SECURITY IN KEYKOS

S.A. RAJUNAS, N. HARDY, A.C. BOMBERGER, W.S. FRANTZ, C.R. LANDAU

Key Logic, Cupertino, California

ABSTRACT

KeyKOS™** is a capability-based system which was designed to meet the performance, reliability, and security goals of the commercial computer service marketplace. KeyKOS's architecture combines several unique features to achieve simple and effective protection mechanisms with a very small amount of privileged code. Particular emphasis is placed here on the protection mechanisms which support security; while some general description of the system is included, a more complete description of the architecture is given elsewhere[1]

INTRODUCTION

KeyKOS consists of a kernel and supplementary code. The KeyKOS kernel is currently implemented in software, but in concept it is an extension of the machine architecture and as such could be implemented in firmware and/or hardware. The present implementation runs on IBM and Amdahl hardware. Any virtualizable architecture[2] with a large virtual address space and demand paging would be a suitable target. KeyKOS might also be ported to other architectures.

KeyKOS is not a traditional operating system; it bears some resemblance to a control program (it manages resources). In the current implementation, the kernel is written in assembler, is unswappable, and runs in supervisor mode with real addresses. The kernel and supplementary code present a different architecture to the programmer from that provided by the bare hardware. KeyKOS supports CMS[3], and currently most software development is done using CMS as the development environment. KeyKOS provides an excellent run time environment for transaction-based applications, and KeyKOS has been running in support of commercial applications since 1983.

KeyKOS is a message-passing system. KeyKOS is also an object-oriented system, where certain primitive objects are supported by the kernel and application designers can build their own objects out of system-defined objects. These user-defined objects are much larger than Smalltalk[4] objects.

Like all capability systems, KeyKOS must prevent the forging of capabilities. This is accomplished not by relying on tagged memory but rather by storing capabilities separate from data.

DEFINITIONS

Capabilities in KeyKOS are called keys, and the reader may substitute "capability" anywhere the term "key" is used. However, in KeyKOS there are many different types of keys (see Table 1). Keys carry several different types of authority, not merely read-only and read/write, and designate objects of all types.

Some of the terminology used in KeyKOS is unique, and the unfamiliar terms have been cited as a source of confusion by individuals first learning about the system. There have been suggestions that we should use "more standard terminology," but each system uses its own terminology. Capabilities are "access descriptors" in the Intel 432[5], and "pointers" in System/38[6]. (See Levy[7] for a survey of implemented capability systems). It has also been suggested that we use the terminology popularized by, say, Hydra[8], certainly the most published system. But KeyKOS is not a copy of Hydra. KeyKOS has different basic design concepts, described by different terms; the terms are different because the concepts are different.

The convention used in this paper is that KeyKOS terms are printed in bold type when they are first defined.

Objects

The basic entities in KeyKOS are keys, pages, and nodes. **Keys** are capabilities, and designate pages, nodes, other primitive objects implemented by the

** Patent Pending.

kernel, or compound constructs made up of pages and nodes. **Pages** consist of 4096 bytes, may contain code or data, but cannot contain keys. Pages may be regarded as permanent storage; KeyKOS implements a single level store. A **node** consists of 16 slots (may be thought of as capability registers), which contain keys, but not code or data. Nodes are also permanent storage.

The three fundamental objects in KeyKOS are domains, meters, and segments. **Domains** are roughly analogous to processes, **meters** represent resources (such as CPU time), and **segments** define ranges of virtual addresses.

A domain is a fundamental object which obeys some code (program) and holds some keys that the code can reference. It is important to note that "process" is not a technical term in KeyKOS. While someone new to KeyKOS may analyze a KeyKOS scenario by envisioning processes, there are no such entities in the implementation. The closest concept to "process" in KeyKOS is the domain. We include what is commonly meant by "process", along with address space and access rights, in the concept of domain. A domain is not just a privilege state, as it is in some systems. In KeyKOS one cannot speak of a process moving from domain to domain; the closest analogy is a thread of control passing from one domain to another.

Programmers may create their own objects from one or more domains. Each domain contains algorithms, data, state information, and access rights. In the current implementation, a domain is three nodes (the root node, the registers node, and the keys node), and the keys found in the slots of the root node have special interpretations (see KeyKOS Principles of Operation[9] for details). The slots in the root node contain keys which designate the domain's address segment, the domain's keeper and its meter (described below), status information (domains may be **available**, **running**, or **waiting**), most of the PSW, etc. The registers node contains the general purpose registers and the floating point registers. The keys node contains keys accessible to the program in the domain. The keys in the domain's root node in general are not accessible to the program the domain obeys, any more than a process header or task control block is accessible to the process or task in a conventional operating system. They are data structures maintained on behalf of the domain or process or task by the operating system. The difference is that in KeyKOS, the domain *could* be given a key to any root node (including its own), and it could affect only that node for which it had a key. In a conventional system, the program would have to be privileged to access its own process header and it would have not merely that privilege but *all* privileges (the pervasive privilege problem).

In KeyKOS terminology, keys in the domain's root node are said to be **elements** of the domain. Keys in the keys node are said to be **held** by the domain, and their use is determined entirely by the program obeyed by the domain. In fact, a program in a domain does not actually handle any keys (it can only address the contents of pages); it refers to the keys it holds by their slot numbers in the keys node and the kernel interprets the requests for access.

Meters are nodes which represent resources that may be consumed by a domain. As a domain uses CPU time (currently the only metered resource) the meter counts down. When it reaches zero, the domain stops, but no information is destroyed and the domain may be restarted by adding more resources. There is a primordial meter from which all meters are descended, and in the hierarchy of meters, meters lower in the tree (inferior) depend on meters higher in the tree (superior), so any use of CPU time is measured and constrained not only by the immediate meter but also by its superiors.

KeyKOS uses the concept of the single-level store. No user written code needs any concept of secondary storage. Also, all access to data is immediate. A domain's address space is implemented by a segment key; a segment consists of other segments or pages. One domain cannot access data in another domain -- it can only access data on pages in its own address space. Pages *can* be shared, and even shared with different authority, but only if prearranged. Either the domains' address spaces must have been built with shared pages or the domains were set up to be able to exchange page keys and insert them in their address spaces.

Because of the single-level store, KeyKOS has no "files," but domains serve a similar role. Domains are as long-lived as files in traditional systems; they continue to exist until deliberately deleted. **Record collections** are domains which perform the specialized function of relating symbolic names to keys. Some record collections function as directories; others serve to emulate the functionality of IBM file access methods.

Key Invocation

The basic operation in KeyKOS is key invocation. There are three ways to invoke a key: **FORK**, **CALL**, and **RETURN** (the traditional meanings apply). Control is passed either to the kernel (if it was a key the kernel implements) or to another domain via the kernel (if it was a gate key). The KeyKOS equivalent to interprocess communication is a **gate key** invocation between domains which passes a message consisting of up to four keys, a byte string, and a parameter word.

The key invocation mechanism unifies message passing and procedure call functionality. An argument has been made[10] that operating systems fall into either the message passing category (characterized by a small, static number of processes with explicit communication paths) or the procedure call category (characterized by a large, rapidly changing number of small processes with process synchronization based on shared data) and that these categories are duals of each other. However, KeyKOS has characteristics of both. Since keys can be passed within messages, communication channels can change dynamically. Domains are composed of only three nodes, and they exist in large numbers. Segments may be as small as one (possibly shared) code page, and optionally one page of private address space. The key invocation which transfers control from one domain to another, while also passing a message, is relatively cheap. The cost is approximately 200 IBM 370 instructions, which is not excessive when one considers that this completely changes not only the executing code but also the address space, the access rights, the operating system environment (see the discussion below on keepers), and the resource utilization policy.

It is important to realize that keys are inscrutable. One may think one holds a key directly to a domain, but there may actually be an additional level of indirection transparent to the user. A domain which holds a gate key (see Table 1 for the types of keys and their authority) has, in general, no way to know anything about the object designated by that key. And once the key has been invoked, control has been passed. The domain which receives control and begins executing has no way of knowing who called it, has no authority over the caller, and does not inherit the authority possessed by the caller. The only authority it acquires is keys passed to it (if any). To access the services of another object a domain commonly invokes the key to that object with a CALL, and the kernel automatically generates a resume key to the CALLer. While the CALLed domain may copy the resume key, upon invocation of the resume key any duplicates of the resume key become invalid (by action of the kernel).

Exception Handling

Exception handling is performed by **keepers**. The kernel provides no fixed exception handling policy, so the policy can be tailored to the application. There are domain, meter, and segment keepers, and many different policies may be supported simultaneously. The domain keeper (designated by the domain's root node) is implicitly invoked on domain faults, such as overflow, underflow, divide check, supervisor call (SVC). The domain keeper may be used to emulate the interface of a traditional operating system. The meter keeper, which is designated by the meter (which in turn is designated by the domain's root node), is implicitly invoked when the resource counter goes to zero. The segment keeper is implicitly invoked on memory faults. Memory faults occur on encountering invalid addresses and write protection violation, *not page faults* (which are handled by the kernel). The segment keeper receives, when CALLed, a node key to the segment so it can fetch from and store into slots containing memory keys (segment or page keys). The kernel builds a domain's page tables based on the segment key. Setup or modification of the address space requires a node key (a particular type of authority described below) to the segment.

In the case of the segment keeper getting a node key to the faulting segment, and in the case of the domain keeper getting a domain key, rights amplification has taken place.

Summary of Terms

In summary, there are nine basic concepts in KeyKOS. The most basic are data (actually includes instructions and data), pages, keys, and nodes. The fundamental objects are domains, meters, and segments. The basic operation upon keys is invocation, and the exception handling is performed by keepers.

THE PERCEIVED PROBLEM

By now it can readily be seen that KeyKOS is a true capability-based system, wherein possession of a key guarantees access. It can also be seen that KeyKOS has a unique design, although the benefits may not yet be apparent. To illustrate further the distinction between KeyKOS and other capability-based systems, we will examine the problem of implementing access control policies on capability-based systems.

Capability-based systems readily lend themselves to discretionary access control. They also combine the right to access an object with the right to grant access to that object. Non-discretionary access control policy dictates that access cannot be granted across certain boundaries. Therefore, it has been concluded by many that capability-based systems are unable to support non-discretionary access control.

The access control issues commonly cited as problems in capability systems are propagation (access ==> grant access), review, revocation, and migration. These issues are relevant to the problem of enforcing an access control policy, and particularly a non-discretionary access control policy. For example, DoD non-discretionary access control policy requires that an individual be cleared for the appropriate security level before he access data classified at that level, that there must be records kept of who can access what data, that access be revocable, that output be accurately labelled, etc.

Table I: Types of keys implemented within the kernel.

key type	object designated	type of authority
node	node	store into or fetch from any slot in a node
fetch	node	fetch from any slot in a node
sense	node	fetch from any slot in a node, but attenuate the rights of the key thus fetched
domain	node	view the node as a domain root; allow fetch and store into some slots, and fetch from some others
gate	node	view the node as a domain root and cause transfer of control to the program in that domain when invoked
segment	node	no authority to store into or fetch from the slots in the node, only authority to access the pages designated via the keys in those slots
meter	node	authority to consume the resources defined by the contents of the slots of the node, but not to change the contents of the slots
page	page	authority to access the data in a page; may be read-only or read-write
data	integer	no authority, but immediately holds a non-negative integer.
range	pages or nodes	authority to manage a group of pages or nodes
device	device	authority to access physical device
misc	utilities	authority to invoke certain kernel objects

To understand how KeyKOS controls access, first consider the problem of restricting access in capability systems in general. Various mechanisms have been proposed, and some actually implemented, to restrict the flow of capabilities. There have been ideas like having uncopiable objects and/or uncopiable capabilities. Conditional capabilities and even temporary capabilities have been proposed. But none of the complicated mechanisms proposed for or implemented on capabilities are convenient to use or even convincingly adequate to restrict the flow of capabilities in the desired manner while still enabling programmers to get work done. The desired goal is *not* complete isolation (even if it could be achieved); it is controlled sharing.

A common misconception is that capability systems could be "fixed" by merely adding a security field to each and every capability and checking this field upon access to the object. This is a poor idea for a number of reasons. Capability systems practice decentralized control; presenting a capability guarantees access. Violating this principle by introducing centralized checking degrades performance and confuses the conceptual clarity of the policy/mechanism dichotomy. It can be argued that at some lower level of realization (interpretation) capabilities pass through some centralized mechanism; but it can be readily seen that this is not the level at which policy should be implemented, since the items to which access is to be controlled do not exist in any recognizable form at the low level of abstraction where capabilities are realized.

Since the KeyKOS kernel is designed as an extension to the machine architecture, it is clear that policy does not belong in the kernel, although the kernel must be trusted to function correctly. As described in the next section, the totality of protection mechanisms responsible for enforcing a security policy in KeyKOS (the trusted computing base[11]) consists of the kernel and some domain code. The code concerned with actually implementing security *policy* does not reside in the kernel; it resides in domains. The *mechanisms* which support that policy are part of the kernel. Thus KeyKOS is a suitable base on which to implement a wide variety of access control policies, including DoD policy.

Historically, people wanted to implement policy within the lowest level of mechanism in the system. It was the only level they trusted to work reliably. Unfortunately, the protected objects do not exist at that level of abstraction. It seems a better approach to get access control out of the kernel, and implement it at a higher level. The approach used in KeyKOS is to allow keys to be freely copied and passed but to provide an encapsulation mechanism to restrict the flow of keys.

SUPPLEMENTARY CODE

The code in KeyKOS whose correct functioning is critical to implementing a security policy consists of the kernel (25,000 lines of assembler code) plus 25,000 lines of domain code. The code in domains includes the official space bank, the factory creator, the receptionist (which performs authentication), the local user directory (a record collection) and the domain creator creator. (Domain creator creator is an unfortunate term, but it accurately describes the functionality and is no worse that Smalltalk's class class.) The kernel has been described; this section concentrates on the portion of the trusted computing base which resides in domains and discusses how it contributes to the overall security of the system.

Space Banks

Space resources (pages and nodes) come from **space banks**. Each user has a gate key to an official space bank. Official space banks are guaranteed to obey certain principles (and trusted to do so), such as only giving a page or node out once, and returning promptly either the desired keys to newly created pages or nodes or else returning an error code indicating that there is insufficient storage. Pages and nodes may be recycled by returning them to the space bank from which they were received. Space banks, like meters, are all descended from primordial space banks which own all the storage in the system, and like meters, may have descendants. There is no common view of how much total storage exists in the system, or how much is still unused. As in a virtual machine environment, the user's view is limited to *his* resources. And as is typical in KeyKOS, in situations where more authority is required, it can be granted selectively.

Data Integrity

An optional type of storage is storage that is replicated on different disk drives. This type of storage has been found to be extremely useful for support of commercial applications, where reliability is important. The failure of a single disk or disk controller (if the storage were not replicated) would result in the failure of some domains.

Another type of failure that may occur is system crash. Since KeyKOS supports a single-level store, a crash could cause complete amnesia if it were not for the checkpoint/restart facility. All of the system's state (outside of the kernel) is in pages and nodes, which are written to disk. If the pages and nodes in memory have been modified, thus becoming inconsistent with the old versions on disk, they are recorded in the system's checkpoint area on disk when a periodic snapshot is taken (currently the interval is five minutes). User code running in domains is unaware of checkpoints, and the domains restart smoothly from the last checkpoint after a crash. So far this functionality has been found sufficient for the existing commercial applications, but in the future some applications might also want journaling. A journaling facility has been implemented but is not currently used by any commercial application.

Domain Creators and Factories

Each user may have one or more domain creators, which he created using the official domain creator creator. Once a domain has been created, one can install code, data, and keys, and then the user has an object (domain) of his own creation for his own use (or anyone else's, should he be given a gate key to that domain). But to handle the problem of mutually suspicious users, another mechanism, trusted to be tamperproof, is required. That mechanism is the **factory**, and factories are analogous to domain creators except that the objects (domains) that they produce are sealed, which means that the the algorithms are already installed and modification of an object created by a factory requires the mutual consent of the owner of the object and the owner of the factory. In current practice, object creation is done almost entirely by use of factories (which themselves invoke domain creators) rather than by directly using domain creators.

There is one additional new mechanism in KeyKOS that is security relevant: **sensory keys**. In other capability systems, a capability to a complex, user-defined object containing data and capabilities allows one the authority to access and modify the object. In KeyKOS one can imagine a tree of nodes: each node contains keys to data (pages) or other nodes. In an attempt to create what would be effectively a "read-only" key to the compound object, the sense key was developed. Even though the fetch key already existed, it was too strong: one could fetch a node key from a slot in a node in the tree structure and use that to modify a node lower in the tree. A sense key performs much like a fetch key, but the result of invocation is always the weakest (sensory) version of the key being fetched. The sensory version of a node, segment, sense or fetch key to a node is the sense key to that node; the sensory version of a page key is the read-only key to that page, etc. Sensory versions of keys are the most primitive rights attenuation mechanism in KeyKOS.

NEW SOLUTIONS

By now it has become apparent that in KeyKOS nothing is shared unless explicitly made so, that access and communication can only be accomplished via keys and that tree structures built of nodes are

common. It can easily be seen that isolation can be achieved in KeyKOS -- simply divide the universe into separate groups of keys at sysgen, and allow no keys to point between groups. No matter how many new objects and keys are created, or how many levels are added to the trees, there will be no keys that point between the groups.

However, it has already been pointed out that controlled sharing, not isolation, is the goal of access control mechanisms in KeyKOS. It is useful to be able to implement appropriate and restricted sharing between compartments, which means, for example, allowing users in one compartment the ability to read information in another compartment. The gaps in the firewalls between compartments are called **holes**, and they are implemented as part of the functionality of factories. Factories are sources of objects of known discretion (no Trojan horses). The measure of discretion is the set of holes permitted in an object produced by that factory. A hole is a non-sensory key which designates some object outside the compartment. Compartments are produced by factories, and all of the objects which are descendants of a particular factory have no less discretion (no additional holes). Holes reference domain code (not kernel code) which must be trusted. This code is in the domain, outside of the factory product, and referenced via the hole.

A number of KeyKOS facilities contribute to the solution of the propagation, review, revocation, and migration problems. The propagation problem is addressed by sense keys, compartments, and factories. Sense keys convey weakened authority. Compartments restrict the flow of keys. "Holes" in compartments allow restricted communication. Factories create objects with measured discretion.

The problem of achieving review is solved by providing a security administrator (not yet implemented) supported by a database which records the connectivity (currently only a prototype exists). Changes in the connectivity of the system cause changes in the database, which keeps track not only of the current state but also history and the time the change occurred. Revocation is handled by more straight forward means. Access to an object can be eliminated by destroying the object, but this is not always desirable. A less drastic and only slightly more expensive way to revoke access is to provide an additional level of indirection in the access path to the object. The intermediate object can then be destroyed, effectively revoking access. An important point to be kept in mind is that if exchange of keys is permitted, the connection cannot be severed. In general, if you need to be sure the connection can be revoked, key exchange cannot be allowed, although in some cases an extra level of indirection to monitor the exchange is sufficient control. Finally, migration is a non-problem in KeyKOS. Keys and data reside in separate spaces; there is no intermingling. And since KeyKOS is a single-level store system, with system-wide checkpoints, no keys ever migrate outside the system.

KEYKOS ACHIEVES CONFINEMENT

KeyKOS was designed to handle the mutually suspicious user problem, and does so without resorting to "memoryless" processes[12]. Domains have retained state between invocations, although communication can be restricted. In a KeyKOS system there is no concept of "file" or even "temporary file." Consider the traditional scenario, wherein the owner of a proprietary algorithm wants to make his algorithm available for others to use but not to copy, and the owner of proprietary data wants to use the algorithm to process but not to copy his data. The algorithm owner can build a factory which will produce objects (domains) which obey his algorithm; then he can provide the key to this factory to those interested in the service. The data owner who receives the factory key can pass that key to a factory of his own for a discretion check which assures that the product of the factory has no keys back to the algorithm owner (it actually can only verify that the keys present are benign, e.g. acceptable miscellaneous keys, sensory keys, or that they are not). If the factory passes the test, the data owner can have an object created from space provided by his space bank (and thus owned by him). He gets back a key to his object, which he can then use without fear of a Trojan horse.

To examine the factory certification in somewhat more detail, consider the situation when the user wishes the objects produced by the factory to have no holes. He passes the factory key to a factory of his own. Say the suspect factory, which produces Trojan horse objects, contains a key to its owner's space bank, from which the factory products would get the storage to make copies of the user's data. The suspect factory will not be certified, because the user's factory will note the presence of a key which is not permissible.

In KeyKOS, where domains are generally isolated unless specifically allowed to communicate, the problem of achieving total or selective confinement reduces to control of communication. And since there is no arbitrary mixing of keys and data, a factory passed a key to another factory for the purpose of verifying its discretion knows exactly where to look to find keys: in the keys node.

In approaching the problem of access control, the point is not how an access control policy is encoded in a system (even though some people argue that it can only be done at the level of the most basic mechanisms in the system); the point is that the

access control policy is unavoidably enforced as encoded.

There are no explicit security labels *within* KeyKOS because they are not necessary. While they would be necessary in a conventional operating system with centralized checking, they are not appropriate in a capability system. It is the *connections* you need to trust in a capability-based system, not the *labels*. And based on the trusted connections, output can be correctly labelled with its security classification.

Internal labels on objects are important in systems where one can somehow name an object (for example, access by knowing an address or guessing the name of a file). If the only way you can access an object is by its true name (capability), then the important issue is with whom one can communicate who possesses and could transmit the true name. So connectivity, not labels, is the important issue, and KeyKOS provides effective mechanisms to control connectivity.

Covert Channels

Once it has been seen that a system does not allow direct leakage of information, the next problem is analysis of covert leakage, since covert channels of large bandwidth could be responsible for significant information leakage. Conventional covert channels which were described as "storage channels" (shared files, shared variables) do not exist in KeyKOS, since nothing is shared unless explicitly made shared. The covert channels related to the system clock (traditionally called "timing channels") do exist to some extent.

For example, a domain could attempt to sense how long it took to access a shared read-only page in order to determine whether or not the page had already been resident in memory. Whether the time required to bring in a page is discernible from the scheduling interval is difficult to determine, but we will agree that it is theoretically possible. Similarly, when a domain invokes a gate key to another domain, if the CALLed domain was not "prepared" (in memory and ready to run) some time is added to the usual gate jump overhead and domain execution time which could be sensed by the invoker when control is returned. The busy or not busy state of a shared service domain could also be used as a channel. But in all these cases the activity of other domains tends to obscure the signal.

There are some additional obscure signalling scenarios involving the interaction between interruptible instructions and program event recording, self-modification by interruptable instructions, and an undocumented unprivileged instruction, all of which describe cases wherein programs do not execute deterministically. Since such events can affect a program's execution, they can be sensed.

It has been noted that the covert channels that exist in KeyKOS seem to be consequences of the underlying machine architecture. In particular, the system clock is responsible for the most promising covert channels. These channels could all be closed by virtualizing the system clock. IBM has already made this architectural modification experimentally[13], but it is not available as a product.

SUMMARY

The security achieved by KeyKOS is the result of its basic design concepts. None of the simplicity of the nine basic concepts has to be complicated or changed to enforce a security policy. Any unimplemented security policy features will be implemented in objects, not the kernel.

REFERENCES

[1] Hardy, N. H., "KeyKOS Architecture," Operating Systems Review, September, 1985.

[2] Goldberg, R. P., *Architecture of Virtual Machines*, Proceedings of NCC, Volume 42, AFIPS Press, 1973.

[3] *Virtual Machine/System Product CMS Command and Macro Reference*, SC19-6209, IBM, 1983.

[4] Goldberg, A. and Robson, D., *Smalltalk-80 The Language and Its Implementation*, Addison Wesley, 1983.

[5] Organick, E. I., *A Programmer's View of the Intel 432 System*, McGraw-Hill, 1983.

[6] *IBM System/38 Technical Developments*, IBM GSD G580-0237-1,1980.

[7] Levy, H. M., *Capability-Based Computer Systems*, Digital Press, Bedford, MA, 1984.

[8] Wulf, W. A., Levin, R., Harbison, S. P., *HYDRA/C.mmp An Experimental Computer System*, McGraw-Hill Book Company, 1981.

[9] *KeyKOS Principles of Operation*, KL02, Key Logic, 1985.

[10] Lauer, H. C. and Needham, R. M., "On the Duality of Operating System Structures," Proceedings of the Second International Symposium on Operating Systems, IRIA,

October, 1978, reprinted in Operating Systems Review, 13,2, April, 1979.

[11] Department of Defense Trusted Computer System Evaluation Criteria, CSC-STD-001-83, Department of Defense Computer Security Center, 15 August 1983.

[12] Lampson, B. W., "A Note on the Confinement Problem," Communications of the ACM, 16,10, October, 1973.

[13] Canon, M. D., Fritz, D. H., Howard, J. H., Howell, T. D., Mitoma, M.F., and Rodriguez-Rosell, J., "A Virtual Machine Emulator for Performance Evaluation," Communications of the ACM, 23,2, February, 1980.