Verifying the EROS Confinement Mechanism*

Jonathan S. Shapiro Sam Weber

IBM T.J. Watson Research Center
shapj,samweber@us.ibm.com

Abstract

Capability systems can be used to implement higher-level security policies including the *-property *if a mechanism exists to ensure confinement. The implementation can be efficient if the "weak" access restriction described in this paper is introduced. In the course of developing EROS, a pure capability system, it became clear that verifying the correctness of the confinement mechanism was necessary in establishing the security of the operating system.*

This paper presents a verification of the EROS confinement mechanism with respect to a broad class of capability architectures (including EROS). We give a formal statement of the requirements, construct a model of the architecture's security policy and operational semantics, and show that architectures covered by this model enforce the confinement requirements if a small number of initial static checks on the confined subsystem are satisfied. The method used generalizes to any capability system.

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

1. Introduction

EROS [20] is a pure capability system that provides a mechanism for creating and instantiating confined subsystems: the **constructor**. This mechanism is part of the system's trusted computing base. EROS is a clean-room reimplementation of the KeyKOS architecture [2, 6] (originally called GNOSIS [3]). It builds on the KeySafe [14] design as the basis for mandatory security policy enforcement. The security enforcement provided by KeySafe is predicated on the correctness of the constructor.

* This work is a continuation of work started at the University of Pennsylvania. Portions of this were 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.

Boebert [1] and Karger [9] have argued that unmodified capability systems cannot enforce even basic mandatory access controls such as the **-property*. Both have proposed solutions in the form of hybrid protection architectures. Karger has also argued that unmodified capability systems cannot enforce confinement [8]. Given that EROS is a pure capability system, and that its security design rests on its ability to enforce confinement, a rigorous verification of the EROS confinement mechanism is necessary.

As described by Lampson [10], the confinement problem is to establish a runtime compartment satisfying two requirements. First, entities inside the compartment may transmit information to entities outside the compartment only via authorized channels of communication. Mutations inside the compartment therefore may not be observed outside the compartment unless explicitly authorized by the client. Second, entities outside the compartment may not inspect entities inside the compartment without their consent. We define a *confinement policy* that ensures these requirements by testing whether a set of initial static preconditions is satisfied:

1. All capabilities initially held by the confined subsystem either (a) convey no mutate authority, (b) are authorized channels, or (c) are capabilities to constructor that (recursively) instantiate confined subsystems. Since no unauthorized mutations are possible, no external observations of unauthorized mutations are possible.

2. No entity outside the confinement boundary holds read authority to any entity within the newly instantiated confined subsystem.

We will show that this policy *can* be enforced in capability systems, and can be enforced efficiently if a new primitive access right is introduced.

This paper does *not* address the issue of covert channel suppression. While important, reducing such channels is largely orthogonal to restricting explicit information flow, and is of interest only when it has first been shown that overt channels have been closed. Similarly, this paper does

not consider issues such as off line data forensics or modification of the store by a party with physical access to the machine.

KeySafe enforces mandatory security policies across confined compartments. Communication between compartments is mediated by a user-level reference monitor, which relies on knowledge of the semantics of the "primitive" objects to decide what operations and authority transfers are authorized. To allow later revocation, the reference monitor inserts transparent forwarding objects in front of all capabilities that are allowed to pass from one compartment to another. Implementing the reference monitor outside the kernel allows policies to be updated as new primitive object types are introduced, and facilitates adaptation of mandatory policies to the needs of particular applications.

In this paper we show that the confinement test used by the EROS constructor mechanism is correct. That is, we model a broad class of capability systems, and show that given a correct implementation of any capability system satisfying our model, the checks performed by the constructor certify that the newly instantiated service is properly sandboxed, and that (unless permitted by the service) the client cannot inspect the service. To verify that EROS can enforce confinement, we have developed a formal statement of requirements and a simplified model, *SW*, that is both strictly more powerful and more general than the EROS architecture. *SW* treats the OS-provided primitive operations as a serializably concurrent language whose operations are the kernel calls. We have modeled EROS's security policy, access control mechanisms, and operational semantics, and have shown that the EROS semantics satisfies the requirements of confinement.

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 [18, 21]). Some related work is discussed in Section 5. Finally, we discuss the implications of this work and its effect on the original system architecture and design.

2. EROS and the Constructor

EROS [20] is a pure capability system. The architecture incorporates what has come to be the conventional nonbuffering interprocess communication primitives [12, 19], with a novel twist providing for "at most one" reply. Metadata such as mapping structures that are typically considered part of the operating system are exported by EROS via capability-controlled structures. Both address translation and process state structures are stored in **nodes** (the EROS term for c-lists). Data and capabilities are partitioned to prevent forgery.

For purposes of verification, the essential EROS resource types are processes, data pages, capability pages, and nodes.[1] A **process** names an address space and executes the program in that address space in the usual way. Load and store operations require that data go to **data pages** and capabilities to **capability pages** and **nodes**. 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.

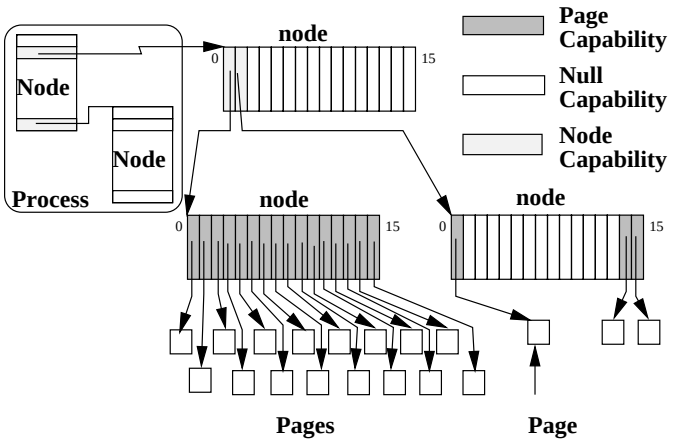

Figure 1. An EROS process

The EROS object and protection model are extended to secondary storage by transparent persistence [11]. 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 kernel-implemented file system; if file-like behavior is required by an application, a process is constructed that implements file behavior.

2.1. The Weak Attribute

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. The

[1] In the interest of brevity, we have omitted from this paper several capabilities and attributes whose semantics are a subset of some other object we have included. We have also omitted explicit consideration of device drivers (which are presumed to be holes) and certain capabilities such as "number capabilities" that convey no authority. Details of these can be found in [17, 18, 21].

weak attribute enforces *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 first reduced in authority by marking them read-only and weak. The read-only restriction ensures that the immediate object cannot be modified. The weak restriction ensures that read-only semantics will be enforced transitively through subsequent capability fetches. The weak attribute is a generalization of the KeyKOS [6] sense capability.

2.2. Constructors

The **constructor** is a trusted EROS application that builds new program (or subsystem) instances and certifies whether these instances are confined. The developer (or installer) of an application obtains a constructor that initially holds no capabilities, and installs those capabilities that an instance of the application 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*
- It is a read-only, weak capability, *or*
- It is a capability to a constructor that (recursively) generates confined products.

If the capability is *not* safe, the constructor records it in a set of known **holes**. When all desired capabilities have been installed, the builder then "freezes" the constructor, after which it will accept no further capabilities. Once frozen, all holes are known to the constructor, and the constructor can therefore determine whether the program instances it creates are confined.

When a client wishes to create a new instance of a subsystem, such as a file or a sort utility, it presents a set of authorized capabilities to the constructor and asks if the constructor output (the "yield") is confined modulo those capabilities. If $\mathbf{holes} \subseteq \mathbf{authorized}$, the product is confined. The intuition is this: there is no way to cause damage via safe capabilities, and the user has authorized all of the others. Because capabilities to safe constructors are safe, the constructor mechanism permits relatively complex subsystems to be built behind a confinement boundary.

Weak capabilities allow a capability pointing to a complex structure to be added safely without requiring that the structure be traversed. Safety might also be accomplished using a "deep copy" of the structure, but the weak access restriction makes the copy unnecessary, and thereby allows a dominating compartment to be granted weak access to content in a lesser compartment without the need for further mediation. It also allows multiple readers to have consistent read-only access to a data structure that is being updated.

3. Verification Requirements and Modeling Method

To verify the constructor mechanism, we must construct *SW*, a model of the EROS system, specify the requirements of confinement in terms independent of both the access control model and the operational semantics, and describe the access control model. Finally, we must show that the system's security policies implement its requirements.

System Model Because EROS is persistent, a natural level of analysis is to examine operations on nodes and pages. The only additional state saved by the EROS checkpoint mechanism is the list of currently running processes at the time of the checkpoint. All operations performed by the kernel consist of modifications to nodes and pages.

SW may safely deviate from the actual system provided that *SW* is at least as powerful as EROS. Any behavior that could occur in EROS must be possible in *SW*. If this is so, and if no information can flow out of a confined subsystem in *SW*, then the same will hold true in EROS. For modeling purposes, we have made several departures from the actual system:

1. The model uses permissions rather than restrictions. This is merely a switch from subtractive to additive logic. While the model is richer than the system, every capability permission existing in the real system can be captured using the permissions of the model.

2. In the model, no distinction is made between nodes, pages, and processes. Note that processes are strictly more powerful than either nodes or pages, and that *SW* is therefore more powerful than EROS as a result of this simplification.

3. The real system uses objects of bounded size. The model uses sets, both to avoid dependency on any particular object size and to generalize the model to cover similar systems.

4. EROS keeps capabilities and data in objects of distinct type. *SW* allows objects to contain both, but the operational semantics has no operations that allow them to be interconverted. One may imagine the model objects to consist of a data part and a capability part. The EROS system represents the subcase in which either the data or the capability part of each object is required to be empty.

5. EROS has process states, three invocation operations, and a special capability type supporting return-once semantics. In this paper, we have simplified this to a single, more general invocation type and omitted the

special capability type. Earlier versions of the operational semantics and proof have included both [18, 21]. Their omission here is for brevity and clarity.

6. Reading a page in EROS produces the value that was last written. *SW* does not keep track of the contents of pages. Ignoring page state results in a more powerful model: processes are maximally hostile. Finally, we demonstrate that the security of EROS does not rely upon values computed by applications.

7. The EROS storage allocator is implemented by trusted user-level software. In *SW*, its behavior is represented by the $\mathbf{create}()$ and $\mathbf{destroy}()$ operations. A complication potentially arises from the fact that the real system reallocates objects. When an object is destroyed, the allocator reinitializes the object and invalidates all capabilities to that object by incrementing its version number. Since there is a finite bound on version numbers, and each version is allocated at most once, we consider each version of an EROS object to be a distinct *SW* object for modeling purposes.

8. Real applications obey sequential programs, and their behavior is highly predictable. *SW* instead assumes that the behavior of a program is completely nondeterministic. If it is possible for a program to perform a particular operation, *SW* assumes that it does so. This means that the verification's correctness does not rely on peculiarities of instruction sequencing.

Each of these changes makes *SW* more powerful than EROS.

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

Similarly, we define the function $\mathbf{read}_E(\vec{e})$ to be the set of entities in the system (confined or otherwise) that E read (directly or indirectly) in the course of the execution. This definition includes both entities read directly via an operation such as a load, and also entities whose value is "observed" during the course of the computation.

Access Control While the confinement problem requirements are stated in terms of the execution of the system, it is desirable for the corresponding security policy to be verifiable based on a modest set of initial conditions. This eliminates the need to dynamically interpret the actions of the confined subsystem. 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 To show that the system's security policies implement its requirements, we must show that if $\vec{e}$ is an execution of the system whose initial state is S_0, and if E is any set of resources, then

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

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

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

$$S_0 \xrightarrow{\alpha} S_1$$

with the intended meaning "When the system was in state S_0, action/kernel call α occurred, which resulted in state S_1." There is no strong distinction in *SW* between processes and objects; a process is simply an object that initiates an action. The term "process" is used solely as a descriptive convenience to distinguish actor from actee.

4.1. Semantic Entities of *SW*

Figure 2 gives the basic semantic entities of *SW*. The system resources consist of objects. If c is a capability, $\mathbf{target}(c)$ is the resource that capability refers to, and $\mathbf{rights}(c)$ is that capability's access rights. Access rights are represented as a subset of $\mathcal{R} = \{\mathbf{wk}, \mathbf{rd}, \mathbf{wr}, \mathbf{exec}\}$. The $\mathbf{rd}$, $\mathbf{wr}$ and $\mathbf{exec}$ rights correspond to the conventional read, write, and execute permissions. The effect of $\mathbf{wk}$ permission is to ensure transitive read-only access as described in Section 2.2. Use of permissions rather than restrictions departs from EROS, but we have found that it makes the model easier to understand and work with. The

Universal Sets

$\mathbf{Object}$	the set of objects
$\mathbf{Caps}$	set of capabilities
$\mathcal{R}$	set of access rights
$\mathcal{S}$	set of system states
$\mathcal{A}$	set of $\mathbf{Object} \times operations$

Capability Types

$\mathbf{ObCap}$	the set of object capabilities

Utility Functions

$\mathbf{target}(\cdot)$	$\mathbf{Caps} \to \mathbf{Object}$
$\mathbf{rights}(\cdot)$	$\mathbf{Caps} \to 2^{\mathcal{R}}$
$\mathbf{actor}(\cdot)$	$\mathcal{A} \to \mathbf{Object}$

Figure 2. Semantic entities

System state: $S = (S^{\mathbf{exist}}, S^{\mathbf{caps}}, S^{\mathbf{dead}}) \in \mathcal{S}$

$S^{\mathbf{exist}} \subseteq \mathbf{Object}$	the set of live objects
$S^{\mathbf{caps}} : \mathbf{Object} \to 2^{\mathbf{Caps}}$	the map between objects and sets of capabilities
$S^{\mathbf{dead}} \subseteq \mathbf{Object}$	the set of all objects which have been destroyed

Sanity conditions:

1. $S^{\mathbf{exist}}$ and $S^{\mathbf{dead}}$ are disjoint
2. $c \in \bigcup_{o \in \mathbf{Object}} S^{\mathbf{caps}}(o) \implies \mathbf{target}(c) \in S^{\mathbf{exist}}$

Figure 3. System state

function $\mathbf{actor}(\cdot)$ associates a unique process with each action. We use $\mathbf{ObCap}$ as a constructor: if $o \in \mathbf{Object}$, then $\mathbf{ObCap}(o, \{\mathbf{wk}\})$ is the unique weak capability that refers to o.

The definition of *SW* state is shown in Figure 3. The sanity conditions ensure that we do not have to deal with nonsensical capabilities. The set of dead objects exists so that we can ensure that newly created objects are distinct from any that have been destroyed. We treat processes as completely non-deterministic, able to invoke any capabilities that they have in their possession.

The set of possible actions in the model are enumerated in Figure 4. We assume that each action is an operation performed by some process. The restrictions on the actions indicate the required access rights to perform the operation.

4.2. Operational Semantics

We define the transition relation $\xrightarrow{\cdot} \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ as follows:

$$S \xrightarrow{\alpha} S'$$

where the state transformation performed by each action α is given in Figure 4. The data 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.

The function $\mathbf{restrict}(c, r)$ returns a copy of the capability c in which all access rights in r have been removed from $\mathbf{rights}(c)$.

$$\mathbf{restrict}(c, r) = \mathbf{ObCap}(\mathbf{target}(c), \mathbf{rights}(c) \cap r)$$

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.

Lemma 1:
The transition relation $\xrightarrow{\cdot}$ is well-defined. That is, if $S \xrightarrow{\alpha} S'$, 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. As discussed in the requirements statement, we define a function $\mathbf{mutated}(\cdot)$ with the intended meaning that if $\vec{e} = S_0 \xrightarrow{\alpha_1} S_1 \xrightarrow{\alpha_2} \ldots S_n$ is an execution, and $E \subseteq \mathbf{Object}$, then $\mathbf{mutated}_E(\vec{e})$ is the set of entities that have been affected by the entities E in the execution (Figure 6). 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. 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 $\mathbf{read}$ function in terms of $\mathbf{mutated}$: if x mutated y, then y must have read from x.

4.4. Correctness of Access Control

The EROS system uses capabilities as its primitive protection mechanism. The access restrictions of these capabil-

Action Type	*Restrictions*	*Description*
$\mathbf{read}(p, c_0)$	$\{\mathbf{wk}, \mathbf{rd}\} \cap \mathbf{rights}(c_0) \neq \emptyset$	read data from an object
$\mathbf{write}(p, c_0)$	$\mathbf{wr} \in \mathbf{rights}(c_0)$	write data to an object
$\mathbf{fetch}(p, c_0, c_1)$	$\{\mathbf{wk}, \mathbf{rd}\} \cap \mathbf{rights}(c_0) \neq \emptyset$, $c_1 \in S^{\mathbf{caps}}(\mathbf{target}(c_0))$	fetch capability from an object
$\mathbf{store}(p, c_0, c_1)$	$\mathbf{wr} \in \mathbf{rights}(c_0), c_1 \in S^{\mathbf{caps}}(p)$	store capability to object
$\mathbf{remove}(p, c_0, c_1)$	$\mathbf{wr} \in \mathbf{rights}(c_0)$, $c_1 \in S^{\mathbf{caps}}(\mathbf{target}(c_0))$	remove capability from object
$\mathbf{invoke}(p, c_0, a)$	$\mathbf{exec} \in \mathbf{rights}(c_0)$	invoke a process
$\mathbf{create}(p, a)$	$a \subseteq S^{\mathbf{caps}}(p)$	create new object with capabilities a
$\mathbf{destroy}(p, c_0)$	$\mathbf{wr} \in \mathbf{rights}(c_0)$	object destruction

In all cases, $p \in S^{\mathbf{exist}}$, $c_0 \in S^{\mathbf{caps}}(p)$, $a \subseteq S^{\mathbf{caps}}(p)$.

α	semantics
$\mathbf{read}(p, c_0)$	$--$
$\mathbf{write}(p, c_0)$	$--$
$\mathbf{fetch}(p, c_0, c_1)$	if $\mathbf{rd} \in \mathbf{rights}(c_0)$ then $S'^{\mathbf{caps}}(p) = S^{\mathbf{caps}}(p) \cup \{c_1\}$ else if $\mathbf{wk} \in \mathbf{rights}(c_0)$ then $S'^{\mathbf{caps}}(p) = S^{\mathbf{caps}}(p) \cup \{\mathbf{restrict}(c_1, \{\mathbf{wk}\})\}$ end
$\mathbf{store}(p, c_0, c_1)$	$S'^{\mathbf{caps}} = S^{\mathbf{caps}}[\mathbf{target}(c_0) \rightarrow S^{\mathbf{caps}}(\mathbf{target}(c_0)) \cup \{c_1\}]$
$\mathbf{remove}(p, c_0, c_1)$	$S'^{\mathbf{caps}} = S^{\mathbf{caps}}[\mathbf{target}(c_0) \rightarrow S^{\mathbf{caps}}(\mathbf{target}(c_0)) - \{c_1\}]$
$\mathbf{invoke}(p, c_0, a)$	let $p' = \mathbf{target}(c_0)$ in $S'^{\mathbf{caps}} = S^{\mathbf{caps}}[p' \rightarrow S^{\mathbf{caps}}(p') \cup a \cup \mathbf{ObCap}(\mathbf{target}(c_0), \{\mathbf{exec}\})]$
$\mathbf{create}(p, a)$	let $o' \in \mathbf{Object} - S^{\mathbf{exist}} - S^{\mathbf{dead}}$ in $S'^{\mathbf{exist}} = S^{\mathbf{exist}} \cup \{o'\}$ $S'^{\mathbf{caps}} = S^{\mathbf{caps}}[o' \rightarrow a][p \rightarrow S^{\mathbf{caps}}(p) \cup \{\mathbf{ObCap}(o', \mathcal{R})\}]$
$\mathbf{destroy}(p, c_0)$	let $C = \bigcup_{r \in 2^{\mathcal{R}}} \mathbf{ObCap}(\mathbf{target}(c_0), r)$ in $S'^{\mathbf{exist}} = S^{\mathbf{exist}} - \{\mathbf{target}(c_0)\}$ $S'^{\mathbf{dead}} = S^{\mathbf{dead}} \cup \{\mathbf{target}(c_0)\}$ $S'^{\mathbf{caps}} = \mathbf{erase}(S^{\mathbf{caps}}, C)$

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

Figure 4. Operational Semantics of *SW*

α	$\mathbf{readfrom}(\alpha)$	$\mathbf{writeto}(\alpha)$	α	$\mathbf{readfrom}(\alpha)$	$\mathbf{writeto}(\alpha)$
$\mathbf{read}(p, c_0)$	$\{p, \mathbf{target}(c_0)\}$	$\{p\}$	$\mathbf{remove}(p, c_0, c_1)$	$\{p\}$	$\{\mathbf{target}(c_0)\}$
$\mathbf{write}(p, c_0)$	$\{p\}$	$\{\mathbf{target}(c_0)\}$	$\mathbf{invoke}(p, c_0, a)$	$\{p\}$	$\{\mathbf{target}(c_0)\}$
$\mathbf{fetch}(p, c_0, c_1)$	$\{p, \mathbf{target}(c_0)\}$	$\{p\}$	$\mathbf{create}(p, a)$	$\{p\}$	$\{p\}$
$\mathbf{store}(p, c_0, c_1)$	$\{p\}$	$\{\mathbf{target}(c_0)\}$	$\mathbf{destroy}(p, c_0)$	$\{p\}$	$\{p\}$

Figure 5. Definitions of $\mathbf{readfrom}(\alpha)$ and $\mathbf{writeto}(\alpha)$

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

$$\begin{aligned}
&\mathbf{mutated}_E(S_0) = E \\
&\mathbf{mutated}_E(\vec{e_1}) = E \\
&\quad \cup \begin{cases} \mathbf{actor}(\alpha_1) & \text{if } E \cap \mathbf{readfrom}(\alpha_1) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
&\quad \cup \begin{cases} \mathbf{wroteto}(\alpha_1) & \text{if } \mathbf{actor}(\alpha_1) \in E \\ \emptyset & \text{otherwise} \end{cases} \\
&\mathbf{mutated}_E(\vec{e_i}) = \\
&\quad \mathbf{mutated}_{\mathbf{mutated}_E(e_{i-1})}(S_{i-1} \xrightarrow{\alpha_i} S_i) \quad i > 1
\end{aligned}$$

$$\begin{aligned}
&\mathbf{read}_E(\vec{e}) = \\
&\quad \{x | \mathbf{mutated}_{\{x\}}(\vec{e}) \cap E \neq \emptyset\}
\end{aligned}$$

Where $\mathbf{readfrom}(\cdot)$ is defined as in Figure 5.

Figure 6. The $\mathrm{mutated}(\cdot)$ and $\mathrm{read}(\cdot)$ relations

ities determine what actions are directly possible between system resources at run time. To avoid the need for additional run-time interpretation, it is desirable to state security policies in terms of information flow expectations that can be statically expressed. The assumed correspondence between the expected information flow and the run time execution of the model must be verified. In this section we formalize some notions about access rights, so that later we can verify:

- 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 our 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 $E \subseteq \mathbf{Object}$, 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 access right, 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 access rights.
- Weak capabilities modify the rights of the capabilities that are fetched through them.
- Some capabilities result in two-way interactions between resources, while others are one-way.

We represent the relationship between two resources as an element in a complete partial order (Figure 7). If $A \subseteq \mathcal{R}_\perp$, let the least upper bound of A, $\mathbf{lub}(A)$, be defined in the usual manner.

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 c_0 and c_1 are capabilities that are identical except for possibly having different attributes, then $\mathbf{rights}(c_0) \leq \mathbf{rights}(c_1)$ means that any operation that can be performed using c_0 can also be done using c_1.

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

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

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

where

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

One detail of the definition of $\mathbf{DirAcc}_S$ deserves emphasis. If $o \in \mathbf{Object}$ has a capability c, then o is clearly related to $\mathbf{target}(c)$. However, if c conveys **exec** rights, this relationship is symmetric: $\mathbf{target}(c)$ is also related to o. The intuition behind this is that interprocess communication may authorize a reply. In our operational semantics,

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

$$x \subseteq y \implies x \leq y \quad \forall x, y \in \mathcal{R}$$

Figure 7. The complete partial order of attributes

a reply capability is explicitly synthesized by the $\mathbf{invoke}()$ operation.

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:

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

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

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

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

where

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

Lemma 2:
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 $\mathbf{mutable}$ and $\mathbf{readable}$ (Definition 1).

Definition 1 (The $\mathbf{mutable}(\cdot)$ and $\mathbf{readable}(\cdot)$ relations):
If $S \in \mathcal{S}$, *then*

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

Lemma 3:
For all $x \subseteq y \subseteq \mathbf{Object}$, *states* S,

$$\mathbf{mutable}_S(x) \subseteq \mathbf{mutable}_S(y)$$

4.5. Verification Proof

We can now state the major theorem (Figure 8). 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. The statement of the theorem is more subtle than one might have at first expected.

Theorem 1 (Main Theorem):
If $S_0 \xrightarrow{\alpha_1} S_1 \ldots \xrightarrow{\alpha_n} S_n$ *is an execution, then for any* $E \subseteq {S_0}^{\mathbf{existed}}$,

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

Figure 8. Main verification theorem

If new objects could not be created, then one could merely require that for any execution $\vec{e}$ proceeding from state S_0, $\mathbf{mutated}_E(\vec{e}) \subseteq \mathbf{mutable}_{S_0}(E)$ [and similarly for read].

What we wish to require of new objects is that no process can use such an object to amplify its authority. However, it would not be reasonable for the system to make any other *a priori* assumptions about the behavior of objects that do not exist. This is captured in our main theorem (Figure 8) by the fact that the mutated function tracks mutations that occur by means of new objects. The intersection with ${S_0}^{\mathbf{existed}}$ removes only the new objects from the set of changed objects, but not their effects on the remainder of the system.

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 2:
If S *is a state, then* $S^{\mathbf{existed}}$ *is defined to be the following subset of* $\mathbf{Object}$*:*

$$S^{\mathbf{existed}} = S^{\mathbf{exist}} \cup S^{\mathbf{dead}}$$

The lemmas that allow us to prove this theorem are shown in Figure 10. These lemmas state the essential properties of *SW* 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 new capabilities that increase its authority. The *number* of capabilities can grow, but the *implied authority* can only shrink. The **Mutation Implies Mutable** lemma states that if a resource becomes mutated by an operation, then it must have previously been mutable.

With these properties, the main theorem follows (Figure 9).

Finally, we must show that the EROS constructor mechanism satisfies the confinement requirements. The requester specifies at construction time a set of capabilities $\mathbf{authorized}$ that are permitted exceptions to the confinement boundary. The constructor builds a new process p from the set of capabilities that it is given by the builder. If S is the current state, then for all executions $\vec{e}$ from S,

We proceed by induction on the value of i. The base case is trivial.
In the induction step, let $\vec{e_i}$ denote the execution $S_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_i} S_i$. Assume that for all sets F,

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

We want to show

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

This follows because:

$$\begin{array}{lll}
\mathbf{mutated}_E(\vec{e}_n) \cap {S_0}^{\mathbf{existed}} & & \\
\quad = \quad \mathbf{mutated}_{\mathbf{mutated}_E(\vec{e}_{n-1})}(S_{n-1} \xrightarrow{\alpha_n} S_n) \cap {S_0}^{\mathbf{existed}} & & \\
\quad \subseteq \quad (\mathbf{mutable}_{S_{n-1}}(\mathbf{mutated}_E(\vec{e}_{n-1})) \cup \mathbf{mutated}_E(\vec{e}_{n-1})) \cap {S_0}^{\mathbf{existed}} & \text{by Lemma 5} & \\
\quad \subseteq \quad (\mathbf{mutable}_{S_{n-1}}(\mathbf{mutated}_E(\vec{e}_{n-1})) \cup \mathbf{mutable}_{S_0}(E)) \cap {S_0}^{\mathbf{existed}} & \text{by induction hypothesis} & \\
\quad \subseteq \quad ((\mathbf{mutable}_{S_0}(\mathbf{mutated}_E(\vec{e}_{n-1})) \cap {S_0}^{\mathbf{existed}}) \cup \mathbf{mutable}_{S_0}(E)) & \text{by Lemma 4} & \\
\qquad\qquad \cap {S_0}^{\mathbf{existed}} & & \\
\quad \subseteq \quad \mathbf{mutable}_{S_0}(\mathbf{mutable}_{S_0}(E)) \cup \mathbf{mutable}_{S_0}(E) & \text{induction hypothesis,} & \\
 & \text{mutable() is monotonic} & \\
\quad = \quad \mathbf{mutable}_{S_0}(E) & \text{by definition} &
\end{array}$$

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

Figure 9. Proof outline for main theorem

Lemma 4 (Execution Reduces Authority):
If $S_0 \xrightarrow{\alpha} S_1$, *then for all* E,

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

Lemma 5 (Mutation Implies Mutable):
If $S_0 \xrightarrow{\alpha} S_1$, *then for all* E,

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

Figure 10. Major system properties

confinement requires that

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

The constructor requires that $\mathbf{holes} \subseteq \mathbf{authorized}$, and therefore requires (by Lemma 3) that $\mathbf{mutable}_S(\mathbf{holes}) \subseteq \mathbf{mutable}_S(\mathbf{authorized})$. In other words, the client has stated permission for information to flow via the holes. Provided the non-holes do not leak information, the "yield" of the constructor is properly sandboxed.

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 capabilities it gives p are members of the authorized set, so the statement of correctness follows immediately from the theorem.

The encapsulation of the constructor "yield" follows from the fact that the requester initially holds no capabilities to the objects comprising the yield. When the yield first begins execution, only those capabilities contained in the holes represent possible leaks to the requester. Unless the yield consents to be inspected by transmitting capabilities through one of these holes, its encapsulation is assured.

5. Related Work

Noninterference: Rushby's discussion of intransitive non-interference [16] applies a similar kind of information flow analysis to the one used here, extending earlier work by Haigh and Young [5] and by Goguen and Meseguer [4]. That formulation is more general than the technique used here, but would require extension to deal with systems in which access rights are transferable. In particular, there is no provision in the Rushby formulation for the actions of the *step* function to revise the security assertions as access rights are legitimately propagated or mandatory constraints are revised.

PSOS: Neumann *et al.* constructed extensive proof synthesis mechanisms in connection with PSOS [13]. While a proof sketch of the security properties of this system was included in the report, no proof of correctness for confinement in the PSOS system has been published. The proof sketched in the report fails to demonstrate that the operational semantics of the system architecture actually satisfies the specification.

Mandatory Policy Enforcement: Boebert [1] and

Karger [9] show that pure capability systems cannot enforce the **-property*. While their conclusion is correct, capability systems *do* provide sufficient strength to construct mandatory policies at a higher level of abstraction with reasonable performance, as has been done in KeySafe [14].

Karger has also shown that unmodified capability systems cannot enforce the confinement policy [8]. The apparent discrepancy results from differences in term definition. Karger's confinement policy is a mandatory access control policy: "this piece of information must not be disclosed to that set of unauthorized parties." That is, it is a policy concerning the flow of information to subjects. Lampson's confinement problem [10] imposes a weaker constraint: information can flow out of the subsystem only through authorized *channels*. That is, in the Lampson definition the channels define an encapsulation boundary to be enforced.

KeySafe: The KeySafe system [14] implements a reference monitor that mediates interactions across compartments. Each compartment is a confined subsystem created by the reference monitor. Creating compartments in this fashion ensures that no unauthorized channels out of the compartment exist, and also that no inspection of the compartment contents is permitted.

Compartments in KeySafe can be used to encapsulate security levels, users, intersections of level and user, or any other entity that the security policy wishes to describe. The granularity of policy enforcement under KeySafe is the compartment rather than the process.

Where the security policy permits transfer of capabilities from one compartment to another, the reference monitor substitutes a capability to a transparent forwarding object to allow later revocation. This is similar to the indirection mechanism proposed by Redell [15]. Cross-compartment calls are relatively rare, and the overhead of this indirection (once established) is low. Intracompartment invocations incur no overhead from the reference monitor.

We believe that the KeySafe architecture can enforce both the **-property* and Karger's confinement policy, but this does not directly contradict their claims. KeySafe is a reference monitor built on top of a more primitive capability mechanism; such a reference monitor constitutes a modified capability system in the sense of Karger's discussion.

Memoryless Execution: Jones' dissertation [7] discusses computing a closure containing all protection states that can be derived from an initial protection state by the performance of "environment transforming functions." She demonstrates the enforcement of a number of security policies using this sort of transitive closure, including a "Memoryless Execution" policy closely related to our confinement policy. The memoryless execution policy allows the called procedure to modify its parameters, but does not allow modification of other objects. There appears to be no capability in Jones' model that names environments directly. Consequently, there is no need to consider transitively accessible mutation rights. Were such a capability introduced, some variant of the **wk** right described here would be required.

By permitting the client to authorize exceptions to the immutability constraint, the confinement policy defined here slightly extends the memoryless execution policy. By permitting recursive use of constructors that produce confined subsystems, the confinement policy enables a form of object-based software construction that is not possible under the memoryless execution constraint. Recursive construction enables EROS to bootstrap "copy on write" address spaces, and can therefore be used to perform object instance creation. The most common use of the EROS constructor is to create new process instances in this way.

Jones also proposes a rights amplification operation that must be handled with considerable care, lest the supposedly restricted subsystem gain mutate rights in unforeseen ways. Amplify operations appear to have been excluded from the systems verified in the thesis.

6. 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 many capability-based architectures. The constructor implementation performs the confinement check in no worse than $n \log n$ time, where n is the number of holes known to the constructor. Construction is significantly faster than the equivalent UNIX *fork* and *exec* combination [20].

The *SW* model faithfully captures EROS. This was not accidental; both EROS and KeyKOS intentionally provide as "pure" a model as the architects could contrive to build. Related work on the EROS implementation has demonstrated that a pure capability system of this form can be made to perform quite well [19, 20].

Verifications such as this one have considerable practical utility. *SW*'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. Best of all, it did so at modest cost.

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.

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.

Capability systems provide a natural framework for typed, protected objects. Component architectures such as COM and CORBA are increasingly used for critical systems. Applications constructed using these technologies may be viewed as subsystems connected by capabilities. We expect that forthcoming e-Commerce applications will be built using similar application frameworks. Therefore, security policy enforcement in capability systems is a critical area of future research.

7. Acknowledgements

Norm Hardy, the principal architect of KeyKOS, first suggested the need for this proof and its feasibility. Carl Gunter, Jonathan Smith, and Insup Lee of the University of Pennsylvania provided comments and feedback on this paper at various stages. Maria Ebling, Stephen Holton, Guerney Hunt, and Paul Karger of IBM also provided extensive feedback, as did Chris Okasaki of Columbia University.

References

[1] W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proc. 7th DoD/NBS Computer Security Conference*, pages 291–293, Gaithersburg, MD, USA, Sept. 1984. National Bureau of Standards.

[2] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112. USENIX Association, Apr. 1992.

[3] W. Frantz, C. Landau, and N. Hardy. Gnosis: A secure operating system for the '90s. *SHARE Proceedings*, 1983.

[4] J. A. Goguen and J. Meseguer. Inference control and unwinding. In *Proc. Symposium on Security and Privacy*, pages 75–86, Oakland, CA, USA, Apr. 1984. IEEE Computer Society.

[5] J. Haigh and W. Young. Extending the noninterference version of mls for sat. *IEEE Transactions on Software Engineering*, 13(2):141–150, Feb. 1987.

[6] N. Hardy. The KeyKOS architecture. *Operating Systems Review*, pages 8–25, Oct. 1985.

[7] A. K. Jones. *Protection in Programmed Systems*. PhD thesis, Carnegie-Mellon University, 1973.

[8] P. Karger. *Improving Security and Performance for Capability Systems*. PhD thesis, University of Cambridge, Oct. 1988. Technical Report No. 149.

[9] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proc. of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, Oakland, CA, Apr. 1984. IEEE.

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

[11] C. R. Landau. The checkpoint mechanism in KeyKOS. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE, Sept. 1992.

[12] J. Liedtke. Improving IPC by kernel design. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 175–188. ACM, 1993.

[13] 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 Report CSL-116, Computer Science Laboratory, may 1980.

[14] S. A. Rajunas. The KeyKOS/KeySAFE system design. Technical Report SEC009-01, Key Logic, Inc., Mar. 1989. http://www.cis.upenn.edu/~KeyKOS.

[15] D. D. Redell. *Naming and Protection in Extensible Operating Systems*. PhD thesis, University of California at Berkeley, 1974.

[16] J. R. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, Computer Science Laboratory, SRI International, Dec. 1992.

[17] J. S. Shapiro. *The EROS Object Reference Manual*.

[18] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, Philadelphia, PA 19104, 1999.

[19] J. S. Shapiro, D. J. Farber, and J. M. Smith. The measured performance of a fast local IPC. In *Proc. 5th International Workshop on Object Orientation in Operating Systems*, pages 89–94, Seattle, WA, USA, Nov. 1996. IEEE.

[20] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, Dec. 1999. ACM.

[21] J. S. Shapiro and S. Weber. Verifying operating system security. Technical Report MS-CIS-97-26, University of Pennsylvania, Philadelphia, PA, USA, 1997.