Composing Abstractions using the null-Kernel

James B. Litton[ħ,ϕ], Deepak Garg[ħ], Peter Druschel[ħ], Bobby Bhattacharjee[ϕ]
Max Planck Institute for Software Systems[ħ], University of Maryland[ϕ]

Abstract

The trend towards heterogeneous hardware and the needs of specialized application workloads demand new OS abstractions. Deploying new, high-level, well-integrated abstractions in current general-purpose OSs, however, takes years. As a result, new hardware is often exposed using low-level, close-to-the-metal interfaces at the expense of ease of use, portability, safety, and the ability to safely virtualize the hardware. We propose the null-Kernel, a novel OS structuring principle for new and retrofitted existing OSes. The principle revolves around a set of inter-dependent abstract machines (AM) governed by an extensible capability system. When combined with a set of well-chosen, safe-to-use AMs, the null-Kernel enables the rapid deployment of new OS abstractions that expose new hardware capabilities, or exploit existing hardware resources in ways that better meet evolving application demands.

1 Introduction

Abstractions imply choice, one that OS designers must confront when designing a programming interface to expose. Traditional monolithic kernel designers choose a high-level portable interface that necessarily hides many hardware details, whereas on the other extreme, designs such as the Exokernel optimize for performance and low-level hardware access. Abstraction choices have further implications in how easily and quickly new hardware and application models can be supported. In this paper, we describe the null-Kernel, a new model for structuring system software that attempts to relieve OS designers of this choice and enable access to and composition of OS interfaces at different layers of abstraction.

The null-Kernel is designed to address the growing need to easily provide high level programming interfaces to new hardware, such as GPUs, crypto/AI accelerators, smart NICs/storage devices, NVRAM etc., and to efficiently support new application requirements for functionality such as transactional memory, fast snapshots, fine-grained isolation, etc., that demand new OS abstractions that can exploit existing hardware in novel and more efficient ways.

At its core, the null-Kernel derives its novelty from being able to support and compose across components, called Abstract Machines (AMs) that provide programming interfaces at different levels of abstraction. The null-Kernel uses an extensible capability mechanism to control resource allocation and use at runtime. The ability of the null-Kernel to support interfaces at different levels of abstraction accrues several benefits: New hardware can be rapidly deployed using relatively low-level interfaces; high-level interfaces can easily be built using low-level ones; and applications can benefit from being able to use interfaces, and compose abstractions using more than one if necessary, as appropriate.

The null-Kernel capability system can also be used to partition hardware between different components (AMs) that provide different abstractions. For instance, the null-Kernel can be used to simultaneously support a traditional OS that provides a system call interface, and an exokernel that provides low-level access to new hardware. Such a system could easily enable optimizations not possible now: for instance, the BSD Socket interface necessarily implies copying incoming data from kernel to user space memory, and typically an additional copy is incurred when the data is initially copied from the NIC. Zero copy stacks exist, but do not eliminate the initial copy. A null-Kernel that combines a BSD-like AM and an exokernel interface for Smart NICs can be programmed to assemble incoming TCP segments and copy them directly into process memory, bypassing the BSD AM entirely. Applications can use the BSD interface for traditional high-level services while simultaneously benefiting from a very high performance networking stack.

We describe design principles for AMs that go beyond strict partitioning, and enable cooperating AMs to provide additional functionality. These design principles can be used to retrofit existing kernels, allowing applications to simultaneously use the high-level interface they provide while benefiting from additional access to low-level hardware features that were previously hidden by the traditional OS. For instance, cooperating high- and low-level AMs can simultaneously provide virtual memory (high-level) and access to page-access bits (low-level, currently

unavailable). Such a facility could be used to implement new primitives (e.g., software transactional memory) or optimize existing (e.g., garbage collection).

In the next section, we describe the null-Kernel structure, its capability system, and how interfaces provided by AMs can be composed using the null-Kernel. In Section 3 we discuss how an existing kernel can be retrofitted to interface with an exokernel, and provide examples of how new types of application primitives such a hybrid system can support. We discuss related work in Section 4 and conclude in Section 5.

2 The null-Kernel

Figure 1 shows a high-level schematic of the null-Kernel. The null-Kernel architecture decomposes the system into three components: abstract machines (AMs), callers, and the null-Kernel itself.

Abstract machines are software layers that provide specific functionality, and expose a set of operations that *callers* may invoke. In a traditional OS, the kernel is the AM, and this set of operations is the system call interface. Callers are processes or threads, as recognized by the kernel. In a null-Kernel architecture, there may be other layers of software (i.e., different AMs) that provide different interfaces, which would also be available to eligible callers, which may include other AMs.

The null-Kernel, shown in green in the figure, controls access to AM operations by only allowing invocations when the caller presents capabilities with sufficient access rights. The capability structure supported by the null-Kernel is extensible: AMs define new capabilities and specify which access rights are required for any given AM operation. Since the capability system is extensible, the null-Kernel can recognize new operations (and indeed complete AMs) at runtime.

null-Kernel Structure Figure 1 shows how OS software in the null-Kernel model is structured. The hardware presents a programming interface, which we term the Hardware-AM[1]. The other AMs in the figure export different sets of operations that ultimately make use of the Hardware-AM. AMs can be layered, e.g., AM-2 is partially built using AM-1's operations. In a null-Kernel, callers, with proper capabilities, may invoke operations exported by a "high-level" AM such as AM-2, or by "low-level" AMs such as AM-0, or any combination simultaneously. This is the key insight behind the null-Kernel: as long as a caller has proper capabilities, they may invoke operations at any level of abstraction, and thus the OS architecture is not confined to one model. More importantly, if the underlying resources are disjoint, or if the AMs cooperate (as described next), these calls compose, and can safely be executed in parallel or in any combination.

AMs structured with the null-Kernel capabilities permit many patterns of resource access and optimizations that are either cumbersome or impossible otherwise. These include "bypassing" layers by delegating capabilities and controlled sharing of resources between "peer" AMs. Next we describe the capability subsystem in more detail followed by examples demonstrating these access and optimization patterns.

2.1 null-Kernel Capabilities

In this section, we describe the null-Kernel capability system in more detail. AMs, including the Hardware-AM, define "objects" and "access rights" on objects. The null-Kernel capability system is extensible in that it operates over (dynamically defined) AM objects and rights. Capabilities are unforgeable references to a pair consisting of an AM object and a set of access rights on that object. Operations defined by AMs refer to one or more pairs of objects and access rights. For example, the Hardware-AM may define a memory page as an object, and `read` and `write` as access rights. A DMA operation that copies data onto a page would require the `write` access right on that page object. This requirement is reflected to the null-Kernel as described below; a caller may invoke an operation (DMA-write) only if they have the capabilities associated with the operation (in our example, the caller must have a capability that grants the `write` right to that memory page).

The null-Kernel capability system derives directly from prior work in capabilities [10, 16]. Like existing systems, in the null-Kernel, capabilities can be associated with object, rights pairs, delegated to others, derived to produce weaker capabilities (by reducing the rights set), and revoked. In the null-Kernel, when a capability is revoked, all derived capabilities are also revoked. Much like other capability systems, the basic security of the null-Kernel requires that a principal (a caller or AM) can only get access to a capability by either being granted the capability explicitly or deriving it from a stronger capability. In particular, colluders cannot grow their collective set of capabilities beyond what is explicitly granted to them.

Extensibility The novelty of null-Kernel capabilities derives from the observation that the operations capabilities guard are extensible by AMs. Specifically, AMs can define new objects at their level of abstraction (e.g., the Hardware-AM can define memory pages as objects, whereas a higher-level VM-AM that provides virtual memory can define address-spaces and memory regions as objects). AMs also define custom rights on objects, and

[1] Obviously, the Hardware-AM is not an "abstract" machine but we (ab)use the term for uniformity.

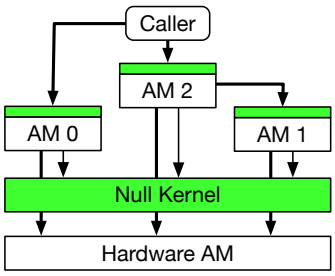

Figure 1: An overview of the null-Kernel showing system components: the null-Kernel, abstract machines, and callers.

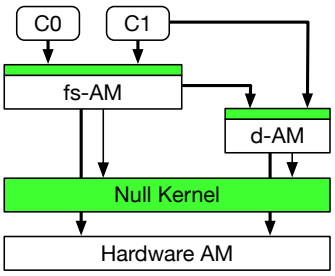

Figure 2: A representation of a file system AM built on top of and exposing capabilities for a disk AM.

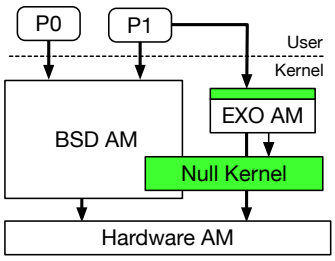

Figure 3: Architecture for retrofitting the null-Kernel into a BSD system to expose include safe exokernel like AM.

this set too is extensible at runtime. Again, as an example, both the Hardware-AM and the VM-AM can define `read` and `write` as rights on their respective objects (physical pages for the Hardware-AM, memory regions and address spaces for the VM-AM). The operations supported by the low-level Hardware-AM mirror those of the access rights (the read/write operation succeeds only if the caller has `read/write` access to a memory page). The VM-AM can associate much richer semantics with operations: for example, it may define a **mapReadable** function that takes an address space and a memory region as input, and the caller may only map a memory region into an address space if they have capabilities that provide `write` on the address space and `read` on the memory region. The null-Kernel provides an API that allows AMs to express capability requirements for each call. As long as the capabilities are delegated correctly any caller at any 'layer' of the system may use operations exported by an AM. The invoked AM maintains its correctness as long as the capabilities are checked prior to the operation being executed.

Capability hierarchies and delegation The null-Kernel naturally allows AMs to build and, in turn, export interfaces based on capabilities received from lower layers. These exported interfaces (and their associated capabilities) implicitly form a capability hierarchy. Hierarchical capabilities are different from simple delegation in which an AM directly grants received capabilities to others. (Delegation is useful for the layer bypassing model we discuss later.) For both hierarchical and delegated capabilities, AMs should follow two basic principles to ensure correctness for higher-layers:

- Logical separation: An AM should give potentially conflicting capabilities (e.g., write capabilities to the same object) to mutually trusting principals only (principals who understand and respect each other's invariants).
- Essential capability hiding: A higher-level AM should not give out any capability it has on a lower-level AM, if the capability can be used to violate the higher-level AM's own invariants.

If an AM violates one of these principles, the invariants of AMs or callers who receive these capabilities may be violated. For example, an AM should provide capabilities with `write` access to the same memory page to two callers only if these callers wish to implement a write-shared page with each other. Importantly, even if an AM violates these principles, only the callers or dependent AMs are affected; the correctness of other AMs or the rest of the system is not.

Capability Checks It is crucial to note that the null-Kernel, as described, is a schematic for how OSs should be structured. This schematic does not specify *how* capability checks are implemented, only that operations across AMs should be guarded by checks. This lack of specificity of implementation is on purpose since it provides unconstrained latitude in how (and when) the checks are implemented. For example, capability checks could be implemented in hardware (using the ISA [4, 20], MMU, processor protection rings [6, 16]), with programming language techniques (a safe compiler only generates code for capabilities it is provided [3]), using virtualization (guest OSs implement AMs constrained using the hypervisor interface [13]), and so on. Similarly, capability unforgeability can also be implemented using different mechanisms: EROS [16] protects capabilities using protection rings, whereas Amoeba [18] uses random placement of capabilities in a sparse address space. Other systems [1] [19] protect capabilities with cryptography primitives. The null-Kernel could employ any or all of these methods.

2.2 null-Kernel Structures

We conclude this section with two examples of how null-Kernel capabilities can be used to create interesting optimizations and sharing structures between AMs.

Layer Bypassing Consider a system (Figure 2) that exposes both a high-level filesystem AM (fs-AM) that operates on the level of files and directories, as well as a low-level disk AM (d-AM) that operates at the level of blocks. The fs-AM is implemented on top of the d-AM using raw block read/writes exported by the d-AM.

Most callers may prefer to use the file system through the fs-AM. However, applications, such as high performance databases, that want low level control over how data is arranged, may use the d-AM directly. With a null-Kernel, both these cases can be supported simultaneously by exposing both AMs to callers, subject to constraints of hierarchical and delegated capabilities. In particular, following the principle of "logical separation", the d-AM should give the direct callers and the fs-AM capabilities to disjoint disk blocks to ensure that they do not overwrite the other's data. Indeed, in existing systems, raw disks or partitions are often provided exclusively to high performance applications for exactly this reason (and with exactly this constraint).

A more interesting use-case is that the fs-AM itself can delegate block capabilities it receives from the d-AM to its callers. This would enable applications to write to file-system managed data blocks directly without going through the fs-AM. The null-Kernel enables such *layer bypassing* since it allows any caller with the appropriate capabilities to call any AM (the d-AM in this case). In this case, the fs-AM must adhere to the principle of "essential capability hiding" by never delegating write capabilities that pertain to file system metadata blocks to guarantee file system integrity.

AM peering The null-Kernel also supports non-hierarchical, peering structures between AMs. We illustrate this using VM paging as an example. Consider the virtual memory AM (VM-AM). Upon memory pressure, the VM-AM writes pages to disk. To accomplish this, we assume that the VM-AM has been delegated write capabilities to a set of disk blocks by the disk AM (d-AM). The VM-AM uses these capabilities to write pages to disk as needed. To page these items back in, the VM-AM invokes an operation in the d-AM that requires a block capability with read access and a page capability with write access. The d-AM may then asynchronously write into the page from the block and notify the VM-AM when the operation has completed. This peer-to-peer interaction between cooperating AMs is natively supported by the null-Kernel.

3 null-Kernel in Practice

In the previous section, we outlined the basic structure of the null-Kernel and described use cases where the relevant subsystems were written to conform to our model. Many null-Kernel ideas, however, are applicable to current OSs as well; in this section, we describe how salient parts of the null-Kernel can be integrated into production kernels and the types of optimizations this can enable.

Figure 3 depicts a production OS, such as FreeBSD, extended to recognize the null-Kernel as we describe next. The system also includes a new AM, the EXO AM, which exports a low-level interface to hardware, similar to that provided by exokernels. FreeBSD is not structured as a null-Kernel AM and there are several options as to how an EXO AM could co-habit with FreeBSD. One option to give the EXO AM access to the hardware is to let it run in supervisor mode, alongside the BSD AM.

Callers in this hybrid system are BSD processes, augmented with capabilities which can be used to access the EXO AM. Processes (which run in processor ring-3) calling into the EXO AM must incur a processor ring switch, and hence the "user-kernel boundary" separates processes from the EXO AM as well.

In this structure, the BSD AM and the EXO AM cooperate, and must share the hardware capabilities without conflict. For instance, the BSD AM could choose to not use its hardware capabilities for certain devices. The EXO AM can safely export its minimal interface and be used as a base for higher level abstractions on these devices. With more cooperation, the EXO AM could also provide read-only access to hardware primitives that are used by the BSD AM (e.g., by exporting processor status and memory reference bits as BSD executes). Such hybrid access to high- and low-level interfaces enables new use patterns that are not possible with either interface in isolation.

Access to new hardware The EXO AM can provide low-level access to new types of hardware such as GPUs, FPGAs, or smart-NICs for which the BSD kernel does not have support. New devices added to the machine would add additional hardware AMs to the system. Hardware vendors or kernel developers would then write a thin abstraction of the hardware AM and expose it via the EXO AM. At this point, the new hardware could be directly used by processes (with proper capabilities).

New Abstractions The ability to layer AMs would give us the opportunity to build higher level AMs in terms of the EXO AM. These higher level AMs would offer different abstractions that might be suitable for the hardware, and each application that wanted to use the new hardware could choose the AM that best meets its needs.

New abstractions need not be limited to new hardware. For instance, in cooperation with the BSD AM, the EXO AM could expose hardware features such as page reference bits in page tables which are usually hidden by the

BSD AM. These tracking bits could be used by applications to augment the BSD VM subsystem and implement novel features such as efficient software transactional memory (TM) or fast garbage collection. Current software implementations of TM require compiler augmentation of every single memory access [9]. This overhead can be entirely avoided if page reference bit were made available through the EXO AM.

Simultaneous high- and low-level access The hybrid BSD/EXO AM system can be used to implement layer bypassing as discussed earlier. The BSD AM could provide capabilities for disk blocks to processes, which could then use the EXO AM to implement their own optimizations *within* the blocks allocated by the BSD filesystem.

High-level AM over different low-level AMs The hybrid system would allow different AM's functionality to enable new use cases. For example, suppose new hardware in the form of NVRAM storage devices is available, and the EXO AM exports a low-level block interface to these devices. A higher-layer AM could provide a memory-mapped file interface to the NVRAM storage, and process logic could use this facility to implement efficient crash recovery. Here the null-Kernel allows programmers to use a high-level, well-understood paradigm (memory-mapped files) to program their application logic, and integrate it with low-level access to new hardware to implement new functionality (efficient crash recovery).

AM composition The examples above assume that either AMs partition resources, or are able to expose safe "enough" interfaces such that composite services, that use operations from multiple AMs do not cause deadlock or fault the system in some other manner. A sufficient condition to ensure both safety and progress is for each exported AM call to run to completion upon invocation, and for the AM to maintain all of its safety and progress invariants (e.g., release held locks) prior to call return (including for calls that it services in parallel). The OS system call interface maintains such an invariant, but internal kernel interfaces, that assume specific locking sequences and at times undocumented pre-conditions, do *not*, thereby making kernel modifications fraught with danger. To support composability, AMs could simply implement the sufficient condition we have described. Articulation of more precise and efficient criteria is likely feasible and remains part of our future work.

4 Related Work

VINO [17] and SPIN [3] offer mechanisms to safely extend monolithic kernels. Both systems require extensions to be written against a restricted, internal interface that maintains kernel invariants. These systems can be thought of as a limited instantiation of the hybrid system presented earlier, but access to the internal interface cannot be shared in a structured manner. This limits how extensions (AMs) relate:for instance, layer bypassing is not possible in either.

Microkernels [12] seL4 [5] and Barrelfish [15] export kernel objects to user space as capabilities. Capability types exported by the system are static. As a result, layer bypassing via delegated capabilities is not supported.

Exokernels [6] provide a minimal, non-portable hardware-like interface. Exokernel abstractions allow for the allocation and revocation of hardware resources in a manner similar to capability allocation, but unlike capabilities, these resources cannot be shared or reduced except by proxying through the resource owner.

EROS [16], derived from KeyKOS [8], is a stateless kernel that maps hardware into a set of capabilities. Applications use the operations permitted by these capabilities to construct higher level abstractions. EROS is equivalent to a specific instantiation of the null-Kernel that only exports a low level AM. Other instantiations, such as our hybrid model, are incompatible with EROS. HiStar [21] also exposes a limited set of kernel objects to user space, limiting access to those objects by tracking information flow.

The Cal timesharing system [11], Cambridge CAP computer [14] and Fluke [7] all allow an interface's operations to be implemented and over-ridden in a nested manner that is similar to subclasses. This layering is constrained by capabilities. Unlike the null-Kernel, the interface for these interfaces is fixed.

Dune [2] uses hardware support for virtualization to grant applications safe access to virtual memory instructions without the use of explicit capabilities. This is akin to a form of layer bypassing supported by null-Kernel; however, Dune does not support the simultaneous use of both low and high level memory interfaces.

5 Conclusions

This paper describes the null-Kernel, a new structure for system software that enables abstractions for efficient access to new hardware and admits new optimizations for existing hardware. The null-Kernel posits an extensible capability mechanism that distributes system resources across software that provides programming interfaces at different layers of abstraction. Equipped with proper capabilities, callers, such as user processes, can simultaneously program to any or all of these abstractions as appropriate. We describe requirements of the null-Kernel's basic capability mechanism and show how the null-Kernel can be used to implement new abstractions and optimizations.

References

[1] ANDERSON, M., POSE, R., AND WALLACE, C. S. A password-capability system. *The Computer Journal 29*, 1 (1986), 1–8.

[2] BELAY, A., BITTAU, A., MASHTIZADEH, A., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe user-level access to privileged CPU features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 335–348.

[3] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C., AND EGGERS, S. Extensibility safety and performance in the SPIN operating system. *ACM SIGOPS Operating Systems Review 29*, 5 (1995), 267–283.

[4] CARTER, N. P., KECKLER, S. W., AND DALLY, W. J. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1994), ASPLOS VI, ACM, pp. 319–327.

[5] ELKADUWE, D., DERRIN, P., AND ELPHINSTONE, K. Kernel design for isolation and assurance of physical memory. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems* (2008), IIES '08, pp. 35–40.

[6] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 251–266.

[7] FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. Microkernels meet recursive virtual machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation* (New York, NY, USA, 1996), OSDI '96, ACM, pp. 137–151.

[8] HARDY, N. Keykos architecture. *SIGOPS Oper. Syst. Rev. 19*, 4 (Oct. 1985), 8–25.

[9] KESTOR, G., DALESSANDRO, L., CRISTAL, A., SCOTT, M. L., AND UNSAL, O. Interchangeable back ends for stm compilers. *a= a 5*, 6, 7.

[10] LACKORZYNSKI, A., AND WARG, A. Taming subsystems: Capabilities as universal resource access control in l4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (2009), IIES '09, pp. 25–30.

[11] LAMPSON, B. W., AND STURGIS, H. E. Reflections on an operating system design. *Communications of the ACM 19*, 5 (1976), 251–265.

[12] LIEDTKE, J. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 237–250.

[13] MADHAVAPEDDY, A., AND SCOTT, D. J. Unikernels: the rise of the virtual library operating system. *Communications of the ACM 57*, 1 (2014), 61–69.

[14] NEEDHAM, R. M., AND WALKER, R. D. The cambridge cap computer and its protection system. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1977), SOSP '77, ACM, pp. 1–10.

[15] SCHÜPBACH, A., PETER, S., BAUMANN, A., ROSCOE, T., BARHAM, P., HARRIS, T., AND ISAACS, R. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems* (2008), vol. 27.

[16] SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. Eros: A fast capability system. *SIGOPS Oper. Syst. Rev. 33*, 5 (Dec. 1999), 170–185.

[17] SMALL, C. A., AND SELTZER, M. I. Vino: An integrated platform for operating system and database research. details

[18] TANENBAUM, A. S., MULLENDER, S. J., AND RENESSE, R. V. Using sparse capabilities in a distributed operating system.

[19] VOCHTELOO, J., RUSSELL, S., AND HEISER, G. Capability-based protection in the mungi operating system. In *Object Orientation in Operating Systems, 1993., Proceedings of the Third International Workshop on* (1993), IEEE, pp. 108–115.

[20] WOODRUFF, J., WATSON, R. N., CHISNALL, D., MOORE, S. W., ANDERSON, J., DAVIS, B., LAURIE, B., NEUMANN, P. G., NORTON, R., AND ROE, M. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 457–468.

[21] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 263–278.