Submitted to the *2004 IEEE Symposium on Security and Privacy*,
Oakland, CA 2004

The Practical Application of a Decidable Access Model*

SRL Technical Report SRL-2003-04

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

November 3, 2003

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*, fail to model issues that are needed to account for safety and security in capability-based operating systems.

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. These cases prove particularly important for enforcement of the **-property* and multi-level security in capability-based operating systems.

We also describe the relationship between the model and a real system: EROS. EROS is a fast capability system for commodity processors whose higher-level security properties are based on the model described here. We show that the *diminish* operation is a critical underpinning for an efficient realization of several design patterns in EROS, including confinement, protected subsystems, and user-supplied memory managers.

1 Introduction

Capability systems are sometimes dismissed because of confusion about their ability to enforce higher-level security properties. In particular, a note by Boebert claiming that a particular type of unmodified capability system cannot enforce the **-property* [Boe84] has directly and indirectly influenced key subsequent publications [Gon89, CDM01, Kar88, WBDF97], because its limitations were not carefully understood. Later work by Kain and Landwehr [KL86] notes that Boebert's claim applies only in systems where capability and data transmission cannot be separated. This includes cryptographic and sparsely allocated capability systems, and some password capability systems. It does not include capabilities protected by the operating system, by hardware tags, or by a safe language runtime. Regrettably, Kain and Landwehr's analysis of systems that *can* enforce the **-property* is flawed. In principle, several of the capability systems that had actually been built and reported by 1984, including PSOS [NBF$^+$80] *could* enforce the **-property*. In one such system (GNOSIS [FLH83]) this was later demonstrated [Raj89]. In another case, relevant research was suppressed:

> Kain and I had come up with a hardware solution which the Government sponsor of the project chose not to release for publication – all the while strongly encouraging us to support the conference. So I wrote up the problem statement, omitted the solution, and moved on [Boe03].

* 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.

Given that capability-based operating systems *can* enforce the **-property* (and therefore multilevel security [DoD85]), and can also enforce the confinement property [Lam73, SW00], it seems appropriate to investigate more carefully whether there may exist an access model for capability systems that (a) remains decidably safe, and (b) corresponds closely enough to the behavior of real capability implementations to be directly applicable in their analysis. The canonical formal access model for capability systems, *take-grant* [JLS76, Sny77], has limited practical utility for several reasons:

- Deciding safety requires a global scan of a snapshot of the access graph. In practical application, global scans of this form are infeasible; a locally-decidable access model is required.
- It models a class of capability system that is not widely familiar to either students or industrial practitioners. Indeed, confusion concerning the meaning of the term "capability" has led to no fewer than *four* distinct protection models that have been labeled "capability systems" [MYS03].
- From the standpoint of the system implementor, there is a significant semantic gap between the model and real practice: the absence of indirection rights. These are needed to model the behavior of memory translation.
- It does not address the protection problems arising from transitive reachability when shared state must be accessable to parties with differing access rights. This is also an issue in modeling languages-based security kernels, such as W7 [Ree96].

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 for security policies governing information flow; if the propagation of access rights cannot be controlled, then the flow of information likewise cannot be controlled. 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 capability systems in which safety is decidable in linear time. Subsequent work by Bishop and Snyder [BS79, 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 revised and extended version of the *take-grant* model providing indirection and incorporating *diminish*, a new fundamental rights weakening 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. Systems modeled by *diminish-take* can enforce both the confinement property [SW00, Lam73] and the **-property*, and the model has been reduced to practice in a real system. 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.

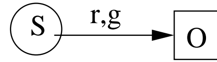

Figure 1: Sample access graph.

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.

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 S_0 is able to obtain **take**, **read**, and **write** access to object O (dashed arc) by taking S_1's capability to O. For purposes of closure computation, we assume that all

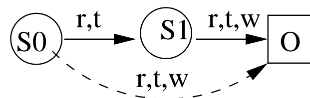

Figure 2: Transitive closure.

processes are maximally misbehaved, and that any operation that *can* be performed *will* be performed. The closure rules for *take-grant* are summarized in Figure 3.

$$s \xrightarrow{\mathbf{g}} a \quad \wedge \quad s \xrightarrow{\alpha} b \quad \Rightarrow \quad a \xrightarrow{\alpha} b \quad (s \text{ any subject})$$
$$s \xrightarrow{\mathbf{t}} a \quad \wedge \quad a \xrightarrow{\alpha} b \quad \Rightarrow \quad s \xrightarrow{\alpha} b$$

Figure 3: *Take-grant* closure rules

.

[1] We follow the description given by [BS79], which is somewhat simplified from the original version given by [Sny77].

2.2 *De Facto* Access

Bishop [BS79] 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 3. 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 interactions between subjects. For example, if S_0 can read from S_1, and S_1 can in turn read from O, then S_0 can read from O provided that S_1 conspires (Figure 4). We assume that all non-trusted processes will conspire. We will note *de facto* access rights by annotating arcs

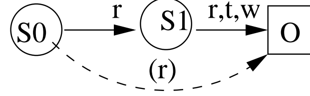

Figure 4: Presumed conspiracy.

in the graph with 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 5. The *de facto* take and grant access arises from the possibility of two subjects having the ability to read and write capabilities to a commonly shared object.

$$
\begin{array}{lll}
s_0 \xrightarrow{\mathbf{r}} a \;\Rightarrow\; s_0 \xrightarrow{(\mathbf{r})} a & & \textit{base} \\
s_0 \xrightarrow{\mathbf{w}} a \;\Rightarrow\; s_0 \xrightarrow{(\mathbf{w})} a & & (s_0, s_1 \text{ subjects}) \\
s_0 \xrightarrow{\mathbf{t}} a \;\Rightarrow\; s_0 \xrightarrow{(\mathbf{t})} a & & \\
s_0 \xrightarrow{\mathbf{g}} a \;\Rightarrow\; s_0 \xrightarrow{(\mathbf{g})} a & & \\
\\
s_0 \xrightarrow{(\mathbf{r})} a \;\Rightarrow\; a \xrightarrow{(\mathbf{w})} s_0 & & \textit{symmetry} \\
s_0 \xrightarrow{(\mathbf{w})} a \;\Rightarrow\; a \xrightarrow{(\mathbf{r})} s_0 & & \\
s_0 \xrightarrow{(\mathbf{t})} a \;\Rightarrow\; a \xrightarrow{(\mathbf{g})} s_0 & & \\
s_0 \xrightarrow{(\mathbf{g})} a \;\Rightarrow\; a \xrightarrow{(\mathbf{t})} s_0 & & \\
\\
s_0 \xrightarrow{(\mathbf{r})} a \;\wedge\; s_1 \xrightarrow{(\mathbf{w})} a \;\Rightarrow\; s_0 \xrightarrow{(\mathbf{r})} s_1 & & \textit{transitivity} \\
s_0 \xrightarrow{(\mathbf{t})} a \;\wedge\; s_1 \xrightarrow{(\mathbf{g})} a \;\Rightarrow\; s_0 \xrightarrow{(\mathbf{t})} s_1 & &
\end{array}
$$

Figure 5: Additional closure rules for *de-facto* access

2.3 Weaknesses

From the standpoint of operating system modeling and implementation, the *take-grant* model has two weaknesses:

- It is insufficiently expressive. Conventional page tables may be viewed as capability structures, but

the *take-grant* model does not account for their access properties. Some method of handling indirect access is required.

- It is insufficiently restrictive. It cannot describe access graphs in which connected subgraphs can be shared in transitively read-only form.

The first problem is reasonably self-explanatory, but the second problem may benefit from an illustrating example.

Suppose subject S_0 builds a graph of objects as shown in Figure 8(a). S_0 wishes to allow other subjects to access the elements of this graph in read-only fashion, but does not wish these other subjects to be able to modify the objects or use them as a vehicle for communication between subjects. At the same time, S_0 wishes to *retain* the ability to modify the elements of this object graph. To protect $O1$, the **grant** and **write** rights must be clearly removed when handing out capabilities to the $O1$, which yields the structure shown in Figure 8(b). The problem is that no good choice of access rights exists for the capabilty to $O2$.

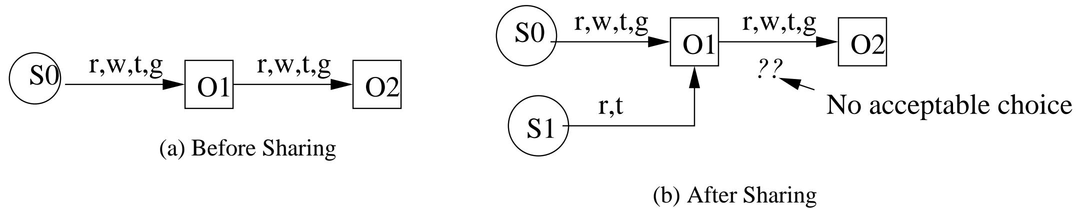

Figure 6: A shared object graph.

In the general case, we may wish the provider and the recipient to share an arbitrary object graph that the provider is free to update but the client can only examine. In this situation, the client must be able to extract capabilities from the object graph, but these capabilities must be restricted to read-only form. What is desired is a form of access that provids transitive read-only access to S_1 without altering the graph as seen by S_0. 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 The Diminish Operator

The *diminish* operation is a form of capability copy that weakens the transferred capability as the copy is performed. If A is a set of access rights, then

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

That is, the only rights that are preserved by the diminish transformation are the **dimtake** and **read** rights. The net effect of this is that the diminish operation enforces *transitive* read-only access.

This operation is authorized by two new access rights: $\mathbf{dimtake}$ and $\mathbf{dimgrant}$ (abbreviated $\mathbf{dt}$ and $\mathbf{dg}$). As a sanity condition, we will require that $\mathbf{take}$ access should imply $\mathbf{dimtake}$ access and $\mathbf{grant}$ access should imply $\mathbf{dimgrant}$ access. The closure rules associated with these rights are shown in Figure 7. Note that the ability to perform a diminished take does not imply full-fledged grant authority in the opposite direction. Instead, it implies a form of diminished grant authority.

$$
\begin{array}{rclcl}
 & & s_0 \xrightarrow{\mathbf{dt}} a & \Rightarrow & s_0 \xrightarrow{(\mathbf{dg})} a \quad \textit{base} \\
 & & s_0 \xrightarrow{\mathbf{dt}} a & \Rightarrow & s_0 \xrightarrow{(\mathbf{dg})} a \\
 & & s_0 \xrightarrow{(\mathbf{dt})} a & \Rightarrow & a \xrightarrow{(\mathbf{dg})} s_0 \quad \textit{symmetry} \\
s_0 \xrightarrow{(\mathbf{dt})} a & \wedge & a \xrightarrow{\alpha} b & \Rightarrow & s_0 \xrightarrow{dim(\alpha)} b \quad \textit{transitivity} \\
a \xrightarrow{(\mathbf{dg})} b & \wedge & a \xrightarrow{\alpha} c & \Rightarrow & b \xrightarrow{dim(\alpha)} c
\end{array}
$$

Figure 7: Additional rules with *diminish*

With the diminish operator in place, there is now a straightforward resolution to the problem of graph sharing: the subject S_1 can be given diminished access to the shared object graph (Figure 8).

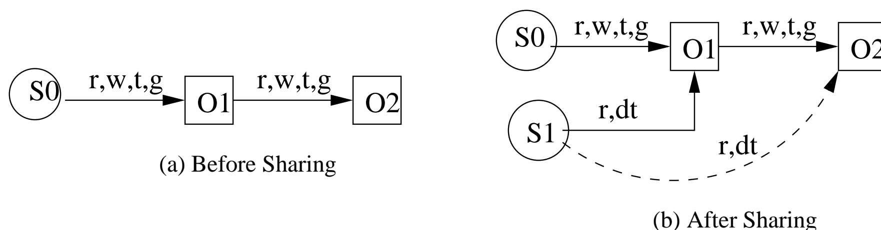

Figure 8: Executable and symbol table using diminish.

3.2 Indirection

The diminish right provides a solution to read-only graph sharing, but it does not allow us to account for one of the most common forms of capability protection in use today: paged memory management. A page table entry is a capability protected by the operating system. The *take-grant* model does not allow us to account for the resulting data access rights seen by the application, nor does it allow us to account for the behavior of application-level memory managers as used in Mach [GD91], EROS [SSF99], L4 [Lie93], and similar systems. The missing link is that the hardware distinguishes between the ability to read data and metadata: an application can read its own data, and can observe the effects of changes to its metadata (the mappings), but it cannot directly read the mapping structures themselves or fetch the capabilities that they contain. To address this, the model requires a mechanism that describes indirection.

Indirection rights introduce the ability to perform reads across multiple subjects or objects. The *take-grant* $\mathbf{read}$ right is divided into $\mathbf{read\ access}$ and $\mathbf{read\ indirect}$, respectively denoted $\mathbf{r}$ and $\mathbf{ri}$. Under the

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

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

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] All other rights are similarly divided, and the fi nal transitive closure rules for the *diminish-take* model are shown in Figure 9.

The fi nal form of the *diminish* operation is:

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

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. Nor are they weakened forms of the **take** right – the referencing subject does not ever possess copies of the indirect capabilities.

3.3 Decidability

An earlier paper [SW00] provides a verifi cation proof of a decision procedure for testing confi nement built on a version of the *diminish-take* model lacking indirection rights. The model described here more fully corresponds to the EROS implementation. 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.

The linear time decidability of *take-grant* demonstrated by Jones, Lipton and Snyder [JLS76] is not altered by the introduction of a fi nite number of distinguished **take** and **grant** rights. Therefore, the addition of **dimtake** and **dimgrant** does not materially alter their decidability result. In addition, observe that the **dimtake** and **dimgrant** rights are strictly less powerful than the **take** and **grant** rights. The transitive safety effects introduced by these rights are therefore strictly less than those of the pre-existing **take** and **grant** rights. Finally, note that indirect rights do not alter the convergence of the decidability result. A path consisting of an indirect right *x***i**:*x* sequence converges exactly as fast as path consisting of **t**:*x*.

It follows that for every safety problem in *diminish-take*, there exists an equivalently complex safety problem in the *take-grant* model that can be constructed by replacing all indirect rights with **take** and all diminished rights with their undiminished equivalents. While the safety outcome will in some cases be different, the resulting problem is decidable in linear time.

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 operating system to yield a system that is simultaneously flexible and securable. We sketch several of the design patterns that are pervasively used in EROS and to show how the *diminish* operation and indirection rights play a pivotal role.

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

$$
\begin{array}{lllll}
 & s_0 \xrightarrow{\mathbf{r}} a & \Rightarrow & s_0 \xrightarrow{(\mathbf{r})} a & \textit{base} \\
 & s_0 \xrightarrow{\mathbf{w}} a & \Rightarrow & s_0 \xrightarrow{(\mathbf{w})} a & \\
 & s_0 \xrightarrow{\mathbf{t}} a & \Rightarrow & s_0 \xrightarrow{(\mathbf{t})} a & \\
 & s_0 \xrightarrow{\mathbf{g}} a & \Rightarrow & s_0 \xrightarrow{(\mathbf{g})} a & \\
 & s_0 \xrightarrow{\mathbf{dt}} a & \Rightarrow & s_0 \xrightarrow{(\mathbf{dt})} a & \\
 & s_0 \xrightarrow{\mathbf{dg}} a & \Rightarrow & s_0 \xrightarrow{(\mathbf{dg})} a & \\
 & & & & \\
 & s_0 \xrightarrow{(\mathbf{r})} a & \Rightarrow & a \xrightarrow{(\mathbf{w})} s_0 & \textit{symmetry} \\
 & s_0 \xrightarrow{(\mathbf{w})} a & \Rightarrow & a \xrightarrow{(\mathbf{r})} s_0 & \\
 & s_0 \xrightarrow{(\mathbf{t})} a & \Rightarrow & a \xrightarrow{(\mathbf{g})} s_0 & \\
 & s_0 \xrightarrow{(\mathbf{g})} a & \Rightarrow & a \xrightarrow{(\mathbf{t})} s_0 & \\
 & s_0 \xrightarrow{(\mathbf{dt})} a & \Rightarrow & a \xrightarrow{(\mathbf{dg})} s_0 & \\
 & s_0 \xrightarrow{(\mathbf{dg})} a & \Rightarrow & a \xrightarrow{(\mathbf{dt})} s_0 & \\
 & & & & \\
s_0 \xrightarrow{\mathbf{ri}} a \quad \wedge & a \xrightarrow{(\mathbf{r})} b & \Rightarrow & s_0 \xrightarrow{(\mathbf{r})} b & \textit{transitivity} \\
s_0 \xrightarrow{\mathbf{wi}} a \quad \wedge & a \xrightarrow{(\mathbf{w})} b & \Rightarrow & s_0 \xrightarrow{(\mathbf{w})} b & \\
s_0 \xrightarrow{\mathbf{ti}} a \quad \wedge & a \xrightarrow{(\mathbf{t})} b & \Rightarrow & s_0 \xrightarrow{(\mathbf{t})} b & \\
s_0 \xrightarrow{\mathbf{gi}} a \quad \wedge & a \xrightarrow{(\mathbf{g})} b & \Rightarrow & s_0 \xrightarrow{(\mathbf{g})} b & \\
s_0 \xrightarrow{\mathbf{dti}} a \quad \wedge & a \xrightarrow{(\mathbf{dt})} b & \Rightarrow & s_0 \xrightarrow{(\mathbf{dt})} b & \\
s_0 \xrightarrow{\mathbf{dgi}} a \quad \wedge & a \xrightarrow{(\mathbf{dg})} b & \Rightarrow & s_0 \xrightarrow{(\mathbf{dg})} b & \\
 & & & & \\
s_0 \xrightarrow{(\mathbf{r})} a \quad \wedge & s_1 \xrightarrow{(\mathbf{w})} a & \Rightarrow & s_0 \xrightarrow{(\mathbf{r})} s_1 & \\
s_0 \xrightarrow{(\mathbf{t})} a \quad \wedge & a \xrightarrow{\alpha} b & \Rightarrow & s_0 \xrightarrow{\alpha} b & \\
s_0 \xrightarrow{(\mathbf{g})} a \quad \wedge & s_0 \xrightarrow{\alpha} b & \Rightarrow & a \xrightarrow{\alpha} b & \\
s_0 \xrightarrow{(\mathbf{dt})} a \quad \wedge & a \xrightarrow{\alpha} b & \Rightarrow & s_0 \xrightarrow{dim(\alpha)} b & \\
a \xrightarrow{(\mathbf{dg})} b \quad \wedge & a \xrightarrow{\alpha} c & \Rightarrow & b \xrightarrow{dim(\alpha)} c &
\end{array}
$$

Figure 9: Final *diminish-take* closure rules

EROS [SSF99] is a 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 vector containing 32 capabilities.

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. In practice, this test is run incrementally as the initial capabilities are added to the "in progress" constructor, and the test result is remembered. There is no check required at instantiation time.

In the absence of the **dimtake** authority, it would be necessary to explicitly copy object graphs, reducing the authority of each capability in the graph to satisfy the overall requirement that the graph convey no mutate authority. This would significantly complicate the constructor logic, and it would eliminate the possibility of other design patterns in which the object graph remains mutable by one party while readable by the other. A client can be given diminished access to a terminal capabilities database in a way that leaves the administrative program free to update the database.

An important issue to note about the constructor is that it can make a confinement judgement – and therefore a safety judgement about the confined subgraph – in constant time using only local information. While the linear-time decidability result [JLS76] is theoretically ground breaking, it has limited application in practice. In practice, there is no agent that can sensibly be authorized to perform the required global analysis of the

system protection graph. Even if we were prepared to accept such a high-risk agent, the protection graph of a real system that is executing important programs cannot be frozen long enough to perform the analysis, and in any case the execution of the analysis itself necessarily requires updates to the protection graph as capabilities are temporarily copied in order to perform the examination. The solution using the tests above avoids all of these pitfalls.

4.2 Memory Mapping

EROS address spaces are constructed by building a tree of nodes whose leaf objects are pages (Figure 10). An ordinary node capability conveys $\{\mathbf{ri},\mathbf{wi},\mathbf{ti},\mathbf{gi},\mathbf{t},\mathbf{g}\}$ authority. A read-only node capability conveys

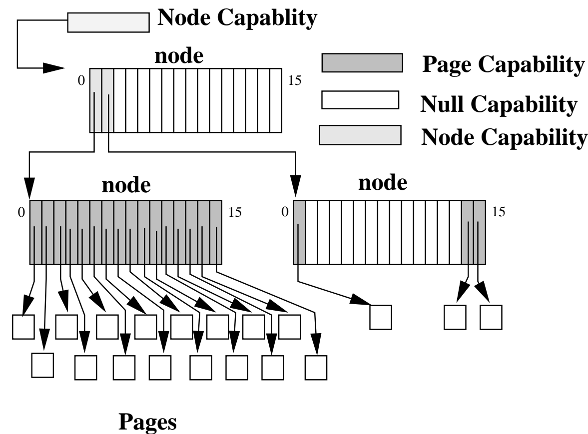

Figure 10: An EROS address space

$\{\mathbf{ri},\mathbf{ti},\mathbf{t}\}$ authority. A segment capability (the name comes from Multics [SCS77]) conveys $\{\mathbf{ri},\mathbf{wi},\mathbf{ti},\mathbf{gi}\}$ or $\{\mathbf{ri},\mathbf{ti}\}$, according to whether it is read-write or read-only.

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 11). Such nodes contain a process capability to the page 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 on behalf of the faulting process. This message includes a node capability to the owned node.[3]

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. A reduced node capability is created conveying $\{\mathbf{ri},\mathbf{dt}\}$ authority to the existing (mutable) node tree.

[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.

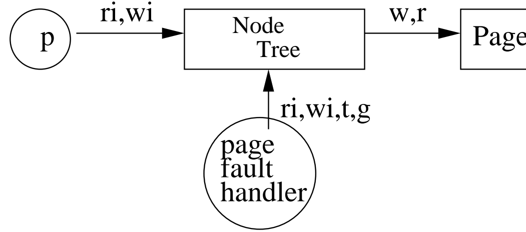

Figure 11: Address fault flow

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 12).

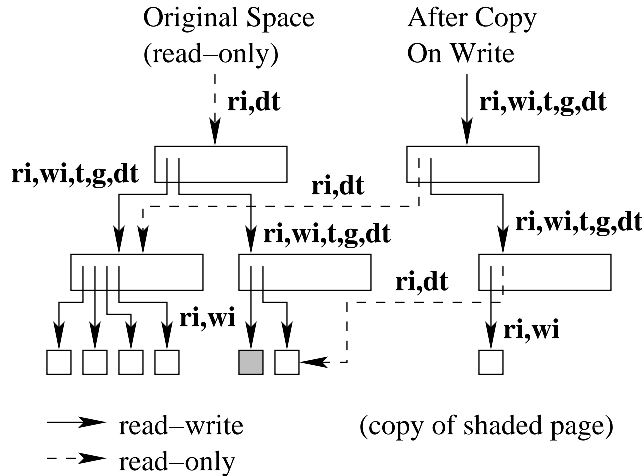

Figure 12: Virtual copy spaces

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 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]. 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. Subsequent analysis [KL86] makes clear that Boebert's claim applies only in systems where capability and data transmission cannot be distinguished by the access control mechanism 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 distinguish between **read** and **take** in the protection system, and 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 $\{\mathbf{ri}, \mathbf{r}, \mathbf{dti}, \mathbf{dt}\}$. This 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, or the use of non-trusted visualizers in a in integrated development environment for secure software.

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.

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. There has been extensive work on modeling non-capability access control systems.

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 multilevel 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 difficult to understand unless you already know how it works. 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. The supervisor described by Dennis and van Horn [Dv66], and the PDP-1 system that followed from it, appears to been the first design to actually separate data read and capability read access rights. There are indications of comparable separation in the Chicago Magic Number machine [Fab67, She68, Yng68]. The Cambridge CAP [WN79] system unquestionably separated data read and capability read access rights. A careful study of any of these machines in conjunction with the Burroughs system 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 **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.

5.3 PSOS

PSOS, the Provably Secure Operating System [NBF+80, NF03], is a capability-based system that supports an access model similar to the one described here. PSOS incorporates a hardware-implemented capability weakening primitive similar in some respects to the diminish operator; existing access rights can be selectively retained. It is not clear from the literature whether the application of this operation was imposed by an access right or was discretionary.[4] The essential innovation in the **dimtake** right is that the application of the diminish function is not discretionary: there is no sequence of operations starting with a diminished capability that can result in a write-authorizing capability.

PSOS also implemented a concept called "store permissions," [FN79] a mechanism that could selectively control which capabilities can be stored to which capability segments. Among other uses, this feature could be used to enforce write down prohibitions. Because there is no "filtered capability read" restriction, this mechanism appears insufficient to manage the sharing problem posed by traversable object graphs such as Figure 6.

5.4 Other

Chander, Dean, and Mitchell have compared access relationships across a number of access control mechanisms including "Lampson capabilities" (capabilities in the sense that we discuss them here) [CDM01]. They conclude incorrectly that the capability model (capabilities in the sense that we discuss them here) is equivalent in power to an access control list system having identical fundamental access rights. This result is true within their formalism, but their formalism does not accurately consider the impact of the predicating conditions that govern legal graph transformations in the respective systems.

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 [BS79] 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.

[4] I will try to clarify this with Peter Neumann before final publication.

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

[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.

[Boe03] Earl Boebert. *Review of "Capability Myths Demolished"*, 2003. Publicly quoted with permission. Full text available at http://www.eros-os.org/pipermail/cap-talk/2003-March/001133.html.

[BS79] Matt Bishop and Lawrence Snyder. 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.

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

[CDM01] A. Chander, D. Dean, and J. Mitchell. A State-Transition Model of Trust Management and Access Control. In *14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, June 2001.

[DoD85] *U.S. Department of Defense Trusted Computer System Evaluation Criteria*, 1985.

[Dv66] J. B. Dennis and E. C. van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–154, March 1966.

[Fab67] R. Fabry. A User's View of Capabilities. *ICR Quarterly Report*, pages C1–C8, November 1967.

[FLH83] W. Frantz, C. Landau, and N. Hardy. GNOSIS: A Secure Operating System for the '90s. *SHARE Proceedings*, 1983.

[FN79] R. Feiertag and P. Neumann. The Foundations of a Provably Secure Operating System (PSOS). In *Proc. 1979 National Computer Conference*, pages 329–334, 1979.

[GD91] David B. Golub and Richard P. Draves. Moving the Default Memory Manager out of the Mach Kernel. In *Proc of the Usenix Mach Symposium*, pages 177–188, November 1991.

[Gon89] Li Gong. A Secure Identity-Based Capability System. In *IEEE Symposium on Security and Privacy*, pages 56–65, 1989.

[Har85] Norman Hardy. The KeyKOS Architecture. *Operating Systems Review*, 19(4):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.

[KL86] Richard Y. Kain and Carl E. Landwehr. On Access Checking in Capability Systems. In *Proc. 1986 IEEE Symposium on Security and Privacy*, pages 95–100, 1986.

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

[Lie93] Jochen Liedtke. A Persistent System in Real Use – Experiences of the First 13 Years. In *Proc. 3rd International Workshop on Object-Orientation in Operating Systems*, pages 2–11, Asheville, N.C., 1993.

[MYS03] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability Myths Demolished. Technical Report Report SRL2003-02, Systems Research Laboratory, Department of Computer Science, Johns Hopkins University, mar 2003.

[NBF+80] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A Provably Secure Operating System: The System, Its Applications, and Proofs. Technical Report Report CSL-116, Computer Science Laboratory, may 1980.

[NF03] P.G. Neumann and R.J. Feiertag. PSOS Revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, Las Vegas, Nevada, December 2003.

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

[Ree96] Jonathan A. Rees. A Security Kernel Based on the Lambda-Calculus. Technical Report AIM-1564, 1996.

[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.

[She68] J. H. Shepherd. Principle Design Features of the Multi-Computer. *ICR Quarterly Report*, pages C1–C13, November 1968.

[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.

[WBDF97] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint-Malo, France, October 1997.

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

[Yng68] V. H. Yngve. The Chicago Magic Number Computer. *ICR Quarterly Report*, pages B1–B20, November 1968.