# The Practical Application of a Decidable Access Model*

SRL Technical Report SRL-2000-01

Jonathan S. Shapiro
*Johns Hopkins University*
Baltimore, MD 21218
shap@eros-os.org

November 6, 2000

**Abstract**

While the safety of a number of access models has been formally established, few of these models are reflected in real systems. Most currently deployed commodity systems are based on access models that have been formally proven either unsafe or undecidable. Models that are decidable and safe, such as *take-grant*, have been too weak to be practically useful.

This paper introduces *diminish*, a new fundamental rights weakening operation, and *diminish-take*, a new access model based on *take-grant* that incorporates this operation and models indirect reference. As with the *take-grant* model, safety in *diminish-take* is decidable in linear time. In certain cases it is decidable in constant time using only local information. Unlike *take-grant*, the new model enforces the **-property*.

We also describe the practical application of this access right in a fast capability system for commodity processors: EROS. We show that the *diminish* operation is a critical underpinning for an efficient realization of several design patterns, including confinement, protected subsystems, and user-supplied memory managers.

## 1 Introduction

The *safety problem* is the problem of determining whether, given an initial configuration of a protection system, a subject $s$ can obtain some access right $r$ on an object $o$. Safety is a precondition to almost all interesting security policies. Harrison *et al.* have shown that safety is undecidable in general, and false in the access models used in current commodity systems (access control lists, UNIX permissions) [HRU76].

Jones, Lipton and Snyder [JLS76, Sny77] have proposed *take-grant*, an access model for which safety is decidable in linear time. The *take-grant* model has limited practical utility for several reasons:

- It cannot enforce the **-property* [Boe84]. This has tended to eliminate it from consideration in potentially secure systems.
- Deciding safety requires a global scan of the access graph. In practical application, a locally-decidable access model is required.
- It models capability systems, which are unfamiliar to most students and industrial practitioners.
- There is a significant semantic gap between the model and real practice: the absence of indirection rights.
- It neither accounts for trusted processes nor accounts for why they are not necessary. This makes the model's application to microkernel designs unclear.

Further work by Bishop and Snyder [Bis79, Sny81] has shown that while *take-grant* is decidable, the transitive consequences of local access decisions can be far-reaching, and especially so in the face of conspiring applications such as Trojan horses.

In this paper, we introduce *diminish-take*, a revision of the *take-grant* model providing indirection and incorporating *diminish*, a new fundamental rights weak-

* 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, Tandem Computer, Intel, and IBM Corporations.

ening operation. As with *take-grant*, safety in the *diminish-take* model is decidable in linear time. In one useful special case, it can be decided in *constant* time using only local information. The *diminish-take* model enforces both the confinement property [SW00, Lam73] and the **-property*, and is readily reduced to practice. Finally, *diminish-take* provides the necessary semantic underpinnings to account for the secure use of user-supplied memory managers and more generally for the safety of decomposing applications into protected subsystems.

In addition to describing the new access model, we give a number of examples of its application in a real system: EROS [SSF99]. Through these examples, it is shown that two "innocuous" changes to the *take-grant* model are sufficient to model a real, high-performance system, and that the new access right appears repeatedly in the design patterns that underly the system's security, flexibility, and performance.

# 2 The *Take-Grant* Model

The *take-grant* model uses an access graph consisting of subjects and objects. This graph is connected by directed arcs, each of which is labelled by the set of access rights it conveys. Access rights are selected from the set {**read**,**write**,**take**,**grant**}, and are abbreviated {**r**,**w**,**t**,**g**}.[1] Subjects are shown as circles and objects as squares. Thus, the graph in Figure 1 indicates that subject $S$ has **read** and **grant** authority on the object $O$.

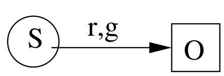


Figure 1: Sample access graph.

The access rights correspond to the primitive operations of a capability system:

**read** The subject copies data from the designated subject or object.

**write** The subject copies data to the designated subject or object.

**take** The subject copies a capability from the designated subject or object.

**grant** The subject copies a capability to the designated subject or object.

A discussion of the **call** access right has been omitted for reasons of space.

## 2.1 Transitivity

In examining a *take-grant* access graph, it is important to understand that the graph represents an *initial* configuration of the system from which execution will proceed, and that the copying of capabilities adds new arcs – therefore new access paths – to the graph.

To expose the latent access paths in a *take-grant* graph, it is necessary to perform a transitive closure on the original graph. The goal of this closure is to perform all possible capability propagations so as to understand where access rights may end up. As a trivial example, Figure 2 shows that the subject $S0$ is able to obtain **take**, **read**, and **write** access to object $O$ (dashed arc) by taking $S1$'s capability to $O$.

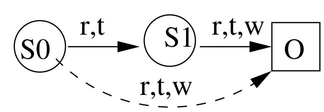


Figure 2: Transitive closure.

For purposes of verifying security policies, we assume that all processes are maximally misbehaved (or maximally cooperative, depending on point of view). That is, any operation that *can* be performed *will* be performed. Therefore, in Figure 3 we assume that subject $S0$ will obtain access to object $O$ because subject $S1$ will cooperate in transmitting that access.

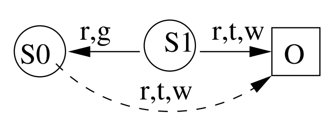


Figure 3: Presumed collusion.

[1] We follow the description given by [Bis79], which is somewhat simplified from the original version given by [Sny77].

The closure rules for *take-grant* are summarized in Figure 4.

$$
\begin{array}{lllll}
a \xrightarrow{\mathtt{g}} b & \wedge & a \xrightarrow{\alpha} c & \Rightarrow & b \xrightarrow{\alpha} c \\
a \xrightarrow{\mathtt{t}} b & \wedge & b \xrightarrow{\alpha} c & \Rightarrow & a \xrightarrow{\alpha} c
\end{array}
$$

Figure 4: *Take-grant* closure rules

.

## 2.2 *De Facto* Access

Bishop [Bis79] points out that there are *two* kinds of access that must be considered: *de jure* and *de facto*.

*De Jure* access is access that can be obtained by performing the transitive closure of Figure 4. It is said to be *de jure* because all operations performed using these rights are specifically permitted by the access model.

*De Facto* access is access that arises from collusion between subjects. For example, if $S0$ can read from $S1$, and $S1$ can in turn read from $O$, then $S0$ can read from $O$ provided that $S1$ conspires (Figure 5). Under our previously stated assumptions, we presume that all processes that do not run trusted applications will conspire. We will note *de facto* access rights by

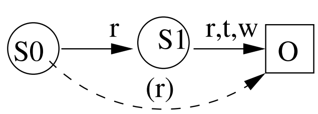


Figure 5: Presumed conspiracy.

annotating arcs in the graph with an additional set of parenthesized labels that indicate *de facto* rights. Where both *de jure* and *de facto* access rights of a given type exist, we will show only the *de jure* rights to avoid overcrowding.

The additional closure rules for *de facto* access in *take-grant* are given in Figure 2.2.

## 2.3 Weaknesses

From the standpoint of practical implementation, the *take-grant* model has two weaknesses:

$$
\begin{array}{lllllll}
 & & a \xrightarrow{\mathtt{r}} b & \Rightarrow & a \xrightarrow{(\mathtt{r})} b & & \textit{base} \\
 & & a \xrightarrow{\mathtt{w}} b & \Rightarrow & a \xrightarrow{(\mathtt{w})} b & & \\
 & & & & & & \\
 & & a \xrightarrow{(\mathtt{r})} b & \Rightarrow & b \xrightarrow{(\mathtt{w})} a & & \textit{symmetry} \\
 & & a \xrightarrow{(\mathtt{w})} b & \Rightarrow & b \xrightarrow{(\mathtt{r})} a & & \\
 & & a \xrightarrow{(\mathtt{t})} b & \Rightarrow & b \xrightarrow{(\mathtt{g})} a & & \\
 & & a \xrightarrow{(\mathtt{g})} b & \Rightarrow & b \xrightarrow{(\mathtt{t})} a & & \\
 & & & & & & \\
a \xrightarrow{(\mathtt{r})} b & \wedge & b \xrightarrow{(\mathtt{r})} c & \Rightarrow & a \xrightarrow{(\mathtt{r})} c & & \textit{transitivity} \\
a \xrightarrow{(\mathtt{w})} b & \wedge & b \xrightarrow{(\mathtt{w})} c & \Rightarrow & a \xrightarrow{(\mathtt{w})} c & &
\end{array}
$$

Figure 6: Additional closure rules for *de-facto* access

- It is insufficiently expressive. It cannot account for the safety of memory mapping tables (indirect access).
- It is overly powerful. It cannot describe access graphs in which connected subgraphs can be shared in transitively read-only form.

The first problem is reasonably self-explanatory. To account for the safety of mapping tables, we require some access right that allows the mapping table to be *traversed* without allowing it to be *read*. The *take-grant* model provides no such access right.

The second problem perhaps does not leap out as quickly. Suppose subject $S$ builds a graph of objects as shown in Figure 7. Object $O1$ might, as an example, be an executable image and Object $O2$ might be it's symbol table. $S$ wishes to allow other subjects to create new processes using this executable image, and also to read the symbol table for purposes of debugging, but does not wish these other subjects to be able to modify either data structure or to use the executable image as a vehicle for communicating between subjects.

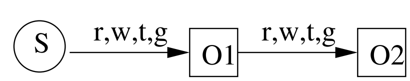


Figure 7: Executable and symbol table.

Clearly the **grant** and **write** rights must be removed when handing out the executable images, which yields the structure shown in Figure 8. Imagine, however,

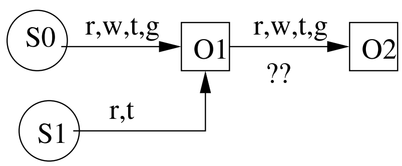


Figure 8: Shared executable and symbol table.

that $S0$ is the developer and $S1$ a potential user. If **grant** and **write** rights are removed, then $S1$ cannot modify the symbol table, but neither can $S0$! If the object graph is copied, identity between the developer version and the client version is destroyed, necessitating additional record keeping information that is prone to mismanagement.

In the general case, we may wish the client not only to be able to read the content of the symbol table, but also to extract from it other capabilities in read-only form. What is desired is a form of access that provids transitive read-only access to $S1$ without altering the graph as seen by $S0$. The *take-grant* model, and most capability systems, lack an access right that meets this need.

# 3 The *Diminish-Take* Model

The *diminish-take* model augments *take-grant* by introducing indirection and providing a new access rights operation: *diminish*. This operation is used by two new access rights: **dimtake** and **dimgrant** (abbreviated **dt** and **dg**).

## 3.1 Indirection

Indirection rights introduce the ability to perform reads across multiple subjects or objects. The *take-grant* **read** right is divided into **read access** and **read indirect**, respectively denoted **r** and **ri**. The **take**, **grant**, and **write** rights are similarly divided.

Under the revised access model, a subject $S$ can read an object $O$ if

$$
\begin{array}{lccc}
\text{Either} & S \xrightarrow{\mathbf{r}} O & & \\
\text{Or} & S \xrightarrow{\mathbf{ri}} O_0 & & \textit{and} \\
 & O_n \xrightarrow{\mathbf{r}} O & & \textit{and} \\
 & \forall i, 0 \leq i < n, e_i \xrightarrow{\mathbf{ri}} O_{i+1} & &
\end{array}
$$

That is, the subject can read the object if there is some path consisting of zero or more indirect rights terminating in an access right of the same kind.[2] The transitive closure rules of Figure 2.2 are revised to those of Figure 9.

$$
\begin{array}{ccccccl}
 & & a \xrightarrow{\mathbf{r}} b & \Rightarrow & a \xrightarrow{(\mathbf{r})} b & & \textit{base} \\
 & & a \xrightarrow{\mathbf{w}} b & \Rightarrow & a \xrightarrow{(\mathbf{w})} b & & \\
 & & a \xrightarrow{\mathbf{t}} b & \Rightarrow & a \xrightarrow{(\mathbf{t})} b & & \\
 & & a \xrightarrow{\mathbf{g}} b & \Rightarrow & a \xrightarrow{(\mathbf{g})} b & & \\
 \\
 & & a \xrightarrow{(\mathbf{r})} b & \Rightarrow & b \xrightarrow{(\mathbf{w})} a & & \textit{symmetry} \\
 & & a \xrightarrow{(\mathbf{w})} b & \Rightarrow & b \xrightarrow{(\mathbf{r})} a & & \\
 & & a \xrightarrow{(\mathbf{t})} b & \Rightarrow & b \xrightarrow{(\mathbf{g})} a & & \\
 & & a \xrightarrow{(\mathbf{g})} b & \Rightarrow & b \xrightarrow{(\mathbf{t})} a & & \\
 \\
a \xrightarrow{\mathbf{ri}} b & \wedge & b \xrightarrow{(\mathbf{r})} c & \Rightarrow & a \xrightarrow{(\mathbf{r})} c & & \textit{transitivity} \\
a \xrightarrow{\mathbf{wi}} b & \wedge & b \xrightarrow{(\mathbf{w})} c & \Rightarrow & a \xrightarrow{(\mathbf{w})} c & & \\
a \xrightarrow{\mathbf{ti}} b & \wedge & b \xrightarrow{(\mathbf{t})} c & \Rightarrow & a \xrightarrow{(\mathbf{t})} c & & \\
a \xrightarrow{\mathbf{gi}} b & \wedge & b \xrightarrow{(\mathbf{g})} c & \Rightarrow & a \xrightarrow{(\mathbf{g})} c & & \\
 \\
a \xrightarrow{(\mathbf{g})} b & \wedge & a \xrightarrow{\alpha} c & \Rightarrow & b \xrightarrow{\alpha} c & & \\
a \xrightarrow{(\mathbf{t})} b & \wedge & b \xrightarrow{\alpha} c & \Rightarrow & a \xrightarrow{\alpha} c & &
\end{array}
$$

Figure 9: Closure rules with indirection

Note that indirect access does *not* imply the ability to obtain direct access. This is why we show the combination of indirect and direct rights as yielding a *de facto* right.

Indirect rights provide a *partial* solution to the problem of transitive access. In our executable and sym-

[2] We are aware of no additional complications introduced if subject can also be traversed indirectly, but have ommitted this from the discussion in the interests of clarity.

bol table example it is now possible to grant a capability to $S1$ conveying $\{\mathbf{ri},\mathbf{r}\}$ authority (Figure 10).

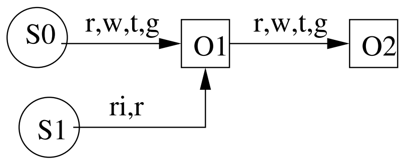


Figure 10: Shared executable with indirect rights.

While this solves the problem in theory, it is inconvenient in practice. Stepping outside of the model for a moment, we must now build into our executable abstraction an interface for making symbol table inquiries. A better solution would be to provide some means to fetch a restricted form of the symbol table capability – once again without requiring alterations to the internal arcs of the graph.

### 3.2 On Indirection and Channels

It is natural to imagine from the names that the **read indirect** and **write indirect** rights are in some fashion data access rights. They are not. The indirect rights are in fact a peculiarly weakened forms of the **take** right. The traversal permitted by indirection follows a path defined by capabilties, and therefore implicitly observes the state of these capabilities. A change to some capability on a traversed path is observable as a change in values at the location named by the path, and the mapping structure itself is therefore observable by traversal. This is not, on reflection, entirely surprising.

Further consideration reveals that there is a non-obvious channel to be considered. In Figure 10, a capabilty grant performed by $S_0$ is observable via the **ri** right by $S_1$. This yields the somewhat discomforting realization that certain combinations of **take** and **grant** create a *data* communication path (Figure 11).

Given that the indirection rights are best considered as a weakened form of the **take** right, what is *really* surprising is that there appears to be no appropriate symmetry rule for the indirection rights. Certainly, indirection does not allow capabilities to be transferred via object $O1$, but *something* certainly flows

$$
\begin{array}{lcl}
a \xrightarrow{(\mathbf{g})} b \;\wedge\; c \xrightarrow{(\mathbf{t})} b & \Rightarrow & a \xrightarrow{(\mathbf{w})} c \\
a \xrightarrow{(\mathbf{g})} b \;\wedge\; c \xrightarrow{(\mathbf{ri})} b & \Rightarrow & a \xrightarrow{(\mathbf{w})} c \\
a \xrightarrow{(\mathbf{g})} b \;\wedge\; c \xrightarrow{(\mathbf{wi})} b & \Rightarrow & a \xrightarrow{(\mathbf{w})} c \\
a \xrightarrow{(\mathbf{g})} b \;\wedge\; c \xrightarrow{(\mathbf{ti})} b & \Rightarrow & a \xrightarrow{(\mathbf{w})} c \\
a \xrightarrow{(\mathbf{g})} b \;\wedge\; c \xrightarrow{(\mathbf{gi})} b & \Rightarrow & a \xrightarrow{(\mathbf{w})} c
\end{array}
$$

Figure 11: Implied *De Facto* Channels

toward $S1$ when the capability state of $O1$ changes. We considered labeling this a *de facto* **write** access from $O1$ to $S1$, but this would in turn imply a symmetric *de facto* **read** from $S1$ to $O1$, which is decidedly a stronger relationship than exists in the real system.

We are left with the discomforting sense that some failure of symmetry has occurred in the model, but this does not appear to be true. All of the possible communications between $S_0$ and $S_1$ through this path are covered by the implied channel closure rules of Figure 11.

### 3.3 The Diminish Operator

The *diminish* operation is a form of capability copy that weakens the transferred capability as the copy is performed. The net effect of this is that the diminish operation enforces *transitive* read-only access. This operation is authorized by two new access rights: **dimtake** and **dimgrant** (abbreviated **dt** and **dg**). As before, each of these rights has associated indirection rights.

If $A$ is a set of access rights, then

$$diminish(\xrightarrow{A} \alpha) \quad \Rightarrow \quad \xrightarrow{A \cap \{\mathbf{ri},\mathbf{r},\mathbf{dti},\mathbf{dt}\}} \alpha$$

That is, the only rights that are preserved by the diminish transformation are the diminish and read rights and their associated indirections. As a sanity condition, we will require that **take** access should imply **dimtake** access and **grant** access should imply **dimgrant** access.

The closure rules of Figure 9 are now augmented by those of Figure 12. Note that the ability to perform a diminished fetch does not imply full-fledged grant authority in the opposite direction. Instead, it implies

$$
\begin{aligned}
&& a \xrightarrow{\mathbf{dt}} b &\Rightarrow a \xrightarrow{(\mathbf{dt})} b \\
a \xrightarrow{(\mathbf{dt})} b &\wedge & b \xrightarrow{\mathbf{r}} c &\Rightarrow a \xrightarrow{\mathbf{r}} c \\
a \xrightarrow{(\mathbf{dt})} b &\wedge & b \xrightarrow{\mathbf{dt}} c &\Rightarrow a \xrightarrow{\mathbf{dt}} c \\
a \xrightarrow{(\mathbf{dg})} b &\wedge & a \xrightarrow{\alpha} c &\Rightarrow b \xrightarrow{dim(\alpha)} c
\end{aligned}
$$

Figure 12: Additional rules with *diminish*

a form of diminished grant authority.

### 3.4 Decidability

In the interest of focusing on practical application, we omit a proof that safety in *diminish-take* is decidable in linear time. Instead, we briefly outline the intuitions behind the claim.

First, note that the decidability of *take-grant* is not altered by the introduction of a second $\mathbf{take}$ right. Let is imagine (arbitrarily) that the two rights will be called **red-take** and **blue-take**. This adds no fundamental complexity to the existing proof presented by Jones, Lipton and Snyder [JLS76].

Second, observe that the $\mathbf{dimtake}$ is strictly less powerful than the $\mathbf{take}$ right, and that the transitive effects introduced by the $\mathbf{dimtake}$ right are therefore strictly less than those of the $\mathbf{take}$ right. Since there is no increase in complexity due to either the addition of a second take right or the reduction in power of that right, the linear time proof for *take-grant* is straightforwardly adapted to *diminish-take*. The same argument applies for $\mathbf{dimgrant}$.

Finally, note that the same line of argument applies to the introduction of indirect rights. The two-step paths given in Figure 13 have the same traversal complexity with respect to the safety verification.

$$
\begin{array}{lllr}
a \xrightarrow{(\mathbf{ri})} b & \wedge & b \xrightarrow{(\mathbf{r})} c & \textit{diminish-take} \\
a \xrightarrow{(\mathbf{r})} S & \wedge & S \xrightarrow{(\mathbf{r})} c & \textit{take-grant}
\end{array}
$$

Figure 13: Comparative path complexity

It follows that for every safety problem in *diminish-take*, there exists an equivalently complex safety problem in the *take-grant* model that is decidable in linear time. The answers may of course be different, but the complexity is the same.

## 4 Practical Applications

We now turn attention to the practical utility of the *diminish-take* model, giving several examples of how it is applied in the EROS system to yield a system that is simultaneously flexible and securable.

EROS [SSF99] is a pure capability-based operating system that implements a transparent single-level store. At the operating system level of abstraction, there are a number of interpretations of the primitive objects, but at the storage abstraction layer there are only two object types: ***nodes*** and ***pages***. A page is an array of bytes whose length is determined by the underlying architecture's page size. A node is a fixed-length C-list containing 32 capabilities.

The purpose of this section is to sketch several of the design patterns that are pervasively used in EROS and to show how the *diminish* operation plays a pivotal role in each of them.

### 4.1 Constructors

The EROS program instantiation mechanism is known as the *constructor*. In it's simplest form, a constructor holds a read-only address space and a set of "initial capabilities." On request, it instantiates a new process that executes from a lazy copy of this address space and begins its life holding a copy of the initial capabilities. This new program is referred to as the "yield" of the constructor.

In addition to instantiating programs, the constructor is able to certify to its requestor whether its yield is confined. Because of the ability to perform this certification, the constructor is also the means of instantiating protected subsystems. The definition of confinement is that the newly instantiated program's capabilities can be divided by the constructor into three categories:

1. Capabilities that trivially convey no write authority.
2. Capabilities that the client has explicitly authorized in a passed set of "authorized holes."

3. Capabilities to constructors that in turn certify that their yield is confined.

The last test involves mechanisms that are beyond the scope of this paper. The second test is performed by set comparison. The first is done by case analysis:

- Number capabilities (integers) convey no write authority.
- Read-only page capabilities convey no write authority.
- Read-indirect, diminished node capabilities convey no write authority.
- Three kernel-implemented capabilities are known to be read-only as special cases.

all other cases are deemed unsafe. The correctness of this mechanism has been formally verified [SW00].

Note that these tests are completed in at most 96 steps (up to three tests for each of the 32 initial capabilities). In principle, if $S$ is the set of authorized holes then the set membership test takes $log_2(|S|)$ time per capability. In practice, the provision for authorized holes has never been used in real programs and this test is ommitted. The safety test therefore runs in constant time in the normal case.

While it is not immediately apparent from the description above, the $\mathbf{dimtake}$ authority plays an essential role in the practical utility of the constructor. Because the address space is a diminished capability, it is safe to incorporate it directly into each yield. There is no path through a diminished capability that will allow different yields to communicate to each other or with the author of the program.

In the absence of the $\mathbf{dimtake}$ authority, it would be necessary to either (a) explicitly copy the address space to avoid communication between the various clients, or (b) use the $\mathbf{ri}$ right to ensure that capabilities could not be extracted from the mapping tree. As we will see in Section 4.3, address fault handlers require access to the mapping structures, and the $\mathbf{ri}$ solution would therefore eliminate the possibility of untrusted user-level address fault handlers.

Because it is *not* necessary to perform such a copy, the constructor is able to instantiate new processes very quickly – roughly 65% faster (664$\mu S$ vs. 1920 $\mu S$) than a conventional UNIX implementation is able to instantiate new processes using the *fork* system call [SSF99].

## 4.2 Memory Mapping

EROS address spaces are constructed by building a tree of nodes whose leaf objects are pages (Figure 14). An ordinary node capability conveys $\{\mathbf{ri},\mathbf{wi},\mathbf{t},\mathbf{g}\}$ authority. A read-only node capability conveys $\{\mathbf{ri},\mathbf{t}\}$ authority. A segment capability (the name comes from Multics [SCS77]) conveys $\{\mathbf{ri},\mathbf{wi}\}$ or $\{\mathbf{ri}\}$, according to whether it is read-write or read-only.

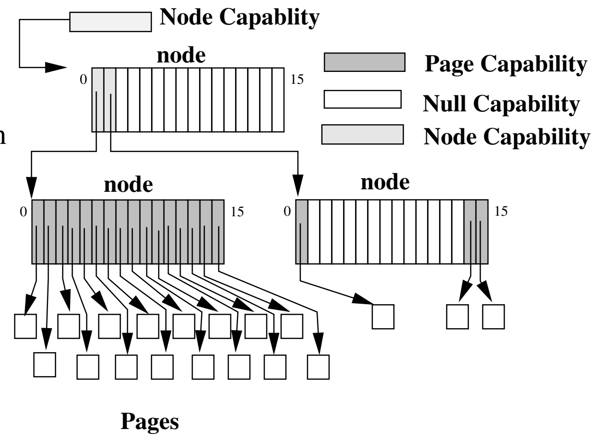


Figure 14: An EROS address space

Under normal circumstances, a process is given only a segment capability to its own address space. This gives it the authority to read and write the content of the address space without being able to modify the mapping information for the space. No comparable restriction is achievable in *take-grant*.

## 4.3 User-Level Memory Managers

Certain address spaces nodes are distinguished by a mode field that indicates that these nodes are "owned" by a fault handler (Figure 15). Such nodes contain a process capability to the fault handler for that space. If an access violation or invalid memory reference occurs within the contained subtree, a message is synthesized by the kernel and directed to the fault handler. This message includes a node capability to the owned node.[3]

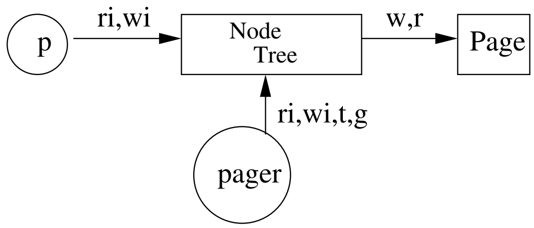


Figure 15: Address fault flow

When a new program is to be created, an initial memory tree is constructed and then *frozen*. Freezing a memory tree proceeds as follows:

1. The initial fault handler constructs a node capability with $\{\mathbf{ri},\mathbf{dt}\}$ authority to the current node tree.
2. A new *constructor* of copy on write handlers is fabricated that initially holds only this restricted capability and a similarly restricted capability to the executable image of the copy on write handler. Note that this satisfies the safety test described in Section 4.1, and that the yield of this new constructor of copy on write handlers is therefore confined.
3. As program instances are instantiated, the copy on write constructor is used to produce new instances of the "copy on write" handler. The newly instantiated handlers operate by lazily copying the protected original space as the new program instance incurs write faults.

The copy on write is performed incrementally by selectively copying nodes and pages as needed in order to construct a new address space tree with modifiable pages (Figure 16).

The key point here is not that there is a "clever" implementation of copy on write, but rather that the owner of the original address space need not have any trust in the new fault handler. The integrity of

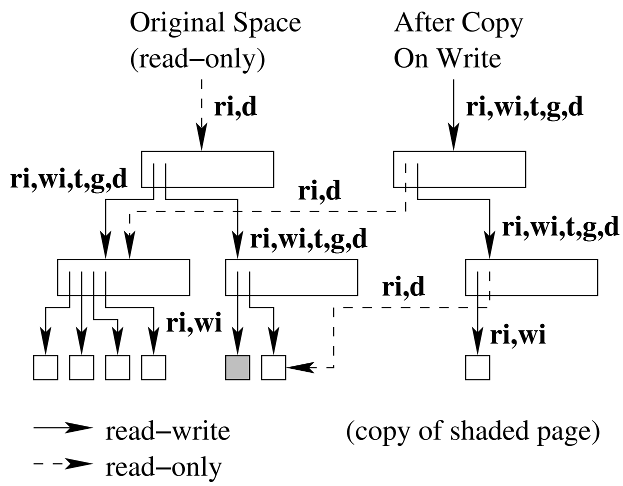


Figure 16: Virtual copy spaces

the original address space is assured not by the good intentions of the fault handler, but by the fact that the fault handler has no write authority to the original address space. While the constructor mechanism implements instantiation and provides confinement assurances, it has no knowledge of the behavior of the programs that it instantiates. In the example, the virtual copy may safely be in a higher security level than the original.

Similarly, a reference monitor need not rely on knowing the behavior of the fault handler to ensure that (e.g.) there is no information transfer from the client process to the program author. The use of a restricted address space capability ensures this.

### 4.4 Enforcing the *-Property

Boebert argues that unmodified capability systems are incapable of enforcing the **-property* [Boe84]. He notes that a low process may create an object with read-write authority, that a high process may then open this object with read authority. The low process then stores a read-write capability to some second object into the first. The high process reads this capability, thereby establishing a read-write channel between low and high.

Boebert's access model is never explicitly stated, but it is clear from the example that he assumes a unification of read and take authorities. That is, there is no

[3] From a security standpoint, this synthesis is safe. It is justified by the fact that the fault handler was the original builder of this memory tree, and therefore was in a position to retain such a capability in the first place. Synthesizing a new capability with each upcall allows a single fault handler to serve several memory spaces if desired.

difference between authority to read a capability and authority to read data in his access model. This is the fundamental reason that Boebert's example leaks: it permits the high-security process to obtain **take** access to a low-security object, which should never have been permitted. The quandry proposed by Boebert is entirely resolved if we impose the restriction that access to lower-security objects be restricted to $\{\mathbf{ri},\mathbf{r}\}$.

Unfortunately, this restriction would preclude the previously described copy on write mechanism. More importantly, it would preclude the entire design pattern by which user-supplied fault handlers and protected subsystems can be executed safely. In systems incorporating the **dimtake** right, it is sufficient to restrict access to lower-security objects to subsets of $\{\mathbf{ri},\mathbf{r},\mathbf{dti},\mathbf{dt}\}$. Which provides just enough power to support user fault handlers without creating undesirable communications paths. Other uses of this design pattern might include a high-security integrity checker to traverse a low security object graph and verify that the graph itself is well formed. A high-security garbage collector can be implemented with assurance that it cannot transmit information to low-security programs.[4]

### 4.5 Secure Compartments

The KeySafe security architecture [Raj89] combines the constructor mechanism and diminished capabilities to support controlled sharing of objects across compartments. Compartments are constructed by instantiating their programs as confined subsystems. The confinement boundary *is* the compartment boundary. Capabilities passed from low security compartments to high security compartments are restricted to safe capabilities (using the test described in Section 4.1) or capabilities to processes that are trusted by the KeySafe reference monitor.

As with copy on write spaces, the efficiency of sharing is greatly enhanced by the absence of any need to perform copies of complex structures before allowing their capabilities to cross compartment boundaries.

[4] There remains a covert channel inherent in the observability of object disappearance, but the garbage collector sweep pass can be ignored as a potential source of improper communications.

### 4.6 Design Pattern

The mechanisms described in this section include most of the core security mechanisms used in EROS. Hopefully, their description has demonstrated that reuse of read-only shared structure is a pervasive design pattern, and an essential source of efficiency in the operating environment that has been built on top of the EROS kernel.

## 5 Related Work

As we have already discussed Boebert's work [Boe84] we will not repeat it here. We have also extensively discussed the work of Jones, Lipton, Snyder, and Bishop.

### 5.1 SCAP

Karger [Kar88] has also asserted that an unmodified capability system cannot enforce confinement. He specifically examines the KeyKOS factory mechanism (the predecessor to the EROS constructor), and concludes that it is impractically slow. In personal discussion, it emerged that his definition of confinement does not match that of [Lam73], but is rather the **-property*. His claim should therefore be taken to be that a capability system conforming to the *take-grant* model is not strong enough to enforce multi-level security (MLS).

For reasons we have discussed in Section 4.4, we believe that this is practically correct but formally mistaken. From Section 4.1, it should be clear that if one is prepared to do a prohibitive number of copies, it is possible in principle to implement confined subsystems without the **dimtake** right. While the underlying capability system still cannot enforce MLS, KeySafe [Raj89] has demonstrated that a reference monitor can be constructed at user level that implements MLS across confined compartments. In the absence of the **dimtake** right, however, we agree that the resulting system requires enough copies to be unusable, and requires that all memory managers be trusted by the reference monitor.

As should be clear from Section 4.1, the constructor is a static precondition check. Once the yield is instantiated by the constructor, the yield and its client communicate directly with no intermediation. Regrettably, the only documentation of the KeyKOS factory (the constructor predecessor) that was available to Karger was the original Key Logic patent document [Key86]. Like most patent documents, the factory patent is somewhere between opaque and incomprehensible. Karger inferred from the patent that all communication between confined subsystems must be mediated on an ongoing basis by the factory. Given the performance of the EROS IPC mechanism it is not clear that this would be fatal, but it is certainly a concern. Such intermediation does not occur.

Karger's SCAP design offers an alternative mechanism for supporting MLS, which is to adopt a hybrid capability system in which access depends both on possessing a capability and satisfying what amounts to an access control list check.

### 5.2 KeyKOS

While both the implementation and the access model described here are new, the EROS architecture is closely derived from that of KeyKOS [Har85]. The *diminish* access restriction implemented in EROS is a refinement on the "sense capability" incorporated in the KeyKOS design.

Norman Hardy, the principal architect of KeyKOS, credits the Burroughs B5000 family [Bur61] as the inspiration for the sense capability. In attempting together to reconstruct the process of invention, we have been unable to identify the source of the inspiration. Certainly the B5000 family was the first machine to implement a hierarchical read-only protection mechanism, but this mechanism applied only to data. CAP appears to have been the first design to actually separate data read and capability read access rights [WN79], and a careful study of the two systems at the same time might have led to the necessary leap.

The EROS *diminish* restriction differs from the KeyKOS sense capability in that sense capabilities are always read-only. That is, the "sense" capability was conceived as a capability type rather than as an access modifier. In the later EROS implementation, a distinct "weak" access right was introduced that corresponds exactly to the $\mathbf{dimtake}$ right described here. Functionally, a KeyKOS sense capability corresponds to an EROS weak, read-only capability. More broadly, exploration of the *diminish-take* model has enabled us to subsume several KeyKOS capability types back into the generic node capability by introducing the missing access rights. This has not significantly changed the power of the system, but it does provide a more uniform access model.

Neither KeyKOS nor EROS currently implements a diminished grant authority, though it would be fairly straightforward to do so should this ever prove useful.

## 6 Acknowledgements

No work derived from KeyKOS could be complete without acknowledging the principal architects and implementors of that system: Norman Hardy, Charlie Landau, and William Frantz. Each of these individuals has participated in and encouraged work on the EROS system.

The discovery of Matt Bishop's paper [Bis79] in the Stanford University engineering library came as something of an uncomfortable shock. At the time, I had just finished the *diminish-take* chapter of my dissertation in which the same ideas and notation had been independently reinvented. Bishop's kind encouragement that the new model should nonetheless be published in the mainstream literature was greatly appreciated, and directly led to this paper. His comments on an early draft of this paper were also very helpful.

## 7 Conclusion

We have introduced a new rights operation, *diminish*, and an extension of the *take-grant* access model that incorporates both this operation and indirection rights. We have given the necessary rules for computing both *de jure* and *de facto* access rights within this model, and given an informal proof-sketch confirming that the *safety problem* is decidable in the new model in linear time. In the special case conditions used by the constructor, safety is decidable in constant time.

We have shown that pure capability systems can enforce the **-property* with a reasonable expectation of efficiency.

We have illustrated a number of design patterns of the EROS design that critically depend on the new access right for correctness, security, performance, or all three. Without some mechanism providing transitive read-only authority, we believe that an efficient and securable pure capability system is infeasible.

Finally, we have shown a real system design that maps well onto a decidable access model without sacrificing performance. To our knowledge, this is the first time that such a combination has been achieved.

The EROS system is publicly available and executes on Pentium-class processors. Further information on EROS can be obtained from the system web site at `http://www.eros-os.org`

## References


[Bis79] Matt Bishop. The transfer of information and authority in a protection system. In *Proc. 7th ACM Symposium on Operating Systems Principles*, pages 45–54, December 1979. Published as *Operating System Review*, Vol 13, No 4.

[Boe84] 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, September 1984. National Bureau of Standards.

[Bur61] Burroughs Corporation, Inc. *The Descriptor – a Definition of the B5000 Information Processing System*. Detroit, MI, USA, February 1961.

[Har85] Norman Hardy. The KeyKOS architecture. *Operating Systems Review*, pages 8–25, October 1985.

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

[JLS76] A K. Jones, R. J. Lipton, and L. Snyder. A linear time algorithm for deciding security. In *Proc. 17th Symposium on Foundations of Computer Science*, pages 33–41, Houston, Texas, 1976.

[Kar88] Paul Karger. *Improving Security and Performance for Capability Systems*. PhD thesis, University of Cambridge, October 1988. Technical Report No. 149.

[Key86] Key Logic, Inc. *U.S. Patent 4,584,639: Computer Security System*. U. S. Patent Office, 1986.

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

[Raj89] S. A. Rajunas. The KeyKOS/KeySAFE system design. Technical Report SEC009-01, Key Logic, Inc., March 1989. http://www.cis.upenn.edu/˜KeyKOS.

[SCS77] Michael D. Schroeder, David D. Clark, and Jerome H. Saltzer. The MULTICS kernel design project. In *Proc. 6th ACM Symposium on Operating Systems Principles*, pages 43–56. ACM, November 1977.

[Sny77] Lawrence Snyder. On the synthesis and analysis of protection systems. In *Proc. 6th ACM Symposium on Operating Systems Principles*, pages 141–150, November 1977. Published as *Operating System Review*, Vol 11, No 5.

[Sny81] Lawrence Snyder. Theft and conspiracy in the take–grant protection model. *J. Computer and System Sciences*, 23:233–347, 1981.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David 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, December 1999. ACM.

[SW00] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proc. 2000 IEEE Symposium on Security and Privacy*, pages 166–176, Oakland, CA, USA, 2000.

[WN79] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System*. Elsevier North Holland, 1979.