem with the model is that no access rights are required to remove capabilities from an object. Recall that capabilities are data and *capability flow is data flow.* Both SW and SDM ensures that all potential data flow are modeled as such. Unfortunately, no access rights are required to perform either *SysRemove* or *SysRevoke* in the Elkaduwe model. This creates an information channel when capabilities are deleted from objects without holding a *Write* or *Grant* capability.

In the case of *SysRemove*, a capability with no permissions still authorizes data to be written. The deletion of a capability can be observed by a subject through the attempt to use a now defunct capability. This could be caused by attempting to read from a page no longer in virtual memory, or by attempting to invoke a capability no longer in a CNode. Because this information flow is not captured by a *SysWrite* operation, it is not modeled as data motion.

The capability derivation tree, or CDT, is modeled as an axiom that also hides information flow at a much grander scale. The capability derivation tree is a map from capabilities to their parent. When the parent capability is revoked, all children are also revoked. The CDT is not modeled as part of the system and is assumed as a function that returns some unspecified collection of capabilities to remove. Performing *SysRevoke* could remove any capability in the model; it follows that performing *SysRevoke* may write data to *as many as all objects in the system.*

The CDT in the seL4 *implementation* is not so unconstrained. However, the issue can still arise as the revocation of a capability will delete all derived child capabilities. If a system is constructed such that a child capability crosses an intended subsystem boundary, then it is possible that an unauthorized communication channel between subsystems exists.

These flaws could have been detected if the Elkaduwe model had described information flow occurring at each operation, as is the case in SW and SDM. If the *SysRemove* and *SysRevoke* operations had been modeled with data motion, then proofs of consistency between the permission state of the model and the information flow by operations would not have been possible. Because the system is finite, such a proof would often contain enough information to infer the conditions for a counterexample, namely when no permissions are present. This would obligate an update to the model, presumably requiring a *Grant* permission to perform the *SysRevoke* and *SysRemove* operations.

These specification flaws highlight two subtleties arising when building a system model. Deletion, even deletion of access control permissions, is a form of data flow and can be easily overlooked when building a system model. Using emergent behavior to define expected behavior *weakens* a model.

The Boyton model is the second take-grant model developed for seL4. [Boy09] It's primary contribution is to extend the Elkaduwe model to reason about shared capability storage. Like most capability-based systems, seL4 uses capabilities to build high-level structures such as address spaces and applications. Some of these structures may be understood by the kernel itself, as is the case for address spaces. This causes some interpretations of the security infrastructure to appear to modify objects from a distance, such as threads modifying pages or capability-lists through an address space.

The solution presented by the Boyton proof is to introduce a *store* permission into the system, which is required to place a capability into thread storage. Object A is *directly store-connected* to B if A holds a capability with *Store* permission naming B. Objects are *store-connected* by transitive, reflexive closure over direct store connections. When examining the capabilities which may be used by an object during any operation, all capabilities accessible in store-connected objects are considered. The *SysGrant* operation is updated to write to any store-connected objects and the *SysCreate* operation returns a $\{Grant, Store\}$ capability. Taken together, these admit loads and stores through memory traversal to be captured by a single model operation.

The definition of *leak* is nearly unchanged in the Boyton proof. Because shared storage is another mechanism of transferring capabilities, any two objects that are store-connected to a third can *leak* capabilities and are therefore *connected.* The definition of a *subsystem* is still emergent as those collections of objects that are not connected. As with the Elkaduwe proof, this system can only reason about emergent mandatory properties and cannot reason about security policies, such as confinement. Additionally, no change to capability revocation is made. Capability deletion and the CDT remain a back-channel of information flow.

The Boyton proof does refine the notion of information flow. The definition *set-flow* is defined as A can flow to B if there is a capability with *Write* from A to B or a capability with *Read* from B to A. *Flow* is the transitive, reflexive closure of *set-flow.* This two-stage emergent behavior is slightly more refined than the *canAccess* definition of Elkaduwe as it does not consider all connections bidirectional. However, this definition only discusses data reads and writes that could occur via the read and write operations excluding all others.

It should be noted that these models have never been and cannot be connected to the current security proofs of seL4. The policies that have been verified are specified on a per-subject basis and remain low-level. The policy developer must have deep knowledge of primitive system structures to successfully define a policy. The policies that can be described are global, mandatory, and cannot describe dynamic border policies, such as confinement. Extending these policies to a high-level of abstraction remains an unsolved problem.

13.5 CHERI

CHERI is a capability-based instruction set architecture (ISA) and compiler that provides hardware support for the simultaneous execution of legacy and capability-based software. [WWN$^+$15] This support is extended to link legacy and capability-based software libraries at either the source or binary level and does not require a transition away from the C programming language. Offering a hybrid environment is intended to facilitate the transition of legacy TCB software into capability-based software.

The CHERI ISA is an extension of the 64-bit MIPS RISC ISA extended with capability support. The CHERI CPU is implemented as an FPGA soft-core processor with a standard MMU and capability co-processor. When the capability co-processor is disabled, CHERI can run MIPS operating systems without modification. A system supervisor can enable the capability co-processor to provide capability-based protection within applications.

CHERI capabilities are segment descriptors that also include permissions and a type field. Holding a CHERI capability grants the specified access to a range of memory defined at byte-granularity. Unlike IA32 segments, capability address translation is applied *before* MMU virtual address translation with the effect that capability-based segments are internal to each MMU-based address space. The CHERI ISA permits both capabilities and data to reside in the same MMU-based address space and maintains a hardware-enforced separation between them. Capabilities are tagged by a capability bit and, whenever a capability is modified by a data instruction, the capability bit is cleared and can not be recovered. Instructions modifying capabilities are always monotonically non-decreasing over the segment range and permissions.

Capabilities can be used either explicitly or implicitly in CHERI. They can be loaded and stored in the capability register file, or referenced and dereferenced in memory. Therefore, CHERI capabilites can be used as safe pointers by C compilers similar to Cyclone's fat pointers. [JMG$^+$02] "Global" capabilities are universally accessible in an MMU-based address space, and can be used to provide legacy software and libraries access to specific portions of an application heap.

CheriBSD is a modification of FreeBSD to support hybrid application compartmentalization using CHERI. Like a traditional Unix, each application is given its own MMU-based address space. Capability-based compartmentalization is primarily managed via the construction of a per-thread trusted stack that links a chain of disjoint per-compartment stacks. Invocations of *CCall* (or resp. *Return*) push (or pop) a capability to a new (or the previous) compartment stack to (or from) the trusted stack to preserve compartmentalization. CheriBSD also provides ABI wrappers to link capability-based software libraries with legacy libraries within an application, and vice versa. These wrappers come in two varieties with different goals: one enforcing the greatest compartmentalization and one enforcing the most compatibility. To support the use of capabilites when communicating with the system, CheriBSD provides capability-based wrappers for many system objects.

SDM models pure capability-based systems and is not intended to model hybrid systems like CheriBSD. Although CheriBSD guarantees compartmentalized code is capability-based and limits legacy access within compartments, it necessarily implements standard Unix mechanisms for legacy applications and libraries. These applications are not bound by capability-based access control, making cross-compartment policies difficult to implement.

SDM can be applied to a capability-based system running on CHERI, but may also be applicable to a compartment running in CheriBSD. Compartments in CheriBSD may be made purely capability-based, even when linked against legacy code using an

ABI wrapper. This might be accomplished by a compartment that does not permit global capabilities and only links legacy libraries fully sandboxed using capabilities. All access within such a compartment must be governed by the use of capabilities, including access outside the system. Provided no system object can be used to fabricate capabilities, the constructor pattern can be applied by a one-to-one correspondence with SDM objects and operational semantics.

Chapter 14

Conclusion

This dissertation presents SDM: a general, extensible model for capability-based systems and a subsequent mechanically checked proof of confinement. Confinement is the security policy requiring all outward information flow from a subsystem to be authorized. Despite practical implementations of confinement in capability-based systems, the enforcability of confinement continues to be questioned. The mechanically verified proof herein is designed to allay remaining concerns in that regard.

Confinement is relevant to security in capability-based systems for many reasons. Confinement sits at the border of mandatory and discretionary policies and is capable of implementing either. Given a composable confinement mechanism, applications can enforce robust security policies, scoping authority precisely where it is needed and consequently limiting the impact of faults. Different portions of the system can operate under distinct security policies implemented by independent security managers and mutually suspicious security managers can limit the interaction of the subsystems under their control. Confinement is not a built-in policy for capability-based systems; it is application-defined and non-privileged. The enforcability of confinement validates further use of application-defined polices. Confinement also forms the foundation of *agency*, providing a user with reasonable certainty that a program wielding a their authority is acting on their behalf.

The constructor pattern is used to implement confinement in many capability-based systems. The constructor is a trusted subsystem used to define and fabricate new confined subsystems. The constructor is responsible for performing the confinement test on the subsystem image, permitting parents to establish the confinement of new subsystems in advance of requesting their instantiation.

New subsystems are confined by the manner of their instantiation. The constraints placed on the construction of a confined subsystem are simple. All unauthorized capabilities in the new subsystem must be weak-only, trivially non-mutating, or completely internal to the subsystem. In the constructor pattern, capabilities to recursively confined constructors may also be permitted. Such constructors preserve confinement by induction.

SDM increases confidence in the confinement mechanism by using automated verification to increase rigor and simplify review. All of SDM is embedded and checked in the Coq proof assistant providing assurance for a correct proof execution. While this allows reviewers to safely elide complexities of proof execution, relevant definitions and theorems in SDM are intended to be amenable to review. Operations are defined legibly in three parts: a pre-condition, state-transition, and an upper-limit on information flow. The entire model is axiom-free and includes a trivial implementation for each abstraction. The implementation functions terminate and its data structures are finite. These two features eliminate the places where verification executions might contain flaws overlooked during review. Therefore, SDM presents a proof that capability-based systems can enforce confinement which is both comprehensive and comprehensible.

SDM contains the first mechanically verified proof of the safety property for capability-based systems. It defines a least upper bound on all potential access and describes how potential access evolves with each system operation. The potential access of the system is always decreasing between all existing objects and must also be decreasing between any existing subset of those objects. As an upper-bound, potential access solves the safety property for capability-based systems.

SDM demonstrates a least-upper bound on information flow for object-capability systems. Even though the system models potential data flow at each operation, the potential for all data motion corresponds to potential access. Intuitively, this is because each operation requires a capability with access rights which correspond to the information flow that occurs.

SDM provides the first mechanically verified proof of confinement for capability-based systems. The resulting upper-bound on information flow is used to validate the confinement test as a post-condition on the construction of new subsystems. The potential information flow out of a subsystem passing the confinement test is verified to be identical to the potential information flow of all subsystem consisting only of authorized capabilities. Therefore, the confinement test must produce a confined subsystem.

This dissertation also demonstrates how SDM can be applied to many capability-based systems and related models. SDM can be applied to KeyKOS, EROS, Coyotos, and seL4 and may be generally applied to many other capability-based systems. SDM specifically targets modeling Coyotos, a microkernel designed to be embedded for verification. The seL4 microkernel is also of interest as it has already been mechanically verified to meet its abstract specification. The SW proof upon which this model was originally based has been updated with the results of SDM. This update makes no substantive changes to the statement of the main theorem or underlying system and consequently preserves any results based on the SW proof. SDM conclusively demonstrates the enforcability of confinement in capability-based systems.

Bibliography

[AP67] William B. Ackerman and William W. Plummer. An implementation of a multiprocessing computer system. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, pages 5.1–5.10, New York, NY, USA, 1967. ACM.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, May 2004.

[BFH+92] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Berkeley, CA, USA, 1992. USENIX Association.

[Bii88] BiiN, Hillsboro, OR. *BiiN CPU Architecture Reference Manual*, preliminary edition, 1988.

[BL73] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547, MITRE, March 1973.

[Boe84] W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proceedings of the 7th DoD/NBS Computer Security Conference*, 1984.

[Boy09] Andrew Boyton. A verified shared capability model. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 254:25–44, October 2009.

[caj] Google Caja: A compiler for making third-party HTML, CSS and JavaScript safe for embedding. Online, `https://developers.google.com/caja/`.

[CDM01] Ajay Chander, Drew Dean, and John C. Mitchell. A state-transition model of trust management and access control, 2001.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[EKE08] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. In *VSTTE '08: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 99–114, Berlin, Heidelberg, 2008. Springer-Verlag.

[EKK06] Kevin Elphinstone, Gerwin Klein, and Rafal Kolanski. Formalising a high-performance microkernel. In Rustan Leino, editor, *Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06)*, Microsoft Research Technical Report MSR-TR-2006-117, pages 1–7, Seattle, USA, August 2006.

[Elk10] Dhammika Elkaduwe. *A Principled Approach To Kernel Memory Management*. PhD thesis, University of New South Wales, Sydney, New South Wales, Australia, 2010.

[Fab74] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[Fen74] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, February 1974.

[FL04] Jean-Christophe FilliÂtre and Pierre Letouzey. Functors for proofs and programs. In David Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *Lecture Notes in Computer Science*, pages 370–384. Springer Berlin Heidelberg, 2004.

[FN79] Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *In Proceedings of the National Computer Conference*, pages 329–334. AFIPS Press, 1979.

[Gut00] Peter Gutmann. *The Design and Verification of a Cryptographic Security Architecture*. PhD thesis, The University of Auckland, Auckland, New Zealand, 2000.

[Har85] Norman Hardy. KeyKOS architecture. *SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[Har86] Norman Hardy. Computer security system. United States Patent number 4,584,639, April 1986.

[HKN05] Shai Halevi, Paul A. Karger, and Dalit Naor. Enforcing confinement in distributed storage and a cryptographic model for access control. Technical report, Complete Characterization of Adversaries Tolerable in General Multiparty Computation. PODC '97, 2005.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[IBM81] IBM, Rochester, MN, USA. *IBM System/38 Functoinal Reference Manual*, second edition, February 1981.

[Int83] Intel Corporation, Santa Clara, CA. *iAPX 432 General Data Processor Architecture Reference Manual*, fourth edition, 1983.

[JMG+02] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James

Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[JSR05] Yves Jaradin, Fred Spiessens, and Peter Van Roy. SCOLL – a language for safe capability based collaboration. Technical report, Universitè catholique de Louvain, 2005.

[KAE+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

[KL87] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. *IEEE Transactions on Software Engineering*, 13:202–207, 1987.

[Kle10] Gerwin Klein. The l4.verified project - next steps. In GaryT. Leavens, Peter O'Hearn, and SriramK. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 86–96. Springer Berlin Heidelberg, 2010.

[KZB+90] Paul A. Karger, Mary Ellen Zurko, Douglas W. Benin, Andrew H. Mason, and Clifford E. Kahn. A VMM security kernel for the VAX architecture.

In *In Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 2–19. IEEE Computer Society Press, 1990.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[Lam74] Butler W. Lampson. Protection. *SIGOPS Operating Systems Review*, 8(1):18–24, January 1974.

[LE96] Jochen Liedtke and Kevin Elphinstone. Guarded page tables on MIPS R4600 or an exercise in architecture-dependent micro optimization. *SIGOPS Operating Systems Review*, 30(1):4–15, January 1996.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems.* Butterworth-Heinemann, Newton, MA, USA, 1984.

[Lie96] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[LS77] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM (JACM)*, 24(3):455–464, July 1977.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[MS03] Mark S. Miller and Jonathan S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *In 8th Asian Computing Science Conference (ASIAN03)*, pages 224–242, 2003.

[NBF$^+$80] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report Computer Science Laboratory Technical Report, CSL-116, 2nd ed., SRI International, May 1980.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Pro03] Niels Provos. Preventing privilege escalation. In *In Proceedings of the 12th USENIX Security Symposium*, pages 231–242, 2003.

[Raj89] S. A. Rajunas. The KeyKOS/KeySAFE system design. Technical Report SEC009-01, Key Logic, Inc., March 1989.

[Ree96] Jonathan A. Rees. A security kernel based on the lambda-calculus. *A.I. MEMO 1564, MIT*, 1564, 1996.

[SA08] Jonathan S. Shapiro and Jonathan W. Adams. Coyotos microkernel specification, September 2008.

[SDS08] Jonathan Shapiro, Scott Doerrie, and Swaroop Sridhar. BitC 0.10 language specification, 2008.

[Sha03] Jonathan S. Shapiro. The practical application of a decidable access model, 2003.

[SM02] Marc Stiegler and Mark S. Miller. A capability based client: The DarpaBrowser. Technical Report Technical Report Focused Research Topic 5 / BAA-00-06-SNK, Combex Inc., June 2002.

[Sny77] Lawrence Snyder. On the synthesis and analysis of protection systems. *SIGOPS Operating Systems Review*, 11(5):141–150, November 1977.

[Sol96] F.G. Soltis. *Inside the AS/400.* Duke Communications International, 1996.

[Spi07] Alfred Spiessens. *Patterns of Safe Collaboration.* PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2007.

[SSF97] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A capability system. Technical report, University of Pennsylvania, 1997.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, New York, NY, USA, 1999. ACM.

[SW00] Jonathan S. Shapiro and Sam Weber. Verifying the EROS confinement mechanism. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 166, Washington, DC, USA, 2000. IEEE Computer Society.

[TST14] NICTA Trustworthy Systems Team. seL4 reference manual API version 1.3, July 2014.

[UsL85] An Unknown, Dod std, and Donald C. Latham. Department of defense standard department of defense trusted computer system evaluation criteria, 1985.

[WWN$^+$15] Robert N. M. Watson, Jonathan Woodruff, Peter G Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[YH10] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. *SIGPLAN Not.*, 45(6):99–110, June 2010.

Vita

M. Scott Doerrie was born the son of Mary Lou Doerrie and Steve Hardy on October 3rd, 1979 in Robbinsdale, MN. After graduating from high school in 1998 from Coon Rapids, MN, he studied Computer Science at the Gustavus Adolphus College in Saint Peter, MN, where he earned his Bachelor of Arts in 2002. He joined the Systems Research Laboratory (SRL) as a Research Assistant at the Johns Hopkins University in 2003.

While at SRL, he was involved in the design of both Coyotos, BitC, and SDM. He notably contributed formalisms for reasoning about on-line address-space traversal and update algorithms for the preliminary Coyotos GPT design. He also assisted with the development of BitC with a focus on keeping the language tractable for future analysis. Ultimately, his graduate research focused on SDM, demonstrating through automated verification that capability-based systems could enforce security policies, namely confinement.

M. Scott Doerrie gained experience as an instructor while completing his doctoral research. As a co-instructor with Swaroop Sridhar for three consecutive years, he designed and presented novel course material for the “Operating Systems” course at Johns Hopkins University. He also prepared a half-semester seminar for the Systems Research Laboratory on the use of the logical frameworks.

In his doctoral thesis, M. Scott Doerrie combined multiple disciplines of Computer Science; he drew from the domains of operating systems, access control mechanisms, machine-assisted verification, safe language run-times, and advanced type systems. His future research interests include building and improving mechanisms to produce secure systems.