Verifying Operating System Security

Computer and Information Sciences Technical Report MS-CIS-97-26

J. S. Shapiro, S. Weber[1]

Distributed Systems Laboratory

University of Pennsylvania, Philadelphia, PA 19104-6389

shap@dsl.cis.upenn.edu

samweber@central.cis.upenn.edu

July 19, 1997

Abstract

A *confined* program is one which is unable to leak information to an unauthorized party or modify unauthorized resources. Confinement is an essential feature of any secure component-based system. This paper presents a proof of correctness of the EROS operating system architecture with respect to confinement. We give a formal statement of the requirements, construct a model of the architecture's security policy and operational semantics, and show that the architecture enforces the confinement requirements if a small number of initial static checks on the confined subsystem are satisfied. The mechanism does not rely on the run-time values of user state or analysis of the programs' algorithm(s).

Our verification methodology borrows heavily from techniques developed in the programming languages community. We view the operating system as a programming language whose operations are the kernel calls. This has the advantage that the security requirements of concern can be stated in forms analogous to those of type inference and type soundness – which programming language techniques are well suited to deal with. The proof identifies a set of necessary fundamental lemmas that any system must observe in order to be able to confine information flow. The method used generalizes to any capability system.

Keywords: proof of correctness, capability systems, operating systems, confinement, verification, formal specification.

1 Introduction

Component-based systems, such as web applets and ActiveX components, have increased awareness of a certain class of security problems. Users wish to execute subsystems which are provided by many parties, some of whom are unknown, untrusted or even actively adversarial. General users are typically not in a position to evaluate these subsystems, nor would it be practical for them to do so. A secure architecture should therefore provide a means for composing untrusted modules without compromising security.

Informally, the requirements for such a system are:

1. A component should not be able to leak information to any unauthorized party, or modify any unauthorized resource, or acquire illicit means by which to do so (the *principle of confinement* [Lam73]).
2. Components should be restricted to exactly and only the authority necessary for them to operate (the *principle of least privilege*).
3. Components should be composable. This requires that authority be delegatable and that a component be able to load subcomponents subject to the resource limits imposed by the user. Such composition must not violate points (1) or (2).

[1] This research was supported by DARPA under Contracts #N66001-96-C-852, #MDA972-95-1-0013, and #DABT63-95-C-0073. Additional support was provided by the AT&T Foundation, and the Hewlett-Packard and Intel Corporations.

Copyright ©1997, Jonathan S. Shapiro and Sam Weber. Permission is granted to redistribute this document in electronic or paper form, provided that this copyright notice is retained.

4. Clients must be able to authorize specific holes in the confinement boundary at run time (e.g. to allow working state to be saved).

The confinement restriction induces a partition on resources that must be dynamically maintained. Any modification of resources inside this partition (i.e. the confined resources) must not be visible outside the partition. Explicit interprocess communication, mutable memory, and shared mutable name spaces (such as file systems) are all potential sources of information leakage. Sandboxing does not permit dynamically authorized channels, nor does it address information flow in multiple process subsystems. Ideally, a confinement mechanism should make it possible to verify that a subsystem is confined by performing a small number of static checks before starting a given subsystem.

A proof of correctness for such an architecture is challenging for several reasons:

- The formal statement of requirements is complex, and must be independent of the system policy specification if it is to be useful.
- It is difficult to choose an appropriate level of detail for modeling. Most models include extraneous information which greatly increases the burden of verification.
- Operating systems are notoriously complex. Specifying the semantics of their operation is difficult, tedious, and error-prone.
- In many systems the primitive operations most directly related to security are notoriously difficult to reduce to formal statements. For example, the UNIX $chown(\cdot)$ operation has global impact on every process in the system and has unpredictable consequences.

Few security proofs have made significant progress on nontrivial systems.

The purpose of this paper is to describe how existing programming language techniques can be extended to address security issues in a particular class of operating systems known as capability systems, and to prove the correctness of the confinement mechanism for one such system. A capability is an unforgeable (*resource descriptor*, *access rights*) pair. It designates a specific resource and authorizes a set of actions that can be performed by invoking the capability. Possession of a capability is a necessary *and sufficient* proof of authority to perform the actions it authorizes on the designated resource. While confinement is not a traditional programming language problem, the capability framework enables apply some of the proof techniques generated by the programming language community to prove that confinement can be enforced by a suitable capability architecture. We have verified the security properties of EROS, the Extremely Reliable Operating System.

To verify that EROS can support confinement, we have developed a formal statement of requirements and a simplified model, *Agape*, that is both strictly more powerful and more general than the EROS architecture. Agape views the OS-provided primitive operations as a serializably concurrent language whose operations are the kernel calls. We have modeled Agape's security policy, access control mechanisms, and operational semantics, and have shown that the Agape semantics satisfies the requirements of confinement. Our proof provides a small number of essential lemmas that must be satisfied for any system to provide confinement, and should generalize to other capability systems.

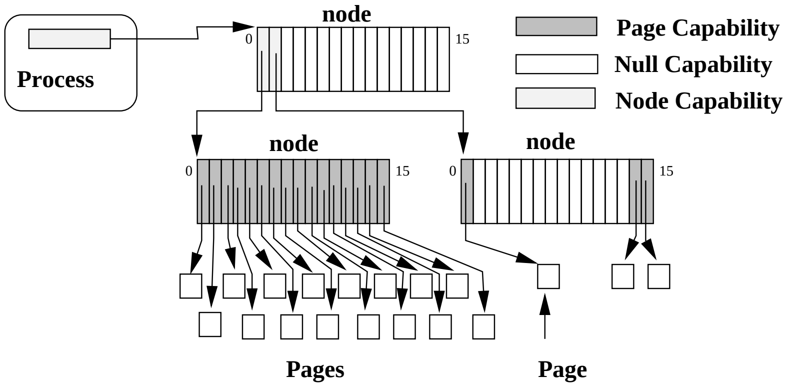
The balance of this paper proceeds as follows. Section 2 presents a brief summary of EROS and the constructor mechanism, and provides an informal intuition for why the constructor mechanism results in confinement. Having described the architecture and mechanism, we describe and justify key parts of our modeling and proof methodology (Section 3). Section 4 presents the model itself, the formal statement of requirements, and the key pieces of the correctness proof (an unabridged proof may be found in [SW97a]). Finally, we discuss the implications of this work and its effect on the original system architecture and design.

2 The EROS Architecture

EROS is a small, high-performance operating system designed for security and reliability, and specifically to support efficient confinement [Sha96a]. The system provides a small number of primitive resource types and a small set of precisely specified actions that can be performed on each resource. It is intended for currently being used as a substrate for active networking research. EROS's predecessor, KeyKOS, has been used to support production VISA transaction workloads [Har85, Lan93]. EROS provides an efficient capability implementation [Sha96c] whose performance comes within a few cycles of the limits of the underlying hardware. The system includes a generic utility for building confined subsystems: **constructors**

For purposes of verification, the essential EROS resource types are: processes, data pages, and nodes.[2] A **process** names an address space and executes the program in that address space in the usual way. To prevent forgery, data and capabilities are partitioned. Load and store operations require that data go to **pages** and capabilities to **nodes**. Operations on processes take care to enforce the appropriate type restrictions. Address spaces are constructed as a tree of nodes whose leaves are data pages. A complete process showing all resource types is shown in Figure 1.

Figure 1 A complete EROS process



The EROS object and protection model are extended to secondary storage by persistence. This ensures that the security model does not change at the secondary storage boundary. Periodically, the system performs an efficient checkpoint operation, recording the state of all objects changed since the last snapshot, including processes. There is no file system; if file-like behavior is required by an application, a process is constructed that implements file behavior.

2.1 Capabilities and Invocation

Every EROS resource has an associated capability type. Node and page capabilities come in **read-only** and **weak** (see below) variants.[3] Because EROS is a partitioned capability system, and because there are no operations that amplify the authority of a capability, three statements can be made about the architecture:

- All relationships are explicit: unless a process holds a capability to a resource it cannot invoke that resource.
- Capabilities cannot be forged. There is no way for a program to illicitly obtain access to a resource.
- A confined set of resources is closed with respect to addition. If the subsystem does not initially have access to a resource, there is no way to obtain access except through some resource explicitly authorized by the user.

There are two capability types that name applications: start capabilities and resume capabilities. A **start capability** allows the holder to invoke the application running within a process (as distinct from the process itself).

[2] The multimedia and scheduling resource control constructs have been omitted from this paper for simplicity.
[3] In the interest of brevity, we have omitted from this paper several capabilities and attributes that are not relevant to the proof or whose semantics are a subset of some other object we have included. Details can be found in [Sha96a].

Invocation	*Invoker* Transition	Cap. Type	*Recipient* Transition
CALL	$running \rightarrow waiting$	Start	$available \rightarrow running$
RETURN	$running \rightarrow available$	Resume	$waiting \rightarrow running$
SEND	$running \rightarrow running$		

Table 1: State transitions for invocations

To perform an action on an object, a process performs a **call**, **return**, or **send** invocation on a capability. Processes can be in one of three states: **running**, **waiting**, or **available**. The state transition of the invoker is determined by the invocation type, while the state transition of the recipient is determined by the type of the capability (Table 1). To ensure return-once semantics, the call operation generates a distinguished capability called a **resume capability**; invoking a resume capability causes all of its copies to be invalidated, which ensures that a call will receive at most one reply. Collectively, these operations realize a somewhat more general mechanism than conventional remote procedure calls; clever uses of **return** can be used to perform efficient demultiplexing.

One unusual attribute of EROS is the introduction of **weak capabilities**. While a read-only capability prevents modifications to the resource it designates, it permits the holder to fetch any read-write capabilities that may reside in that resource. Weak capabilities enforce *transitively* read-only access. If an invocation on a capability, such as the "fetch" operation on a node capability, would return some other capability, and the invoked capability is weak, then returned capabilities are weakened according to the rules in Table 2 before being returned. The effect of the weak attribute is to provide *transitive* read-only authority, which simplifies the construction of confined subsystems.

Input Type	Output Type
Any Page Capability	RO, Page Capability
Any Node Capability	RO, Weak Node Capability
All others	Null capability

Table 2: The weaken operation

2.2 Constructors

The **constructor** is a trusted EROS application that builds confined subsystems. The builder of an application buys a constructor and installs those capabilities the process should hold when it is first started. As each capability is added to the constructor, the constructor examines it to see if it conveys any authority to mutate objects. A capability is safe if it trivially conveys no mutate authority, or if it is a **requester capability** (see below) to a constructor, and that constructor in turn generates safe products. Weak capabilities allow a capability pointing to complex structures to be added safely without requiring that the structure be traversed. If the capability is *not* safe, the constructor adds it to a set of known **holes**. The builder then "freezes" the constructor, after which it will accept no further additions.

When a client needs a new instance of a subsystem, such as a file or a sort utility, they present a set of acceptable holes to the constructor and ask if the constructor output is confined modulo those holes. If the result of $\{holes\} - \{authorized\}$ is empty, the product is safe. The intuition is this: there is no way to cause damage via safe capabilities, and the user has authorized all of the others. Because requester capabilities to safe constructors are safe, the constructor mechanism permits very complex subsystems to be built behind a confinement boundary.

3 Outline of Work

In this section we present an overview of this work, including certain aspects of the methodology that are not apparent from the formal content.

System Model. Before specifying the system operation, or even stating what the requirements of the system are, we have to choose an appropriate level of analysis. Too large a gap between the implementation and the specification enlarges the chances of a mismatch between the two, which often results in failure to correctly preserve the requirements.

A side-effect of EROS persistence is that there is a natural level of analysis. All system state is periodically recorded by the checkpoint mechanism to nodes, pages, and processes. Since the checkpoint works, we know that there is no other state kept in the system, and that all system operations consist of modifications of this state. Thus, a model of this state and these operations is natural.

In any operating system, a certain amount of complexity is due to performance requirements or hardware constraints. We would like to omit such concerns from the model if this can be done without sacrificing correctness. Where Agape departs from EROS, we maintain the invariant that *Agape must be at least as powerful as EROS*. Any operation that can be performed in EROS can be done in Agape. This is extremely important: if Agape meets the security requirements, and the behavior of Agape is a superset of EROS' behavior, then we are assured that EROS also meets the security requirements.

For example, reading a page in EROS produces the value that was last written. Agape, however, does not keep track of the contents of pages – pages hold user-level data and Agape's security relies only on kernel data. This is safe, since it *increases* the possible behavior of the system, and has the advantage that we do not need to keep track of the page values. Also, we demonstrate that the security of EROS does not rely upon this aspect of its behavior.

Requirements Statement. The definition of the confinement requirement is independent of both the access control model and the Agape operational semantics. Failing to do this would have result in a meaningless verification: "EROS provides the security that results from its security policy."

A confined process should not be able to affect any non-authorized entities outside the confinement boundary. Given an execution e and a set of entities E we want to define a function $\mathbf{mutated}_E(e)$ whose value is the set of entities that were affected by E in the execution. By doing so, we state what the communication channels of the system are.

The **mutated** relation is transitive. Suppose a process p modifies a resource x, and another process q subsequently reads x. In this sequence of events, p has mutated q, even though the two processes never directly communicated. Our definition of **mutated** takes this form of indirect communication into account.

Access Control. The confinement mechanism defines a security policy that is assured by the EROS (and Agape) access control model. While the confinement requirements are stated in terms of the execution of the system, the security policy must be statically verifiable. By looking only at a small portion of the current system state, the constructor must be able to judge the confinement of its products.

The function $\mathbf{mutable}_S(E)$ takes a single state S and from that computes the set of all the entities that might in the future be mutated by the entities E. Although this function is not actually computed by the operating system (for obvious efficiency reasons), it is the basis for the access checks performed by the operational semantics. The check performed by the constructor therefore provides a conservative verification of the requirements.

Proof of Correctness. Finally, we have to show that the system's security policies implement its requirements. For example, we must show that if e is an execution of the system whose initial state is S_0, and if E is any set of resources, then

$$\mathbf{mutated}_E(e) \subseteq \mathbf{mutable}_{S_0}(E)$$

In other words, if some entity was mutated in an execution, then the security policy must have said that it was originally mutable. This statement relies upon properties of both the operational semantics, and the security policy. One of the aims of this work is to state these properties in such a way as to be applicable to other situations.

4 EROS Verification

4.1 Semantic Entities of Agape

Agape is a structured operational semantics whose judgments are of the form

$$S_0 \xrightarrow{(p,\alpha)} S_1$$

with the intended meaning "When the system was in state S_0 process p did action/kernel call α, which resulted in state S_1."

Figure 2 Semantic Entities

	Universal Sets		*Capability Types*
$\mathcal{R}$	set of system resources	**PageCap**	the set of page capabilities
Caps	set of capabilities	**NodeCap**	the set of node capabilities
Attr	set of capability attributes	**StartCap**	the set of start capabilities
$\mathcal{S}$	set of system states	**ResumeCap**	the set of resume capabilities
	Resource Types		*Utility Functions*
Page	The set of pages	$\mathbf{target}(\cdot)$	$\mathbf{Caps} \to \mathcal{R}$
Node	The set of nodes	$\mathbf{attribute}(\cdot)$	$\mathbf{Caps} \to \mathbf{Attr}$
$\mathcal{P}$	The set of processes		

Figure 2 gives the basic semantic entities of Agape. The system resources consist of disk pages, processes, and nodes. Capabilities can be either start, resume, page or node capabilities. If k is a capability, $\mathbf{target}(k)$ is the resource that capability refers to, and $\mathbf{attribute}(k)$ is that capability's attributes. Attributes are represented as a subset of $\{\mathbf{readonly}, \mathbf{weak}\}$; a full-fledged capability has the empty set as its attributes, while a weaker capability can be read-only, weak, or both. Not all combinations of capability types and attributes make sense. For instance, there is no such thing as a weak resume capability. The system state will ensure that such nonsensical capabilities cannot appear.

We will use **ResumeCap**, **StartCap**, **NodeCap**, ... as constructors: If $d \in \mathbf{Page}$, then $\mathbf{PageCap}(\{\mathbf{weak}\}, d)$ is the unique weak page capability that refers to d. Since start and resume capabilities always have attribute $\emptyset$, their constructors have only one argument.

One major decision in this work arises in choosing an appropriate representation for the system state. The key observation is that system security must not rely on user-level state or the behavior of user processes. Therefore, in Agape, *system state is kernel state.* We treat processes as completely non-deterministic, able to invoke any capabilities that they have in their possession. Furthermore, since pages are user-level data, we do not maintain any constraints upon the contents of pages. Ignoring user state results in a more powerful model: processes are maximally hostile.

Another notable departure from EROS was in handling the contents of nodes and the capabilities held by processes. EROS nodes are fixed size arrays, as are the data structures in which processes keep their capabilities. However, there is nothing particularly important about the size of these data structures.[4] In Agape, nodes and processes hold sets of capabilities, instead of arrays.

The definition of Agape state is the following:

The sanity conditions ensure that we don't have to deal with nonsensical capabilities or processes in impossible running states. The set of dead processes exists so that we can ensure that newly created

[4] And, in fact, the implementers changed the size of nodes just as this work was nearing completion.

Figure 3 System state

System state: $S = (S^{\mathbf{run}}, S^{\mathbf{wait}}, S^{\mathbf{avail}}, S^{\mathbf{proc}}, S^{\mathbf{node}}, S^{\mathbf{dead}}) \in \mathcal{S}$

$S^{\mathbf{run}} \subseteq \mathcal{P}$	the set of running processes
$S^{\mathbf{wait}} \subseteq \mathcal{P}$	the set of waiting processes
$S^{\mathbf{avail}} \subseteq \mathcal{P}$	the set of available processes
$S^{\mathbf{proc}} : \mathcal{P} \rightarrow 2^{\mathbf{Caps}}$	the map between processes and sets of capabilities
$S^{\mathbf{node}} : \mathbf{Node} \rightarrow 2^{\mathbf{Caps}}$	the map between nodes and sets of capabilities
$S^{\mathbf{dead}} \subseteq \mathcal{P}$	the set of all processes which have been destroyed

Notation: $S^{\mathbf{exist}} = S^{\mathbf{run}} \cup S^{\mathbf{wait}} \cup S^{\mathbf{avail}}$

Sanity conditions:

1. $S^{\mathbf{run}}$, $S^{\mathbf{wait}}$, $S^{\mathbf{avail}}$ and $S^{\mathbf{dead}}$ are disjoint
2. $S^{\mathbf{exist}}$ is finite
3. The following implications hold:
   (a) $k \in \mathbf{extant}(S) \cap \mathbf{StartCap} \implies \mathbf{target}(k) \in S^{\mathbf{exist}}$
   (b) $k \in \mathbf{extant}(S) \cap \mathbf{ResumeCap} \implies \mathbf{target}(k) \in S^{\mathbf{wait}}$
   (c) $k \in \mathbf{extant}(S) \cap (\mathbf{StartCap} \cup \mathbf{ResumeCap}) \implies \mathbf{attribute}(k) = \emptyset$
   (d) $k \in \mathbf{extant}(S) \cap \mathbf{PageCap} \implies \mathbf{weak} \notin \mathbf{attribute}(k)$

$$\textit{where } \mathbf{extant}(S) = \bigcup_{p \in S^{\mathbf{exist}}} S^{\mathbf{proc}}(p) \cup \bigcup_{c \in \mathbf{Node}} S^{\mathbf{node}}(c)$$

processes are distinct from any that have been destroyed.

The set of possible actions that a process can take are enumerated in Figure 4. The restrictions on the arguments of the actions prevent things like attempts to fetch a capability from a page.

4.2 Operational Semantics

Before defining the transition relation, we must state what happens when a capability is retrieved from a node using a weak capability. Below we define the function $\mathbf{weaken}(k)$ which is the result of fetching the capability k by way of a weak capability. In many cases, no capability is returned by the weakened fetch operation, and $\mathbf{weaken}(k)$ is the empty set. Otherwise, $\mathbf{weaken}(k)$ is the singleton set whose member is the appropriately reduced page or node capability.

$$\mathbf{weaken}(k) = \begin{cases} \{\mathbf{PageCap}(\{\mathbf{readonly}\}, \mathbf{target}(k))\} & \text{if } k \in \mathbf{PageCap} \\ \{\mathbf{PageCap}(\{\mathbf{readonly}, \mathbf{weak}\}, \mathbf{target}(k))\} & \text{if } k \in \mathbf{NodeCap} \\ \{\} & \text{otherwise} \end{cases}$$

The function $\mathbf{erase}(m, e)$ removes all elements of e from the mapping m. It is used in the definition when all outstanding capabilities to an entity must be destroyed.

We define the transition relation $\longrightarrow : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ as follows:

$$S \xrightarrow{(p,\alpha)} S'$$

where $p \in S^{\mathbf{run}}$ and the state transformation performed by each action α is given in Appendix A. The read and write operations do not affect the state at all. Since these operations only affect pages, and the system state is only kernel data, this is correct. Note that the fact of a read or write operation *is* observable – the requirements statement will insist upon the proper security behavior.

Figure 4 Action types

Action Type	*Restrictions*	*Description*
$\mathbf{write}(k)$	$k \in \mathbf{PageCap}, \mathbf{readonly} \notin \mathbf{attribute}(k)$	write data to a page
$\mathbf{read}(k)$	$k \in \mathbf{PageCap}$	read data from a page
$\mathbf{fetch}(k, o)$	$k \in \mathbf{NodeCap}, o \in S^{\mathbf{node}}(\mathbf{target}(k))$, $(\mathbf{weak} \in \mathbf{attribute}(k) \implies o \in (\mathbf{PageCap} \cup \mathbf{NodeCap}))$	fetch capability from node
$\mathbf{store}(k, o)$	$k \in \mathbf{NodeCap}, \mathbf{readonly} \notin \mathbf{attribute}(k), o \in S^{\mathbf{proc}}(p)$	store capability to node
$\mathbf{capremove}(k, o)$	$k \cap \mathbf{NodeCap}, \mathbf{readonly} \notin \mathbf{attribute}(k)$, $o \in S^{\mathbf{node}}(\mathbf{target}(k))$	remove capability from node
$\mathbf{call}(k, a)$	$k \in (\mathbf{StartCap} \cup \mathbf{ResumeCap}), a \subseteq S^{\mathbf{proc}}(p)$	call a process
$\mathbf{return}(k, a)$	$k \in (\mathbf{StartCap} \cup \mathbf{ResumeCap}), a \subseteq S^{\mathbf{proc}}(p)$	return to a process
$\mathbf{remove}(k)$		remove capability from process' capability set
$\mathbf{create}(a)$	$a \subseteq S^{\mathbf{proc}}(p)$	create new process with a its set of initial capabilities
$\mathbf{destroy}()$		process self-destruction

In all cases $k \in S^{\mathbf{proc}}(p)$.

Lemma. *The transition relation $\longrightarrow$ is well-defined. That is, if $S \xrightarrow{(p,\alpha)} S'$, then S' is a state.*

This lemma is proven by straightforward case analysis.

4.3 Security Requirements

To state the EROS security requirements we must be able to determine the set of entities that a given subsystem has mutated during the course of an execution. Similarly, we have to define the entities a subsystem has read from. We define a function $\mathbf{mutated}(\cdot)$ with the intended meaning that if $S_0 \xrightarrow{\alpha_1} S_1 \xrightarrow{\alpha_2} \dots S_n$ is an execution, and $E \subseteq \mathcal{R}$, then $\mathbf{mutated}_E(S_0 \dots S_n)$ is the set of entities that have been affected by the entities E in the execution.

Intuitively, we treat mutation as a disease that spreads through the system. If we are concerned with the subsystem $E \subseteq \mathcal{R}$, then initially just the members of E are mutated. However, if any mutated entity modifies another resource (such as by calling it), then the other entity becomes mutated itself. Also, if a resource happens to read information from a mutated resource, the reader is mutated. Once mutated, a resource is forever mutated. Mutation *is* information flow.

We use the expected auxiliary definitions for $\mathbf{wroteto}(\cdot)$ and $\mathbf{readfrom}(\cdot)$ which for each action indicate which resources the executing process could have affected or been affected by, respectively. Their definitions are given in Appendix B.

The notion of a subsystem having read the values of other resources can be considered the inverse of mutation: instead of information flowing out from the subsystem in question, it is flowing inward. Therefore, we define the **read** function in terms of **mutated**: if x mutated y, then y must have read from x.

Definition. *If e is an execution of a system whose initial state is S_0, and $E \subseteq {S_0}^{\mathbf{existed}}$, then*

$$\mathbf{read}_E(e) = \{x | \mathbf{mutated}_{\{x\}}(e) \cap E \neq \emptyset\}$$

4.4 Access Control

The EROS capability system defines a particular access control mechanism. This mechanism determines what information flow is possible between system resources. In this section we formalize this mechanism, so that later we can verify:

Figure 5 The $\mathbf{mutated}(\cdot)$ relation

If $S_0 \xrightarrow{(p_1,\alpha_1)} \ldots S_n$ is an execution, $E \subseteq S_0^{\mathbf{existed}}$, and $e_i = S_0 \xrightarrow{(p_1,\alpha_1)} \ldots S_i$ are subexecutions, then

$$\begin{aligned}
\mathbf{mutated}_E(S_0) &= E \\
\mathbf{mutated}_E(e_1) &= E \\
& \cup \begin{cases} p & \text{if } E \cap \mathbf{readfrom}(\alpha) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
& \cup \begin{cases} \mathbf{wroteto}(\alpha) & \text{if } p \in E \\ \emptyset & \text{otherwise} \end{cases} \\
\mathbf{mutated}_E(e_n) &= \mathbf{mutated}_{\mathbf{mutated}_E(e_{n-1})}(S_{n-1} \xrightarrow{(p_n,\alpha_n)} S_n)
\end{aligned}$$

The definitions of $\mathbf{readfrom}(\alpha)$ and $\mathbf{wroteto}(\alpha)$ are given in Appendix B.

- that these mechanisms correspond to the actual behavior of the system, and
- that the algorithms used by the system services are correct with respect to the access control mechanism, in that they enforce the confinement policy.

We are particularly interested in deriving meaningful, conservative approximations of the set of entities that a given subsystem might be able to mutate, and the set of entities that the subsystem might gain information about. Formally, if $S \in \mathcal{S}$, and $E \subseteq \mathcal{R}$, the intended meaning of $\mathbf{mutable}_S(E)$ (resp. $\mathbf{readable}_S(E)$) is the set of entities that E directly or indirectly mutates (resp. reads) from some execution beginning with state S. Note that the mutable and readable functions are parameterized by a single state – the operating system has to be able to make these judgments based only on the current system state, unlike the requirements which can state what happens during an entire execution.

The weak attribute, while essential to the architecture, introduces significant complication. Intuitively, one can draw a directed, labeled graph showing the relationship between all the resources in the system. Since all interactions between resources occur via capabilities, and capabilities cannot be forged, a graph traversal can be done to compute whether any given resources can affect each other. This is similar to a transitive closure, with the following added complexities:

- The resource relationships are restricted by capability attributes.
- Weak capabilities modify the attributes of the capabilities that are fetched through them.
- Some capabilities result in two-way interactions between resources, while others are one-way.

We will represent the relationship between two resources as an element in a complete partial order (Figure 6). As notation, let $\top = \emptyset \in \mathbf{Attr}_\perp$. If $A \subseteq \mathbf{Attr}_\perp$, let the least upper bound of A, $\mathbf{lub}(A)$, be defined in the usual manner.

Figure 6 The complete partial order of attributes

Let $\mathbf{Attr}_\perp$ be the cpo where $\mathbf{Attr}_\perp = \mathbf{Attr} \cup \{\perp\}$ and $\leq$ is defined by

$$\begin{aligned}
\perp \leq x & \quad \forall x \in \mathbf{Attr}_\perp \\
x \supseteq y \implies x \leq y & \quad \forall x, y \in \mathbf{Attr}
\end{aligned}$$

The access relationship from x_1 to x_2 is the least upper bound of the attributes of the capabilities that x_1 might be able to obtain to x_2. This relationship is *not* symmetric. This representation relies on the fact that if k_1 and k_2 are capabilities which are identical except for possibly having different attributes, then $\mathbf{attribute}(k_1) \leq \mathbf{attribute}(k_2)$ means that any operation that can be performed using k_1 can also be done using k_2.

We first define the *direct access relation* between resources, which describes the relationship between resources in a particular state. Using this, we will define the *potential access relation* which defines the relationships that might exist in the future.

There are two oddities in the definition of $\mathbf{DirAcc}_S$. First, if $n \in \mathcal{R}$ has a start or resume capability k, then n is related to $\mathbf{target}(k)$, despite the fact that only a process can invoke such a capability. Second, this relationship is symmetric: $\mathbf{target}(k)$ is also related to n. The intuition behind this is that if x is related to y, then any process that has access to x can obtain access of the appropriate kind to y. If x has a start (or resume) capability to y, then a process that can access x might *call* that capability, leaving y with a resume capability back to that process. Thus, a start or resume capability means that there is an implicit backwards relationship from y to x.

Definition. *If $S \in \mathcal{S}$, then we define $\mathbf{DirAcc}_S : \mathcal{R} \times \mathcal{R} \to \mathbf{Attr}_\perp$ by*

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

where

$$\begin{array}{rll} \mathbf{DASet} = & \{(c, \mathbf{target}(k), \mathbf{attribute}(k)) & | c \in \mathbf{Node}, k \in S^{\mathbf{node}}(c)\} \\ \cup & \{(p, \mathbf{target}(k), \mathbf{attribute}(k)) & | p \in S^{\mathbf{exist}}, k \in S^{\mathbf{proc}}(p)\} \\ \cup & \{(\mathbf{target}(k), c, \top) & | c \in \mathbf{Node}, k \in S^{\mathbf{node}}, k \in \mathbf{StartCap} \cup \mathbf{ResumeCap}\} \\ \cup & \{(\mathbf{target}(k), p, \top) & | p \in S^{\mathbf{exist}}, k \in S^{\mathbf{proc}}, k \in \mathbf{StartCap} \cup \mathbf{ResumeCap}\} \end{array}$$

We construct $\mathbf{PotAcc}_S$ by using every capability indicated in $\mathbf{DirAcc}_S$ to fetch every possible capability from $\mathbf{DirAcc}_S$, obtaining a new, stronger relationship, and then repeating:

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*

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

and $\mathbf{combine}(\cdot)$ is defined in Figure 7.

Figure 7 The combine function

If A is an Agape resource relationship, then $\mathbf{combine}(A) : \mathcal{R} \times \mathcal{R} \to \mathbf{Attr}_\perp$ is defined to be:

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

where

$$\mathbf{transAccess}(x, y, z) = \begin{cases} \perp & \text{if } A(x, y) = \perp \\ A(y, z) & \text{if } A(x, y) \geq \{\mathbf{readonly}\} \\ \mathbf{rweaken}(s) & \text{if } \mathbf{weak} \in A(x, y) \end{cases}$$

$$\mathbf{rweaken}(s) = \begin{cases} \{\mathbf{readonly}\} & \text{if } s \in \mathbf{Page} \\ \{\mathbf{readonly}, \mathbf{weak}\} & \text{if } s \in \mathbf{Node} \\ \perp & \text{otherwise} \end{cases}$$

The $\mathbf{combine}(\cdot)$ function uses the auxiliary function $\mathbf{transAccess}$. If x is related to y and y is related to z, what is the relation between x and z? $\mathbf{transAccess}(\cdot)$ says that if x is related to y with a weak key, then x can only obtain a weakened authority to z (which depends upon the type of z). Otherwise, x can obtain from y the complete access to z.

Lemma. *The definition of $\mathbf{PotAcc}_S$ is well-defined. That is, the sequence $T_0, T_1, \ldots$ converges.*

Finally, with $\mathbf{PotAcc}_S$ we can define **mutable** and **readable** (Figure 8).

Figure 8 The $\textbf{mutable}(\cdot)$ and $\textbf{readable}(\cdot)$ relations

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

$$\begin{aligned}\textbf{mutable}_S(E) &= \{y | \exists x \in E, \textbf{PotAcc}_S(x,y) \subseteq \{\textbf{weak}\}\} \\ \textbf{readable}_S(E) &= \{x | E \cap \textbf{mutable}_S(\{x\}) \neq \emptyset\}\end{aligned}$$

4.5 Verification Proof

We can now state the major theorem (Figure 9). In any execution of the system, anything that was actually mutated or read by a subsystem was considered mutable or readable by the operating system. A subtle point now arises. During the execution processes may have been created. Since the operating system had no way of anticipating such creations, it couldn't have stated anything about the effects of these processes.

In many operating systems, such as UNIX, this would be an insolvable problem: new processes are intrinsically created with shared mutable access to many existing entities (the file system, processes, memory descriptor name space). Such a system cannot maintain confinement. In EROS, new processes can only be created by existing processes and can only have a subset of the authority of their creators. For this reason, we can restrict our consideration to only those resources that existed at the time the **mutable/readable** relations were computed: new processes cannot have any additional power than those that existed previously.

We first define the set of resources that exist at a given state of the system, and then use this to state our main theorem.

Definition. *If S is a state, then $S^{\textbf{existed}}$ is defined to be the following subset of $\mathcal{R}$:*

$$S^{\textbf{existed}} = \mathcal{R} - \mathcal{P} \cup S^{\textbf{exist}} \cup S^{\textbf{dead}}$$

Figure 9 Main verification theorem

Theorem (Main Theorem). *If $S_0 \xrightarrow{\alpha_1} S_1 \ldots \xrightarrow{\alpha_n} S_n$ is an execution, then for any $E \subseteq {S_0}^{\textbf{existed}}$,*

$$\begin{aligned}\textbf{mutated}_E(S_0 \xrightarrow{\alpha_1} \ldots S_n) \cap {S_0}^{\textbf{existed}} &\subseteq \textbf{mutable}_{S_0}(E) \\ \textbf{read}_E(S_0 \xrightarrow{\alpha_1} \ldots S_n) \cap {S_0}^{\textbf{existed}} &\subseteq \textbf{readable}_{S_0}(E)\end{aligned}$$

The principle lemmas that allow us to prove this theorem are shown in Figure 10. These lemmas state the essential properties of EROS which account for its security. The **Execution Reduces Authority** lemma states that the power that a subsystem can obtain only decreases during an execution: the subsystem can lose capabilities, but cannot create any from thin air. The **Mutation Implies Mutable** lemma states that if a resource becomes mutated by an operation, then it must have previously been mutable.

Figure 10 Major system properties

Principle Lemma 1 (Execution Reduces Authority). *If $S_0 \xrightarrow{(p,\alpha)} S_1$, then for all E,*

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

Principle Lemma 2 (Mutation Implies Mutable). *If $S_0 \xrightarrow{(p,\alpha)} S_1$, then for all E,*

$$\textbf{mutated}_E(S_0 \xrightarrow{(p,\alpha)} S_1) - E \subseteq \textbf{mutable}_{S_0}(E)$$

With these properties, the main theorem follows.

Proof Outline (Main Theorem). *We proceed by induction on the value of n. The base case is trivial.*

In the induction step, let e_i denote the execution $S_0 \xrightarrow{(p_1,\alpha_1)} \ldots \xrightarrow{(p_i,\alpha_i)} S_i$ Assume that for all sets F,

$$\mathbf{mutated}_F(e_{n-1}) \cap S_0{}^{\mathbf{existed}} \subseteq \mathbf{mutable}_{S_0}(F)$$

We want to show

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

This follows because:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{mutated}_E(e_n) \cap S_0{}^{\mathbf{existed}}} \\
= & \mathbf{mutated}_{\mathbf{mutated}_E(e_{n-1})}(S_{n-1} \xrightarrow{(p_n,\alpha_n)} S_n) \cap S_0{}^{\mathbf{existed}} & \\
\subseteq & (\mathbf{mutable}_{S_{n-1}}(\mathbf{mutated}_E(e_{n-1})) \cup \mathbf{mutated}_E(e_{n-1})) \cap S_0{}^{\mathbf{existed}} & \textit{by Lemma 2} \\
\subseteq & (\mathbf{mutable}_{S_{n-1}}(\mathbf{mutated}_E(e_{n-1})) \cup \mathbf{mutable}_{S_0}(E)) \cap S_0{}^{\mathbf{existed}} & \textit{by induction hypothesis} \\
\subseteq & ((\mathbf{mutable}_{S_0}(\mathbf{mutated}_E(e_{n-1})) \cap S_0{}^{\mathbf{existed}}) \cup \mathbf{mutable}_{S_0}(E)) & \textit{by Lemma 1} \\
 & \quad \cap S_0{}^{\mathbf{existed}} & \\
\subseteq & \mathbf{mutable}_{S_0}(\mathbf{mutable}_{S_0}(E)) \cup \mathbf{mutable}_{S_0}(E) & \textit{induction hypothesis,} \\
 & & \textit{mutable() is monotonic} \\
= & \mathbf{mutable}_{S_0}(E) & \textit{by definition}
\end{array}$$

The proof that **read** *and* **readable** *are related follows easily from the definitions.*

Finally, we must show that the EROS constructor mechanism satisfies its requirements. It constructs a new process p from the set of capabilities that it is given, and has a set of authorized entities **authorized**. If S is the current state, then for all executions e from S, it must be the case that

$$\mathbf{mutated}_{\{p\}}(e) \subseteq \{p\} \cup \mathbf{mutable}_S(\mathbf{authorize})$$

In other words, the client has stated permission for information to flow via the authorized entities – there should be no more.

With the main theorem, this is easy to show. The constructor is very conservative – the capabilities it considers as "safe" are those which will add no elements to $\mathbf{mutable}(p)$. The only other capabilites it gives p are the authorized ones, so the statement of corrections follows immediately from the theorem.

5 Conclusion

We have specified the security requirements and operations of a real operating system, and provided a formal definition for one security policy: confinement. We have developed a methodology and proof structure for this policy, and shown that it is enforced. This methodology generalizes to information flow problems in any capability-based architecture.

Capabilities provide two characteristics that are essential to the proof structure we have adopted. First, they combine denotation and access rights into a single entity, which allows straightforward construction of the mutable and readable relations. Second, they are unforgeable, which guarantees that this construction is closed. An equivalent security analysis for ACL-based systems would be more complex: modifications to a given resource's access control list have non-local consequences in the accessibility graphs, and may violate the closure. Changes to the permissions on a UNIX file, for example, can alter the rights of every current and future process. Virtual machines and programming language approaches have some potential for addressing these kinds of security issues.

The **Execution Reduces Authority** lemma provides a powerful simplifying tool in analyzing security issues. Both the statement of requirements and the corresponding proof are drastically more complicated in systems that permit amplification of authority. Indeed, this lemma seems worth adopting as a basic principle of operating system design.

Verifications such as this one have considerable practical utility for implementors. Agape's operational semantics was constructed by reducing the behavior of a real system to a manageable collection of primitives. Constructing the operational semantics revealed an implementation error in the real system. It also enabled the architecture to be significantly improved by providing a clear identification of those aspects of the semantics that were truly essential to security.

Previous work has claimed that pure capability systems can not support confinement [Kar88]. We have refuted this, and related work on the EROS implementation has demonstrated that real implementations of capability systems can be made to perform acceptably [Sha96c, Sha96b].

Our thanks to Carl Gunter, who provided comments and feedback on this paper at various stages. Norm Hardy first suggested the need for this proof and its feasibility. Jonathan Smith and Insup Lee provided ongoing advice, guidance, and feedback on this work as it developed.

References

[Den66] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations" in *Communications of the ACM*, vol. 9, pp. 143–154, March 1966.

[Har85] Norman Hardy. "The KeyKOS Architecture" *Operating Systems Review*. Oct. 1985, pp 8-25.

[Kar88] Paul Karger. *Improving Security and Performance for Capability Systems*, Technical Report No. 149, University of Cambridge Computer Laboratory, October 1988 (Ph. D. thesis).

[Key86] Key Logic, Inc. *U.S. Patent 4,584,639: Computer Security System.*

[Lam73] Butler W. Lampson. "A Note on the Confinement Problem." *Communications of the ACM* Vol 16, No 10, 1973

[Lan93] William S. Frantz and Charles R. Landau, "Object Oriented Transaction Processing in the KeyKOS Microkernel", *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, USENIX Association, September 1993.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press. 1984.

[Neu80] 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.* Computer Science Laboratory Report CSL-116, Second Edition, May 7, 1980, SRI International.

[Sha96a] Jonathan S. Shapiro. *The EROS Object Reference Manual.* In progress. Draft available via the EROS home page at
`http://www.cis.upenn.edu/~eros`

[Sha96b] Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. "State Caching in the EROS Kernel – Implementing Efficient Orthogonal Persistence in a Pure Capability System", *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, N.J. 1996

[Sha96c] Jonathan S. Shapiro, David J. Farber, Jonathan M. Smith. "The Measured Performance of a Fast Local IPC". *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, Seattle, Washington, November 1996, IEEE

[SW97a] J. S. Shapiro, S. Weber, *A Proof of Correctness for the EROS Confinement Mechanism* CIS Technical Report *???*, University of Pennsylvania (number to be assigned).

A Operational Semantics

All components of S' are assumed to be the same as in S unless stated otherwise.

α	semantics
$\mathbf{write}(k)$	$--$
$\mathbf{read}(k)$	$--$

$\mathbf{fetch}(k, o)$	if $\mathbf{weak} \in \mathbf{attribute}(k)$ then
	$o' = \mathbf{weaken}(o)$
	else
	$o' = \{o\}$
	end
	$S'^{\mathbf{proc}} = S^{\mathbf{proc}}[p \rightarrow (S^{\mathbf{proc}}(p) \cup o')]$

$\mathbf{store}(k, o)$	$S'^{\mathbf{node}} = S^{\mathbf{node}}[\mathbf{target}(k) \rightarrow S^{\mathbf{node}}(\mathbf{target}(k)) \cup \{o\}]$
$\mathbf{capremove}(k, o)$	$S'^{\mathbf{node}} = S^{\mathbf{node}}[\mathbf{target}(k) \rightarrow S^{\mathbf{node}}(\mathbf{target}(k)) - \{o\}]$

$\mathbf{call}(k, a)$	let $p' = \mathbf{target}(k)$ in
	if ($k \in \mathbf{StartCap}$ and $p' \in S^{\mathbf{avail}}$) or $k \in \mathbf{ResumeCap}$ then
	$S'^{\mathbf{run}} = S^{\mathbf{run}} - \{p\} \cup \{p'\}$
	$S'^{\mathbf{avail}} = S^{\mathbf{avail}} - \{p'\}$
	$S'^{\mathbf{wait}} = S^{\mathbf{wait}} - \{p'\} \cup \{p\}$
	$s'' = S^{\mathbf{proc}}[p' \rightarrow S^{\mathbf{proc}}(p') \cup a \cup \{\mathbf{ResumeCap}(p)\}]$
	if $k \in \mathbf{ResumeCap}$ then
	$S'^{\mathbf{proc}} = \mathbf{erase}(s'', \{k\})$
	$S'^{\mathbf{node}} = \mathbf{erase}(S^{\mathbf{node}}, \{k\})$
	else
	$S'^{\mathbf{proc}} = s''$
	end
	end

$\mathbf{return}(k, a)$	let $p' = \mathbf{target}(k)$ in
	if ($k \in \mathbf{StartCap}$ and $p' \in S^{\mathbf{avail}}$) or $k \in \mathbf{ResumeCap}$ then
	$S'^{\mathbf{run}} = S^{\mathbf{run}} - \{p\} \cup \{p'\}$
	$S'^{\mathbf{avail}} = S^{\mathbf{avail}} - \{p'\} \cup \{p\}$
	$S'^{\mathbf{wait}} = S^{\mathbf{wait}} - \{p'\}$
	$s'' = S^{\mathbf{proc}}[p' \rightarrow S^{\mathbf{proc}}(p') \cup a]$
	if $k \in \mathbf{ResumeCap}$ then
	$S'^{\mathbf{proc}} = \mathbf{erase}(s'', \{k\})$
	$S'^{\mathbf{node}} = \mathbf{erase}(S^{\mathbf{node}}, \{k\})$
	else
	$S'^{\mathbf{proc}} = s''$
	end
	end

$\mathbf{remove}()$	$S'^{\mathbf{proc}} = S^{\mathbf{proc}}[p \rightarrow (S^{\mathbf{proc}}(p) - k)]$
$\mathbf{create}(a)$	if $\mathcal{P} - S^{\mathbf{exist}} - S^{\mathbf{dead}} \neq \emptyset$ then let $p' \in \mathcal{P} - S^{\mathbf{exist}} - S^{\mathbf{dead}}$ in $S'^{\mathbf{avail}} = S^{\mathbf{avail}} \cup \{p'\}$ $S'^{\mathbf{proc}} = S^{\mathbf{proc}}[p' \rightarrow a][p \rightarrow S^{\mathbf{proc}}(p) \cup \{\mathbf{StartCap}(p')\}]$ end
$\mathbf{destroy}()$	$S'^{\mathbf{run}} = S^{\mathbf{run}} - \{p\}$ $S'^{\mathbf{dead}} = S^{\mathbf{dead}} \cup \{p\}$ $S'^{\mathbf{proc}} = \mathbf{erase}(S^{\mathbf{proc}}, \{\mathbf{StartCap}(p), \mathbf{ResumeCap}(p)\})$ $S'^{\mathbf{node}} = \mathbf{erase}(S^{\mathbf{node}}, \{\mathbf{StartCap}(p), \mathbf{ResumeCap}(p)\})$

(SEND is omitted due to space restrictions)

B The Readfrom and Writeto Relations

α	**readfrom**(α)	**wroteto**(α)	α	**readfrom**(α)	**wroteto**(α)
write(k)	-	**target**(k)	**call**(k, a)	-	**target**(k)
read(k)	**target**(k)	-	**return**(k, a)	-	**target**(k)
fetch(k, o)	**target**(k)	-	**remove**(k)	-	-
store(k, o)	-	**target**(k)	**create**(a)	-	-
capremove(k, o)	-	**target**(k)	**destroy**$()$	-	-