Labels and Event Processes in the Asbestos Operating System

STEVE VANDEBOGART, PETROS EFSTATHOPOULOS, and EDDIE KOHLER
University of California, Los Angeles
MAXWELL KROHN, CLIFF FREY, DAVID ZIEGLER, FRANS KAASHOEK, and ROBERT MORRIS
Massachusetts Institute of Technology
and
DAVID MAZIÈRES
Stanford University

Asbestos, a new operating system, provides novel labeling and isolation mechanisms that help contain the effects of exploitable software flaws. Applications can express a wide range of policies with Asbestos's kernel-enforced labels, including controls on interprocess communication and system-wide information flow. A new event process abstraction defines lightweight, isolated contexts within a single process, allowing one process to act on behalf of multiple users while preventing it from leaking any single user's data to others. A Web server demonstration application uses these primitives to isolate private user data. Since the untrusted workers that respond to client requests are constrained by labels, exploited workers cannot directly expose user data except as allowed by application policy. The server application requires 1.4 memory pages per user for up to 145,000 users and achieves connection rates similar to Apache, demonstrating that additional security can come at an acceptable cost.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls, Access controls*; D.4.1 [**Operating Systems**]: Process Management; D.4.7 [**Operating Systems**]: Organization and Design; C.5.5 [**Computer System Implementation**]: Servers

General Terms: Security, Design, Performance

Additional Key Words and Phrases: Information flow, labels, mandatory access control, process abstractions, secure Web servers

This work was supported by DARPA grants MDA972-03 and FA8750-04-1-0090, and by joint NSF Cybertrust/DARPA grant CNS-0430425. E. Kohler, D. Mazières, and R. Morris are supported by Sloan fellowships. E. Kohler is also supported by a Microsoft Research New Faculty Fellowship.
Authors' address: http://asbestos.cs.ucla.edu (Web site).

ACM Reference Format:
VanDeBogart, S., Efstathopoulos, P., Kohler, E., Krohn, M., Frey, C., Ziegler, D., Kaashoek, F., Morris, R., and Mazières, D. 2007. Labels and event processes in the Asbestos operating system. ACM Trans. Comput. Syst. 25, 4, Article 11 (December 2007), 43 pages. DOI = 10.1145/1314299.1314302 http://doi.acm.org/10.1145/1314299.1314302

1. INTRODUCTION

Breaches of Web servers and other networked systems routinely divulge private information on a massive scale [Lemos 2005; News10 2005; Trounson 2006]. The kinds of software flaws that enable these breaches will persist, but systems can be designed to limit exploits' possible impact. An effective way to contain exploits is application-level data isolation, a policy that prevents a server acting for one principal from accessing data belonging to another principal—an instance of the principle of least privilege [Saltzer and Schroeder 1975]. Such a policy, enforced by the operating system at the behest of a small, trusted part of the application, would stop whole classes of exploits, including SQL injection and buffer overruns, making servers much safer in practice.

Unfortunately, current operating systems cannot enforce data isolation. Even the weaker goal of isolating Web *services* from one another requires fiddly and error-prone abuse of primitives designed for other purposes [Krohn 2004]. Most servers thus retain the inherently insecure design of monolithic code with many privileges, including the privilege to access any and all application data. As a result, high-impact breaches continue to occur.

New operating system primitives are needed [Krohn et al. 2005], and the best place to explore candidates is in the unconstrained context of a new OS. This article presents Asbestos, a new operating system that can enforce strict application-defined security policies, and the Asbestos Web server, an efficient, unprivileged server that isolates different users' data.

Asbestos's contributions are twofold. First, all access control checks use *Asbestos labels*, a primitive that combines advantages of discretionary and nondiscretionary access control. Labels determine which services a process can invoke and with which other processes it can interact. Like traditional discretionary capabilities, labels can be used to enumerate positive rights, such as the right to send to the network. Unlike traditional capability systems, Asbestos labels can also track and limit the flow of information. As a result, Asbestos supports capability-like and traditional multilevel security (MLS) policies [Department of Defense 1985], as well as application-specific isolation policies, through a single unified mechanism.

Second, Asbestos's *event process* abstraction lets server applications efficiently support and isolate many concurrent users. In conventional label systems, server processes would quickly become contaminated by multiple users' data and lose the ability to respond to any single user. One possible fix is a forked server model, in which each active user has her own forked copy of the server process; unfortunately, this resource-heavy architecture burdens the OS with many thousands of processes that need memory and CPU time. Event processes let a single process keep private state for multiple users, but isolate that state so that an exploit exposes only one user's data. The event process discipline encourages efficient server construction, and in our experiments, servers can cache tens of thousands of user sessions with low storage costs.

Asbestos labels and event processes let us build data isolation into a challenging application, the Asbestos Web server. This dynamic Web server isolates the data of each application user so that exploited Web application code cannot access the secret data of other users. Measurements on an x86 PC show that an Asbestos Web server can support comprehensive user isolation at a cost of about 1.4 memory pages per user for more than a hundred thousand users. Despite processes whose labels contain hundreds of thousands of elements, the server is competitive with Apache on Unix. Asbestos shows that an OS can support flexible, yet stringent security policies, including information flow control, even within the challenging environment of a high-performance Web server.

The rest of this article is organized as follows. First we examine related work (Section 2) and then explain our technical goals and their consequences for the Asbestos design (Section 3). In Section 4, we give a brief overview of Asbestos before detailing Asbestos labels (Section 5), label persistence (Section 6), and the event process model (Section 7). Section 8 presents the Asbestos Web server and discusses how it uses Asbestos features to define a data isolation policy. The article finishes with a discussion of covert channels (Section 9) and an evaluation of Asbestos's performance in the context of our example application (Section 10).

2. RELATED WORK

Mandatory access control (MAC) systems enforce end-to-end security policies by transitively following causal links between processes. Operating systems have long expressed and enforced these policies using *labels* [Department of Defense 1985]. Labels assign each subject and object a security level, which traditionally consists of a hierarchical sensitivity classification (such as *unclassified*, *secret*, *top-secret*) in each of a set of categories (*nuclear*, *crypto*, and so forth). To observe an object, a subject's security level must dominate the object's. For example, a file with secret, nuclear data should only be readable by processes whose clearance is at least secret and whose category set includes nuclear. Security enhancement packages supporting labels are available today for many popular operating systems, including Linux [Loscocco and Smalley 2001] and FreeBSD [Watson et al. 2003].

MAC systems generally aspire to achieve some variant of the $*$-*property* [Bell and La Padula 1976]: whenever a process P can observe object O_1 and modify object O_2, O_2's security level must dominate O_1's. In the absence of the $*$-property, P could leak O_1's contents by writing it to O_2, leaving O_1's confidentiality at P's discretion. Of course, real operating systems implement some way to declassify or "downgrade" data—for example, as a special privilege afforded certain users if they press the secure attention key [Karger et al. 1990]—but this lies outside the main security model.

Most MAC systems are geared towards military specifications, which require labels to specify at least 16 hierarchical sensitivity classifications and

64 nonhierarchical categories [Department of Defense 1985]. This label format severely limits what kinds of policies can be expressed. The fixed number of classifications and categories must be centrally allocated and assigned by a security administrator, preventing applications from crafting their own policies with labels alone. Thus, MAC systems typically combine labels with a separate discretionary access control mechanism. Ordinary Unix-style users and groups might enforce access control within the secret, nuclear level.

More recent MAC operating systems, such as SELinux and TrustedBSD, generally require administrator privilege to change the active security policy. The SELinux Policy Server [MacMillan et al. 2006] has tried to correct this shortcoming by adding a meta-policy, which specifies how different subjects can modify the policy. However, the security administrator must still anticipate and approve of the policy structure of every individual application. These restrictions prevent applications from using MAC primitives as security tools without the cooperation and approval of the security administrator.

Unlike previous systems, Asbestos lets any process dynamically create nonhierarchical security categories, which we call *tags*. An application can construct an arbitrary security policy involving tags it creates, but remains constrained by external policies involving other tags. This makes Asbestos labels an effective tool for application and administrator use. Asbestos also brings *privilege* into the security model. A process with privilege for a tag can bypass the $*$-property with respect to that tag either by declassifying information or by raising the security clearance of other processes. As described later, the Asbestos system call interface has other novel features that facilitate label use, including discretionary labels that apply to single messages.

The idea of dynamically adjusting labels to track potential information flow dates back to the High-Water-Mark security model [Landwehr 1981] of the ADEPT-50 in the late 1960s. Numerous systems have incorporated such mechanisms, including IX [McIlroy and Reeds 1992] and LOMAC [Fraser 2000]. The ORAC model [McCollum et al. 1990] supported the idea of individual originators placing accumulating restrictions on data, somewhat like creating tags, except that data could still only be declassified by users with the privileged Downgrader role.

Asbestos labels more closely resemble language-level flow control mechanisms. Jif [Myers and Liskov 2000], which supports decentralized privilege by allowing different code modules to “own” different components of a label, was a particular inspiration. Asbestos takes a label model as flexible as Jif’s and applies it *among* processes, rather than within them. Labels in Jif have distinct components for confidentiality and integrity. A confidentiality policy is built from atoms such as “principal a allows principal b to read this object,” where principals are arranged in a hierarchy. Asbestos labels achieve similar goals with a unified namespace for both confidentiality and integrity, and with principals that are not hierarchically related and that can be created at any time. As a programming language extension, Jif can perform most of its label checks statically, at compile time. This avoids affecting control flow on failed checks, a source of implicit information flows [Denning and Denning 1977]. Asbestos’s current design avoids some implicit information flows, and by adopting ideas from successor operating systems—in particular, by requiring processes to actively change their own labels [Zeldovich et al. 2006] rather than updating labels implicitly during message communication—Asbestos could avoid all implicit flows related to label checks.

Asbestos uses communication ports similar to those of previous message-passing operating systems [Rashid and Robertson 1981; Tanenbaum et al. 1990; Rozier et al. 1988; Liedtke 1995; Mitchell et al. 1994; Cheriton 1988], some of which can confine executable content [Jaeger et al. 1999], others of which have had full-fledged mandatory access control implementations [Branstad et al. 1989]. Asbestos ports are a specialized type of tag, and can appear in labels. The combination of ports and tags allow labels to emulate security mechanisms from discretionary capabilities to multilevel security.

In theory, capabilities alone suffice to implement mandatory access control. For instance, KeyKOS [Key Logic 1989] achieved military-grade security by isolating processes into compartments. EROS [Shapiro et al. 1999] later successfully realized the principles behind KeyKOS on modern hardware. Implementing mandatory access control on a pure capability system, such as KeyKOS, requires the deployment of reference monitors at compartment boundaries to prevent inappropriate capabilities from escaping. A number of designs have therefore combined capabilities with authority checks [Berstis 1980], interposition [Karger 1987], or even labels [Karger and Herbert 1984]. Asbestos can implement capability-like policies within its label mechanism for those cases where capability policies are the best fit.

Mandatory access control can also be achieved with unmodified traditional operating systems through virtual machines [Goldberg 1973; Karger et al. 1990]. For example, the NetTop project [VMware 2000] uses VMware for multilevel security. Virtual machines have two principal limitations, however: performance [King and Chen 2003; Whitaker et al. 2002] and coarse granularity. One of the goals of Asbestos is to allow fine-grained information flow control so that a single process can handle differently labeled data. To implement a similar structure with virtual machines would require a separate instance of the operating system for each label type.

3. GOAL

In a nutshell, Asbestos aims to achieve the following goal:

> *Asbestos should support efficient, unprivileged, and large-scale server applications whose application-defined users are isolated from one another by the operating system, according to application policy.*

This is difficult to achieve on any other operating system. We evaluated Asbestos by implementing a secure application that requires all components of the goal, namely a dynamic-content Web server that isolates user data. The rest of this section expands on and clarifies the goal. We concentrate on server applications as they are in many ways the most challenging applications that operating systems must handle. Nevertheless, Asbestos mechanisms should aid in the construction of other types of software; for example, email readers could use policies to restrict the privileges of attachments, reducing the damage inflicted by users who unwittingly run disguised malicious code.

A *large-scale server application* responds to network requests from a dynamically changing population of thousands or even hundreds of thousands of users. A single piece of hardware may run multiple separate or cooperating applications. Examples include Web commerce and bulletin-board systems, as well as many pre-Web client/server systems. Such applications achieve good performance through aggressive caching, which minimizes stable storage delays. By *efficient*, then, we mean that an Asbestos server should cache user data with low overhead. This would be simple if the cache were trusted, but we want to *isolate* different users' data so that security breaches are contained. The Asbestos event process mechanism aims to satisfy this requirement.

Unprivileged means that installing and running secure software should not require system privilege. For instance, the system administrator's cooperation should not be necessary to install an application, and application maintainers should be able to modify the application security policy without administrator approval.

Users are *application-defined*, meaning each application can define its own notion of principal and its own set of principals. One application's users may be distinct from another's, or the user populations may overlap. An application's users may or may not correspond to human beings and typically won't correspond to the set of human beings allowed to log in to the system's console.

By *isolated*, we mean that *a process acting for one user cannot gain inappropriate access to other users' data*. Appropriate access is defined by an *application policy*: the application defines which of its parts should be isolated and how. The policy should also support flexible cross user *sharing* for data that need not be isolated. Of course all users must trust some parts of the application, such as the part that assigns users to client connections. Since bugs in this trusted code could allow arbitrary inter-user exploits, we aim to minimize its size.

The application defines the isolation policy, but the *operating system* enforces it. The OS should prevent processes from violating this policy whether or not they are compromised. For example, processes should not be able to launder data through other services and applications. As a result, isolation policies must restrict *information flow* among processes that may be ignorant of the policies. Unfortunately, systems that control information flow through run-time checks can inappropriately divulge information when those checks fail [Myers and Liskov 2000]; in effect, kernel data structures for tracking information flow provide a covert storage channel. This version of Asbestos aims to eliminate storage channels that can be exploited without multiple processes and limit channels that do. Related operating systems further reduce covert channels [Zeldovich et al. 2006], although some channels, such as timing channels, will likely never be eliminated in any but the simplest information flow systems and languages. Nevertheless, preventing *overt* leaks, as Asbestos already does, would block the breaches seen in the wild, and Asbestos labels can already help prevent high-value secrets from reaching untrusted code through *any* channels, overt or covert.

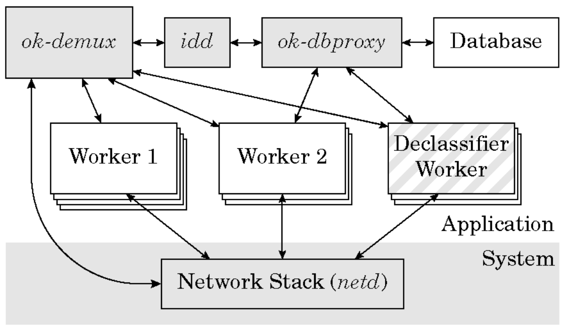

Fig. 1. Processes of the Asbestos Web server. Gray boxes are trusted in the context of this application (only the network stack and kernel are trusted system-wide). Worker processes contain one event process per user session. Striped boxes are semi-trusted; they hold privilege with respect to a single user at a time.

In summary, Asbestos must support a variant of the ∗-*property*, which transitively isolates processes by tracking and limiting the flow of information. Unprivileged applications define their own isolation policies and decide what information requires isolation. Furthermore, OS mechanisms for labeling processes must support highly concurrent server applications.

We show that Asbestos achieves this goal through the design and implementation of the Asbestos Web server, an improved version of the original OKWS for Unix [Krohn 2004]. The server implements a Web site with multiple dynamic *workers*. Separate workers might support logging in, retrieving data, and changing a password, for example. The *ok-demux* process analyzes incoming connection requests and forwards them to the relevant worker. Each worker is its own process and caches relevant user data. Caches for different users are isolated from one another using labels and event processes. A production system would additionally have a cache shared by all workers, and Asbestos could without much trouble support a shared cache that isolated users. We also implemented *declassifier* workers that can export user data to the public. Workers are untrusted, meaning that a worker compromise cannot violate the user isolation policy. Trusted components for this application include the *ok-demux* process, the *ok-dbproxy* database interface, and an *idd* process that checks user passwords, as well as system components such as the network interface, IP stack, file system, and kernel. However, the server's trusted components are not trusted by any other applications on the system: no part of the application runs with root privilege (a concept that on Asbestos does not exist). Declassifier workers are semi-trusted within the application, in that a compromised declassifier can inappropriately leak the compromised user's data but cannot gain access to uncompromised users' data. Figure 1 shows this server's process architecture.

4. ASBESTOS OVERVIEW

The Asbestos operating system design features a small, nonpreemptive kernel and single-threaded processes. Asbestos does not currently support symmetric multiprocessors or shared memory. Processes communicate with one another using asynchronous message passing, somewhat as in microkernels such as Mach; messages are queued in the kernel until they are received. The kernel supports system calls for allocating, remapping, and freeing memory at particular virtual addresses, for creating and destroying processes, for sending and receiving messages, for bootstrapping, and for debugging, in addition to calls supporting tag, label, and event process functionality.

Each message is addressed to a single *port*. A process can create arbitrarily many ports. Messages sent to a port are delivered to the single process with *receive rights* for that port; this is initially the process that created the port, but receive rights are transferable. The right to *send* to a port, however, is determined through label checks, as described later.

Asbestos messaging is, unusually, *unreliable*: the **send** system call might return a success value even if the message cannot be delivered. There are several reasons for this. For one, the kernel cannot tell whether a message is deliverable until the instant that the receiving process tries to receive it, since in the meantime process labels can change to prevent or allow delivery. For another, reliable delivery notification would let a process leak information using careful label changes, for example causing successful delivery to correspond to 1 bits and unsuccessful delivery to 0 bits. However, since only label checks and resource exhaustion will cause dropped messages, careful label management—such as our Web server's—can make delivery reliable in practice.

Conventional mechanisms such as pipes and file descriptors are emulated using messages sent to ports. To read a file, for example, the client sends a READ message to the file server's port and awaits the corresponding READ_R reply. The protocol messages were inspired by Plan 9's 9P [Pike et al. 1995].

When asked to create a port, the kernel returns a new port with an unpredictable name. This is necessary because the ability to create a port with a specific name would be a covert channel. Communication is bootstrapped using environment variables that specify port names for well-known services.

5. ASBESTOS LABELS

The heart of an information flow control system is its definition of the fundamental labeling primitive. Labels with limited flexibility, such as the Perl programming language's one-bit "taint tracking" or traditional MAC systems with fixed category sets, aren't well suited for constructing complex, application-specific policies. Asbestos labels were designed to support complex policies with a unified and self-contained mechanism. Asbestos labels combine the following properties:

Practically Unlimited Information Flow Categories. Asbestos tracks information flow in 2^{61} independent categories called *tags*. Such a large space can support a practically unlimited number of application-specific security policies, so any process may allocate arbitrarily many tags. A given security policy may require one, two, or many tags to implement.

Secrecy and Integrity in a Single Label. A label specifies a sensitivity level for each tag. Normal sensitivity levels range from **0** to **3**. Information can flow freely from **0** up to **3**; the reverse direction represents declassification and

Level	Usage
⋆	Privilege for the tag
0	High integrity for the tag
1	Default tracking level—no restriction for the tag
2	Default clearance level, used in taint tracking
3	High secrecy for the tag

Fig. 2. Asbestos label sensitivity levels and common usage.

requires the intervention of a privileged process. Sensitivity levels let a single label combine the functions of secrecy tracking and integrity protection. Most labels, such as those for processes and files, start with an intermediate level **1** for each tag. The lower level **0** is used in policies such as integrity tracking ("this object was modified only by high-integrity processes"), and the higher levels **2** and **3** are used in policies such as taint tracking and protection of secret information. Figure 2 summarizes the common use for each level.

Decentralized Declassification, Decentralized Privilege, and Unprivileged Security Policy Management. These concepts, which are the core of any decentralized information flow control system, all amount to allowing a privileged process to selectively *reduce* a label's sensitivity for those tags for which it has privilege. For instance, raising a label's level for some tag from **1** to **2** requires no privilege, but reducing that level from **2** to **1** would require the intervention of a process with the corresponding privilege. In Asbestos, privilege is represented as the special sensitivity level ⋆. A process with level ⋆ for some tag can bypass the normal information flow control rules for that tag.

No Explicit Principals and a Single Tag Namespace. The Jif system includes a principal hierarchy distinct from labels, and Jif labels represent constraints among principals. Asbestos labels are self-contained: allocating a tag confers privilege for that tag on the allocating process. Tags may correspond to principals, and our Asbestos Web server uses some tags in this way, but per-principal tags are just one of several usage patterns. Other system objects, such as processes and IPC ports, also use the tag namespace, allowing processes to manage their use with a single label mechanism.

To track information flow, the Asbestos operating system applies labels to processes. The per-process mechanisms that use labels are:

Tracking and Clearance Labels. Each process has two labels, a *tracking label* and a *clearance label*. The tracking label records the information flow a process has observed so far; the clearance label bounds the maximum tracking label a process is allowed to achieve. Thus, tracking and clearance labels behave like IX's current and maximum labels [McIlroy and Reeds 1992]. The initial level for each tag in a clearance label is **2**. This allows communication by default, since the default tracking level, **1**, is less. Level **3**, which is used for high-secrecy information, is higher than the default clearance, so processes cannot learn secrets unless explicitly granted the necessary clearance.

Port Clearance. Each IPC port has a clearance label that further restricts the messages delivered to that port. Port clearance labels let processes limit their possible contamination and support security policies resembling capabilities, where a process cannot send a message to another process's protected port until it is explicitly granted that right.

Discretionary Labels. Where conventional information flow systems simply track information flow, Asbestos applications can use labels as an active and discretionary tool. Four optional *discretionary labels* may be specified when sending a message. These labels let a process (1) send with a higher label than its true label (for example, when a privileged process sends a message that it wishes to mark secret); grant privilege by (2) reducing the receiver's tracking label or (3) raising its clearance label; and (4) pass to the receiver a kernel-verified upper bound of the sender's tracking label, letting a process demonstrate its tracking label state while avoiding ambient authority.

The remainder of this section describes Asbestos labels in more detail. We present notation, IPC rules, and an array of examples.

5.1 Label Notation

An Asbestos label defines a level for each of 2^{61} possible tags. Conceptually, a label is a function from tags to levels. In practice, each label maps most tags to a single level called that label's *default level*; for instance, in tracking labels, most tags map to level **1**. We write a label using set-like notation: a comma-separated list of tag/level pairs followed by the default level. For example, $L = \{v\,\mathbf{0}, w\,\mathbf{3}, \mathbf{1}\}$ assigns level **0** to tag v, level **3** to tag w, and level **1** to all other tags:

$$L(t) = \begin{cases} \mathbf{0} & \text{if } t = v, \\ \mathbf{3} & \text{if } t = w, \\ \mathbf{1} & \text{otherwise.} \end{cases}$$

A partial order on labels determines whether one label dominates another. In this partial order, L_A is less than or equal to L_B if for each tag, the level in L_A is less than or equal to the level for the same tag in L_B:

$$L_A \sqsubseteq L_B \quad \text{iff} \quad \forall t : L_A(t) \leq L_B(t).$$

If one tag's level is less in L_A than in L_B and another tag's level is greater in L_A than in L_B, then the labels are incomparable: $L_A \not\sqsubseteq L_B$ and $L_B \not\sqsubseteq L_A$. The lowest possible label, $\bot = \{\star\}$, is less than or equal to every Asbestos label in the partial order; the highest possible label, $\top = \{\mathbf{3}\}$, is greater than or equal to every label.

The least upper bound of two labels, written $L_A \sqcup L_B$, is the smallest label with both $L_A \sqsubseteq L_A \sqcup L_B$ and $L_B \sqsubseteq L_A \sqcup L_B$. (In the context of classical information flow, this is the lowest label that combines the information flow constraints of both L_A and L_B.) The least upper bound operator works by taking for each tag the highest corresponding level in either label. For example,

$$\{v\,\mathbf{0}, w\,\mathbf{0}, x\,\mathbf{3}, \mathbf{1}\} \sqcup \{v\,\mathbf{0}, \mathbf{1}\} = \{v\,\mathbf{0}, x\,\mathbf{3}, \mathbf{1}\}.$$

This is clearer when we explicitly write out all tags mentioned in either label:

$$\{v\,\mathbf{0}, w\,\mathbf{0}, x\,\mathbf{3}, \mathbf{1}\} \sqcup \{v\,\mathbf{0}, w\,\mathbf{1}, x\,\mathbf{1}, \mathbf{1}\} = \{v\,\mathbf{0}, w\,\mathbf{1}, x\,\mathbf{3}, \mathbf{1}\}.$$

System labels (maintained by the kernel)

$\mathbf{T}_P$	Process P's tracking label.
$\mathbf{C}_P$	Process P's clearance label. The system maintains the invariant that $\mathbf{T}_P \sqsubseteq \mathbf{C}_P$.
$\mathbf{PC}_p$	Port p's port clearance label. May be set arbitrarily by p's owning process.

Discretionary labels (set by the process sending a message)

$\mathbf{T}^+$	Increases the message's effective tracking label. Used when a message contains data at a higher sensitivity level than the sending process itself. Commonly used by privileged processes; defaults to $\bot = \{\star\}$, which implies no increase.
$\mathbf{T}^-$	Decreases the receiving process's tracking label. This represents declassification and/or granting privilege. Defaults to $\top = \{\mathbf{3}\}$, which implies no decrease; setting it to any other value requires privilege for the affected tags.
$\mathbf{C}^+$	Increases the receiving process's clearance label, granting it clearance to further raise its tracking label. Defaults to $\bot = \{\star\}$, which implies no increase; setting it to any other value requires privilege for the affected tags.
$\mathbf{V}$	Declares an upper bound on the sender's tracking label. This might represent the privilege the sender intends to use for this operation. The kernel verifies that $\mathbf{T}_P \sqsubseteq \mathbf{V}$, then passes $\mathbf{V}$ to the receiver along with the message. Defaults to $\top = \{\mathbf{3}\}$, which conveys no information.

Fig. 3. Notation and description for Asbestos system and discretionary labels.

In notation,

$$(L_A \sqcup L_B)(t) = \begin{cases} L_A(t) & \text{if } L_A(t) \geq L_B(t), \\ L_B(t) & \text{otherwise.} \end{cases}$$

Similarly, the greatest lower bound operator $\sqcap$ is defined as

$$(L_A \sqcap L_B)(t) = \begin{cases} L_A(t) & \text{if } L_A(t) \leq L_B(t), \\ L_B(t) & \text{otherwise.} \end{cases}$$

Thus, Asbestos labels form a lattice [Denning 1976].

Sensitivity levels are related as $\star < \mathbf{0} < \mathbf{1} < \mathbf{2} < \mathbf{3}$, which represents privilege as a sort of "super-high integrity." However, unlike true integrity, receiving a message from an unprivileged process should not eliminate privilege. We explicitly represent the preservation of privilege by adding it back after a message is received. This makes use of a *privilege preservation operator*, written $L_A \sqcap L_B^\star$, where

$$(L_A \sqcap L_B^\star)(t) = \begin{cases} \star & \text{if } L_B(t) = \star, \\ L_A(t) & \text{otherwise.} \end{cases}$$

5.2 System Labels and IPC

We now describe the information flow rules that control process intercommunication. In the rules, process P's tracking and clearance labels are denoted $\mathbf{T}_P$ and $\mathbf{C}_P$, respectively. Port p's port clearance label is written $\mathbf{PC}_p$. The four discretionary labels specified by a process when sending a message are $\mathbf{T}^+$, $\mathbf{T}^-$, $\mathbf{C}^+$, and $\mathbf{V}$; their functions are summarized in Figure 3 and described in more depth in the following.

The core rule for process communication is that process Q can receive a message from process P only if Q is cleared to see any information that P may have seen—or, in terms of labels,

$$\mathbf{T}_P \sqsubseteq \mathbf{C}_Q. \tag{1}$$

On receiving the message, the kernel must update Q's tracking label to account for the message. The message is conservatively assumed to carry all of the information that P has seen, so the message carries P's tracking label, and the kernel sets

$$\mathbf{T}_Q \leftarrow \mathbf{T}_P \sqcup \mathbf{T}_Q. \tag{2}$$

These rules are the well-known basis of any classical information flow system.

Asbestos privilege, port clearance, and discretionary labels expand the equations, but introduce necessary flexibility. We list these changes and describe their effects on the information flow rules, then present the complete rules.

Preserving Privilege. The receiver's privilege is preserved after it receives a message. This changes the tracking label update to $\mathbf{T}_Q \leftarrow (\mathbf{T}_P \sqcup \mathbf{T}_Q) \sqcap \mathbf{T}_Q^\star$.

Increasing the Tracking Label. The sending process can raise a message's label beyond $\mathbf{T}_P$, the process's tracking label, by supplying the $\mathbf{T}^+$ discretionary label at send time. The kernel treats the message's tracking label as $\mathbf{T}_P \sqcup \mathbf{T}^+$. The default $\mathbf{T}^+$ label is $\perp$, which does nothing since $L \sqcup \perp = L$ for any L. Increasing the tracking label is particularly important for trusted server processes that handle many classes of differently labeled information, such as file servers and databases. Although such processes will hold privilege for many tags, each message they send should carry a label appropriate to the message data. $\mathbf{T}^+$ lets them increase the message label to an appropriate value.

Declassification and Granting Privilege. The $\mathbf{T}^-$ discretionary label lets the sending process grant some of its privilege to the receiver, or use its privilege to declassify the receiver. In particular, $\mathbf{T}^-$ lowers Q's tracking label when the message is received. For instance, to declassify Q with respect to a tag t, P might set $\mathbf{T}^- = \{t\,\mathbf{1}, \mathbf{3}\}$; this will reduce $\mathbf{T}_Q(t)$ to the default of $\mathbf{1}$ even if Q had previously observed high-secrecy t data. To grant Q privilege with respect to t, P might set $\mathbf{T}^- = \{t\,\star, \mathbf{3}\}$. Since these operations change tracking labels contrary to the classical information flow rules, they require privilege to exercise. Specifically, P must have privilege for t in order to declassify with respect to t: if $\mathbf{T}^-(t) \neq \mathbf{3}$, then $\mathbf{T}_P(t)$ must equal $\star$. Attempting to declassify without the necessary privilege is an error and causes the send operation to fail. This failure is safely reported to the sending process: when a failure involves only the sender's own labels, reporting an error does not inappropriately expose information.

Granting Clearance. The $\mathbf{C}^+$ discretionary label grants a different kind of privilege, namely the right to observe secret data. $\mathbf{C}^+$ raises Q's clearance label as the message is received, allowing a corresponding increase of Q's tracking label as this or a later message is received. For instance, to let Q observe high-secrecy t data, P might set $\mathbf{C}^+ = \{t\,\mathbf{3}, \star\}$; this will raise $\mathbf{C}_Q(t)$ to $\mathbf{3}$. Again, since this operation acts contrary to the classical information flow rules, it requires privilege to exercise. Specifically, when P sends a message with $\mathbf{C}^+(t) \neq \star$, it must have privilege for t. Only a process can reduce its own clearance label. A process may simultaneously grant clearance and increase the receiver's tracking label for a tag t by setting $\mathbf{T}^+ = \mathbf{C}^+ = \{t\,\mathbf{3}, \star\}$.

Port Clearance. Port clearance labels let a process declare different information flow characteristics for each of its communication ports. In particular, a process can restrict which processes can send to a port. Each port q has a clearance label $\mathbf{PC}_q$ that modifies the clearance check. Q can receive a message sent by P to port q only if the process clearance $\mathbf{C}_Q$ and the port clearance $\mathbf{PC}_q$ *both* allow it:

$$\mathbf{T}_P \sqcup \mathbf{T}^+ \sqsubseteq \mathbf{C}_Q \text{ and } \mathbf{T}_P \sqcup \mathbf{T}^+ \sqsubseteq \mathbf{PC}_q$$

or, equivalently,

$$\mathbf{T}_P \sqcup \mathbf{T}^+ \sqsubseteq \mathbf{C}_Q \sqcap \mathbf{PC}_q.$$

As an important example, port clearance labels can provide capability-like semantics. If port p's clearance label is $\mathbf{PC}_p = \{p\,\star, \mathbf{3}\}$, then only processes with privilege for p can send a message to p. Such processes can use $\mathbf{T}^-$ to confer that privilege on others. Thus, when $\mathbf{PC}_p = \{p\,\star, \mathbf{3}\}$, having privilege for p resembles owning a capability that allows sending messages to port p. Since a port clearance label only applies to messages sent to that port, a process can distribute multiple independent "send capabilities," one per port.

Port clearance labels also restrict how far a process's clearance may be raised. In particular, a message sent to port q cannot use $\mathbf{C}^+$ to raise $\mathbf{C}_Q$ above $\mathbf{PC}_q$. A message that attempts to raise $\mathbf{C}_Q$ above this ceiling will not be received.

Verified Upper Bound. Finally, the discretionary label $\mathbf{V}$ is reported to the receiver as a verified upper bound on the sending process's tracking label. The kernel checks that $\mathbf{T}_P \sqsubseteq \mathbf{V}$ (the send attempt fails if it is not), then makes $\mathbf{V}$ available to the receiving process. For example, if $\mathbf{V} = \{t\,\star, \mathbf{3}\}$, then the receiver can verify that P definitely has privilege for tag t. An explicit verified upper bound lets a process declare exactly which privilege it intends to exercise, avoiding, for example, the "confused deputy" problem [Hardy 1988].

A message that cannot be delivered because of a label failure is simply dropped. Usually the sender is not even informed of the failure, since failure notification would act as a covert channel. However, if the failure involves only the sender's process and discretionary labels—for example, the sender tried to grant privilege it does not possess—then it is safe to report an error message, and Asbestos does so.

The complete Asbestos label rules are presented in Figure 4. The Asbestos rule corresponding to $\mathbf{T}_P \sqsubseteq \mathbf{C}_Q$ is

$$\mathbf{T}_P \sqcup \mathbf{T}^+ \sqsubseteq (\mathbf{C}_Q \sqcup \mathbf{C}^+) \sqcap \mathbf{PC}_p. \tag{3}$$

The differences are as follows.

```
// Send a message to $Q$'s port $dest$
$\textbf{send}(dest, \text{data}, \mathbf{T}^+, \mathbf{T}^-, \mathbf{C}^+, \mathbf{V})$
    Let $Q$ be the process with receive rights for $dest$
  Requirements:
    (1) $\forall t : \mathbf{T}^-(t) = \mathbf{3}$ or $\mathbf{T}_P(t) = \star$
    (2) $\forall t : \mathbf{C}^+(t) = \star$ or $\mathbf{T}_P(t) = \star$
    (3) $\mathbf{C}^+ \sqsubseteq \mathbf{PC}_{dest}$
    (4) $\mathbf{T}_P \sqsubseteq \mathbf{V}$
    (5) $\mathbf{T}_P \sqcup \mathbf{T}^+ \sqsubseteq (\mathbf{C}_Q \sqcup \mathbf{C}^+) \sqcap \mathbf{PC}_{dest}$
  Effects:
    $\mathbf{T}_Q \leftarrow ((\mathbf{T}_P \sqcup \mathbf{T}^+ \sqcup \mathbf{T}_Q) \sqcap \mathbf{T}^-) \sqcap \mathbf{T}_Q^\star$
    $\mathbf{C}_Q \leftarrow \mathbf{C}_Q \sqcup \mathbf{C}^+$

$\textbf{new\_port}(L)$
    Let $p$ be an unused port
  Effects:
    $\mathbf{PC}_p \leftarrow L \sqcap \{p\,\mathbf{0}, \mathbf{3}\}$
    $\mathbf{T}_P(p) \leftarrow \star$
    Return $p$

$\textbf{set\_port\_label}(p, L)$
  Requirement:
    $P$ has receive rights for $p$
  Effect:
    $\mathbf{PC}_p \leftarrow L$
```

Fig. 4. Label operations associated with three Asbestos system calls. P is the calling process.

$\mathbf{T}_P \sqsubseteq \mathbf{C}_Q$	Core information flow rule
$\mathbf{T}_P \sqcup \mathbf{T}^+ \sqsubseteq \mathbf{C}_Q$	The $\mathbf{T}^+$ discretionary label increases the message's tracking label
$\mathbf{T}_P \sqcup \mathbf{T}^+ \sqsubseteq \mathbf{C}_Q \sqcup \mathbf{C}^+$	The $\mathbf{C}^+$ discretionary label grants clearance to the receiver, increasing its clearance label
$\mathbf{T}_P \sqcup \mathbf{T}^+ \sqsubseteq (\mathbf{C}_Q \sqcup \mathbf{C}^+) \sqcap \mathbf{PC}_p$	The port clearance label applies additional clearance restrictions

The Asbestos rule corresponding to $\mathbf{T}_Q \leftarrow \mathbf{T}_P \sqcup \mathbf{T}_Q$ is

$$\mathbf{T}_Q \leftarrow ((\mathbf{T}_P \sqcup \mathbf{T}^+ \sqcup \mathbf{T}_Q) \sqcap \mathbf{T}^-) \sqcap \mathbf{T}_Q^\star. \tag{4}$$

The differences are as follows.

$\mathbf{T}_Q \leftarrow \mathbf{T}_P \sqcup \mathbf{T}_Q$	Core information flow rule
$\mathbf{T}_Q \leftarrow \mathbf{T}_P \sqcup \mathbf{T}^+ \sqcup \mathbf{T}_Q$	The $\mathbf{T}^+$ discretionary label increases the message's tracking label
$\mathbf{T}_Q \leftarrow (\mathbf{T}_P \sqcup \mathbf{T}^+ \sqcup \mathbf{T}_Q) \sqcap \mathbf{T}^-$	The $\mathbf{T}^-$ discretionary label grants privilege to the receiver, lowering its tracking label
$\mathbf{T}_Q \leftarrow ((\mathbf{T}_P \sqcup \mathbf{T}^+ \sqcup \mathbf{T}_Q) \sqcap \mathbf{T}^-) \sqcap \mathbf{T}_Q^\star$	Q's privilege is preserved by restoring $\star$ levels to its tracking label

Figure 4 also presents the label rules for two operations involving ports, namely, creating a new port and changing a port's label. Processes supply an initial port label when creating a port; most often this is $\top = \{\mathbf{3}\}$, which adds no restrictions relative to the process's clearance label, but it can be $\{\mathbf{2}\}$ or anything else. One wrinkle is that when a process supplies the initial port clearance label L, it has no way of defining a specific level for the port's tag, which has not yet been assigned. Therefore, the kernel sets the new port's clearance to a more restrictive value $L \sqcap \{p\,\mathbf{0}, \mathbf{3}\}$, where p is the new port. Since initially $\mathbf{PC}_p(p) \le \mathbf{0}$ and any other process X has $\mathbf{T}_X(p) \ge \mathbf{1}$ (the default send level), no other process can send to p until P explicitly grants access. Processes can easily remove this restriction since only the initial port label is modified.

5.3 IPC Examples

We now work through several examples to demonstrate the different features of the IPC mechanism.

Examples A, B, and C demonstrate the core rule for process communication, Equation (3). In Example A, all tags have level $\mathbf{1}$ in $\mathbf{T}_P$ and level $\mathbf{2}$ in $\mathbf{C}_Q$; thus, $\mathbf{T}_P \sqsubseteq \mathbf{C}_Q$ and the message can be delivered. In Example B $\mathbf{T}_P \sqsubseteq \mathbf{C}_Q$ still holds (t: $\star < \mathbf{2}$, u: $\mathbf{0} < \mathbf{2}$, all others: $\mathbf{1} < \mathbf{2}$) so the message is delivered. However, in Example C, tag t has level $\mathbf{3}$ in $\mathbf{T}_P$ but the lower level $\mathbf{2}$ in $\mathbf{C}_Q$. Intuitively, Q doesn't have the clearance required to receive data labeled $t\,\mathbf{3}$; mathematically, $\mathbf{T}_P(t) \not\leq \mathbf{C}_Q(t)$, so $\mathbf{T}_P \not\sqsubseteq \mathbf{C}_Q$ and the message cannot be delivered. The kernel will silently drop the message when Q attempts to receive it. (The message shouldn't be dropped at send time because Q might change its clearance after the message is sent, but before attempting to receive.)

	$\mathbf{T}_P$	$\mathbf{C}_Q$	Old $\mathbf{T}_Q$	New $\mathbf{T}_Q$	Result
A.	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{\mathbf{1}\}$	Success
B.	$\{t\,\star, u\,\mathbf{0}, \mathbf{1}\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{\mathbf{1}\}$	Success
C.	$\{t\,\mathbf{3}, \mathbf{1}\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	—	Silent failure: $\mathbf{T}_P(t) = \mathbf{3} > \mathbf{C}_Q(t) = \mathbf{2}$, so $\mathbf{T}_P \not\sqsubseteq \mathbf{C}_Q$

In Examples D and E Equation (3) allows message delivery, but further label operations are necessary because the sender's tracking label is not less than or equal to the receiver's tracking label. When the message is received, the receiver's labels are updated according to Equation (4) to account for the new information; specifically, the receiver's tracking label is increased to the least upper bound of the sender's tracking label. However, as Example F shows, privilege is preserved.

	$\mathbf{T}_P$	$\mathbf{C}_Q$	Old $\mathbf{T}_Q$	New $\mathbf{T}_Q$	Result
D.	$\{t\,\mathbf{3}, \mathbf{1}\}$	$\{t\,\mathbf{3}, \mathbf{2}\}$	$\{\mathbf{1}\}$	$\{t\,\mathbf{3}, \mathbf{1}\}$	Success
E.	$\{t\,\mathbf{2}, \mathbf{1}\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{t\,\mathbf{2}, \mathbf{1}\}$	Success
F.	$\{t\,\mathbf{2}, \mathbf{1}\}$	$\{\mathbf{2}\}$	$\{t\,\star, \mathbf{1}\}$	$\{t\,\star, \mathbf{1}\}$	Success, but privilege is preserved

The remaining examples demonstrate how the discretionary labels interact with message delivery. In Examples G and H, the $\mathbf{T}^+$ label increases a message's effective tracking label: the kernel behaves as if the sending process's tracking label were $\mathbf{T}_P \sqcup \mathbf{T}^+$. The labels in Examples G and H have the same effect as those in Examples C and D, respectively, although in these examples the sending process's tracking label is the default $\{\mathbf{1}\}$.

	$\mathbf{T}_P$	$\mathbf{T}^+$	$\mathbf{C}_Q$	Old $\mathbf{T}_Q$	New $\mathbf{T}_Q$	Result
G.	$\{\mathbf{1}\}$	$\{t\,\mathbf{3}, \star\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	—	Silent failure: $\mathbf{T}^+(t) = \mathbf{3} > \mathbf{C}_Q(t) = \mathbf{2}$
H.	$\{\mathbf{1}\}$	$\{t\,\mathbf{3}, \star\}$	$\{t\,\mathbf{3}, \mathbf{2}\}$	$\{\mathbf{1}\}$	$\{t\,\mathbf{3}, \mathbf{1}\}$	Success

Examples I, J, and K show how $\mathbf{T}^-$ lets the sender declassify the destination's tracking label by reducing its levels for some tags. As demonstrated by Example I, the kernel requires that the sender have privilege for each tag it attempts to declassify. Examples J and K show how $\mathbf{T}^-$ supports both conventional declassification—a tag's level is reduced to the default—and the granting of privilege.

	$\mathbf{T}_P$	$\mathbf{T}^-$	$\mathbf{C}_Q$	Old $\mathbf{T}_Q$	New $\mathbf{T}_Q$	Result
I.	$\{\mathbf{1}\}$	$\{t\,\mathbf{0}, \mathbf{3}\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	—	Error: $\mathbf{T}^-(t) < \mathbf{3}$, but $\mathbf{T}_P(t) \neq \star$
J.	$\{t\,\star, \mathbf{1}\}$	$\{t\,\mathbf{1}, \mathbf{3}\}$	$\{t\,\mathbf{3}, \mathbf{2}\}$	$\{t\,\mathbf{3}, \mathbf{1}\}$	$\{\mathbf{1}\}$	Success with declassification
K.	$\{t\,\star, \mathbf{1}\}$	$\{t\,\star, \mathbf{3}\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{t\,\star, \mathbf{1}\}$	Success with granting privilege

Similarly, $\mathbf{C}^+$ lets the sender grant clearance to view secret data by increasing the receiver's clearance label. Again, privilege is required to use $\mathbf{C}^+$ (Example L).

	$\mathbf{T}_P$	$\mathbf{C}^+$	$\mathbf{C}_Q$	Old $\mathbf{T}_Q$	New $\mathbf{T}_Q$	Result
L.	$\{\mathbf{1}\}$	$\{t\,\mathbf{3}, \star\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	—	Error: $\mathbf{C}^+(t) > \star$, but $\mathbf{T}_P(t) \neq \star$
M.	$\{t\,\star, \mathbf{1}\}$	$\{t\,\mathbf{3}, \star\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{\mathbf{1}\}$	Success with granting clearance: new $\mathbf{C}_Q = \{t\,\mathbf{3}, \mathbf{2}\}$

Finally, Examples N, O, and P demonstrate the use of $\mathbf{V}$, the kernel-verified upper bound label passed to the receiver. In Example N, the sender is prevented from claiming an upper bound lower than its actual tracking label. In Example O, the sender informs the receiver of its precise tracking label; in Example P, the sender shows that no tags in its tracking label have level $\mathbf{3}$, but does not reveal that label's precise contents.

	$\mathbf{T}_P$	$\mathbf{V}$	$\mathbf{C}_Q$	Old $\mathbf{T}_Q$	New $\mathbf{T}_Q$	Result
N.	$\{\mathbf{1}\}$	$\{t\,\mathbf{0}, \mathbf{1}\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	—	Error: $\mathbf{T}_P \not\sqsubseteq \mathbf{V}$
O.	$\{t\,\mathbf{2}, \mathbf{1}\}$	$\{t\,\mathbf{2}, \mathbf{1}\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{t\,\mathbf{2}, \mathbf{1}\}$	Success
P.	$\{t\,\mathbf{2}, \mathbf{1}\}$	$\{\mathbf{2}\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{t\,\mathbf{2}, \mathbf{1}\}$	Success (as long as $\mathbf{V}$ is accurate, it need not be precise)

5.4 Label Idioms

The Asbestos label mechanism can implement many different policies, but most policies will generally be built from the handful of idioms we detail below.

Secrecy. A process that wants to protect a piece of information from unauthorized processes can create a new tag t and use $\mathbf{T}^+ = \mathbf{C}^+ = \{t\,\mathbf{3}, \star\}$ when sending that information to other processes. These discretionary labels simultaneously mark the message as secret and grant other processes the ability to receive the secret. Specifically, any process that sees the information will have its tracking label updated to level $\mathbf{3}$ for tag t. However, since these processes lack the necessary privilege, they cannot send the secret beyond those processes cleared by the secret's "owner." The owner process could grant clearance to receive the secret without sending the secret by only using $\mathbf{C}^+ = \{t\,\mathbf{3}, \star\}$. Level $\mathbf{3}$ prevents all processes except those explicitly granted clearance from viewing the secret information. When using this idiom, we often refer to processes that have seen the secret as *contaminated* with respect to the corresponding tag.

Multilevel Security. An extension of the secrecy idiom can build a policy with different levels of secrecy, similar to multilevel security (MLS). A policy where each security category has levels of *unclassified*, *confidential*, *secret*, and *top-secret* requires three tags per category. For example, n_c might represent nuclear-confidential information, n_s nuclear-secret information, and n_t nuclear-top-secret information. The unclassified state is represented by a tracking label with the default level $\mathbf{1}$ for all three tags. The tracking label of a process that has seen nuclear-classified information would contain $n_c\,\mathbf{3}$, which in turn requires clearance of $n_c\,\mathbf{3}$. Secret and top-secret information could be expressed with $n_s\,\mathbf{3}$ and $n_t\,\mathbf{3}$, respectively, but the implied hierarchy among MLS security levels is better modeled by also marking secret data as classified and top-secret data as both secret and classified. This leads to labels containing $n_s\,\mathbf{3}, n_c\,\mathbf{3}$ and $n_t\,\mathbf{3}, n_s\,\mathbf{3}, n_c\,\mathbf{3}$, respectively. The privilege to declassify nuclear-top-secret data to the secret level is represented by a tracking label containing $n_t\,\star$, and similarly for other declassification privileges.

Integrity. In an integrity policy, the application wants to know that all information used in a computation came from sources that are considered high integrity. For this policy, we consider a source (a process or file) as high integrity for a tag if its tracking label has that tag at level $\mathbf{0}$. After allocating tag t from the kernel, a process can mark other processes as high integrity for that tag using $\mathbf{T}^- = \{t\,\mathbf{0}, \mathbf{3}\}$. Any communication thereafter will update this integrity appropriately. For instance, if two high-integrity processes P and Q communicate, then both will remain high-integrity since $(\mathbf{T}_P \sqcup \mathbf{T}_Q)(t) = \mathbf{0}$. However, if P receives a message from a process without high integrity for t, then $\mathbf{T}_P(t)$ will raise to at least level $\mathbf{1}$ by Equation (4), causing P to lose its integrity.

Taint. Perl and some other languages support taint tracking, where selected functions are not allowed to operate on "tainted" data (that is, data originating from an untrusted source). Asbestos can enforce this idiom at the process level. For example, the network daemon could mark all incoming data with a *network taint tag*, and other sensitive processes might refuse to accept messages with this taint tag. To mark data as tainted in a particular category, a process creates a new tag t for that category of taint, then uses $\mathbf{T}^+ = \{t\,\mathbf{2}, \star\}$ when sending that information. Most processes will be able to receive the message because the default clearance level is $\mathbf{2}$. However, process that are not willing to receive that taint can lower their clearance label to $\{t\,\mathbf{1}, \mathbf{2}\}$. This policy is made possible by the different default levels for the tracking and clearance labels. Without different default levels, this policy would require system-wide label changes to add t to most processes' clearance labels.

Capabilities. With clearance and port clearance labels, a process P can implement a capability-like communication pattern in which the ability to send messages to P is distributed to other processes as an explicit right. P requests a tag t from the kernel, then reduces its clearance label to level $\star$ for t. Only processes with $t\,\star$ in their tracking label satisfy Equation (3) and thus may send messages to P. The $\mathbf{T}^-$ discretionary label can be used to send the capability to other processes. Alternately, P may modify port clearance labels to implement capability-like semantics with per-port granularity. This idiom doesn't directly support capability revocation, but Asbestos can support revocation using common patterns such as object capabilities.

Isolation. This idiom isolates a process P behind a proxy process Q, which acts as a sort of firewall: the isolated process P can communicate with no other process but Q, while Q has no restrictions on its communication. Isolation is implemented as a simple combination of the secrecy and capability idioms and declassification. Q implements the isolation idiom by creating two tags,

t and u. This gives Q privilege for the tags. Q adds $t\,\mathbf{3}$ to its clearance label, allowing it to receive secret t data, and starts up P with tracking label $\mathbf{T}_P = \{t\,\mathbf{3}, u\,\mathbf{0}, \mathbf{1}\}$ and clearance label $\mathbf{C}_P = \{t\,\mathbf{3}, u\,\mathbf{0}, \mathbf{2}\}$. The t components prevent P from sending messages to any process other than Q, while the u components prevent P from receiving messages from any process but Q. (Strictly speaking, of course, P can always communicate with itself and with other processes with identical labels, but only P and Q are able to create such processes.) Only Q has the ability to change P's labels in a way that will circumvent the isolation policy.

5.5 Discussion

Levels. Asbestos's five label levels ($\star$, $\mathbf{0}$, $\mathbf{1}$, $\mathbf{2}$, $\mathbf{3}$) are required to support privilege, integrity, taint tracking, and secrecy within the framework of one unified tracking label per process. Since privilege ($\star$) is handled differently by the update rules, it must be represented by a unique level. The secrecy and integrity idioms require at least one level below ($\mathbf{0}$) and one level above ($\mathbf{3}$) the tracking label and clearance label default levels, respectively. Finally, supporting the taint idiom requires different default levels for tracking labels ($\mathbf{1}$) and clearance labels ($\mathbf{2}$). While additional levels might increase expressivity, for instance by directly supporting additional hierarchical security classifications in the multi-level security idiom, we have preferred to keep the base mechanism relatively simple and implement such policies with multiple tags.

Tag Names. Asbestos security requires that when a process creates a new tag, that process is initially the only process in the system with privilege for that tag. This implies that tags must be unique since boot. Also, tag names must not follow a predictable order, since such an order could be used as a covert channel to leak information between processes. To ensure that tags are not reused and that their names follow no discernible pattern, the kernel generates tags by encrypting a counter with a 61-bit block cipher derived from Blowfish [Schneier 1993]. The cipher is a one-to-one mapping: as long as the counter does not wrap, the resulting tag values will be unique. In the current implementation, a process allocating tags as fast as possible would take over 500,000 years to allocate them all. Nevertheless, the allocation procedure ensures system security by skipping tags that are in use at allocation time.

Privilege. Asbestos privilege is decentralized in that the only way to obtain privilege for a particular tag is by receiving it from a process that already has that privilege. Therefore, there is no inherently powerful "root" user, though by convention users may grant most privilege to some administrator.

The label rules generally exercise privilege implicitly. For instance, receiving a message preserves process privilege with no explicit process intervention. However, one aspect of Asbestos privilege requires explicit process action: for a sending process to prove its privilege to a receiving process, it must explicitly indicate what privilege it intends to exercise with the $\mathbf{V}$ discretionary label. An alternative design might eliminate $\mathbf{V}$ and supply message recipients with a copy of the sender's tracking label, in effect conveying all of a process's credentials with every message it sends. However, such designs lead to security problems in which an attacker can trick a process into exercising unintended privileges [Hardy 1988].

5.6 Implementation

Asbestos applications represent labels, such as the discretionary labels provided when sending a message, using a default level plus an array of tag-level pairs. A 64-bit number represents a label entry: the upper 61 bits are the tag, the lower 3 bits encode its level in that label. A library provides a basic implementation of label operations for application use; to get reasonable run time for label operations, it stores the array of tags in sorted order.

The kernel represents each active tag with an 84-byte data structure called a *tagnode*. For ports, this structure includes the port clearance label and a reference to the process with receive rights. A hash table maps tags to tagnodes. Tagnodes are reference counted; when all kernel references to a tagnode disappear, the kernel may reuse its memory.

Our initial work represented labels in the kernel with a sorted array of pairs of tagnode pointers and levels [Efstathopoulos et al. 2005], somewhat like our current application implementation. Unfortunately, while applications compute and operate on labels relatively infrequently, and discretionary labels tend to be quite small, the Asbestos kernel performs several label operations per IPC and processes with privilege for many different tags will have large tracking labels. The privileged components in our Web server have privilege for each application user; we measure the Web server with as many as 145,000 active users, which gives some processes as many as 300,000 tags (two tags per user) at a nondefault level. A simple sorted array implementation imposes label operation costs that always scale linearly with the number of tags in the largest label, and these costs quickly became a performance bottleneck.

In the worst case, any label operation can have overhead linear in the size of the input labels. (For example, consider $L = \{t_0\,\mathbf{2}, t_2\,\mathbf{2}, \ldots, t_{2i}\,\mathbf{2}, \mathbf{0}\}$ and $L' = \{t_1\,\mathbf{3}, t_3\,\mathbf{3}, \ldots, t_{2i+1}\,\mathbf{3}, \mathbf{1}\}$; the least upper bound operation $L \sqcup L'$ requires the construction of a label with $2i + 2$ components $\{t_0\,\mathbf{2}, t_1\,\mathbf{3}, \ldots, t_{2i}\,\mathbf{2}, t_{2i+1}\,\mathbf{3}, \mathbf{1}\}$.) However, the label operations performed on behalf of most Asbestos applications fit into categories amenable to optimization. In the Asbestos Web server, we observe that large labels consist mainly of privilege, with at most a handful of non-privileged tags. Label comparison operations usually succeed: when they fail a message is silently dropped, which indicates either an exploit attempt or an application bug—both, we hope, rare. Label update operations usually do not increase the receiving process's tracking label. However, tracking labels change frequently, usually to add or remove privilege for connection tags as connections enter and leave the system. Furthermore, the use of discretionary labels creates many short-lived labels that differ only slightly from longer-lived tracking and clearance labels. These qualities seem general enough to be common to other challenging applications, not just Web serving.

We considered several strategies for improving label performance. Hierarchical privilege groups could shrink the size of privileged processes'

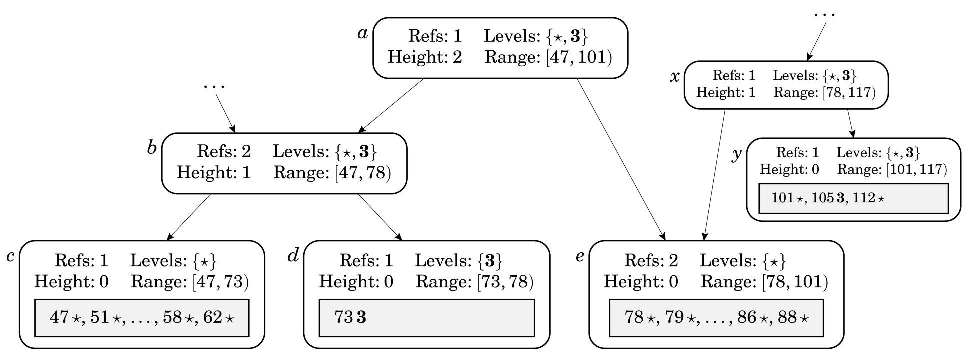

Fig. 5. Labels are AVL trees. The label rooted at node a references nodes a through e; nodes x and y belong to other labels. Nodes c, d, e, and y are leaf nodes containing sorted chunks of tagnode/level pairs. Nodes b and e are referenced from other labels and therefore have a reference count of two. If the label rooted at a is L, then a comparison operation such as $L \sqsubseteq L'$ can skip nodes c and e: all tags in these nodes have level $\star$, which is less than or equal to any possible level. In larger labels, entire subtrees can often be skipped, leading to logarithmic performance scaling for typical operations.

labels, letting one “superprivilege” stand in for 300,000 separate tags. Such a design would complicate individual privilege comparisons. We also wanted to avoid privilege groups, worrying that users would fix application bugs by giving processes the group equivalent of “root” privilege. Another potential strategy, namely caching the results of past label operations, would perform poorly due to the high frequency of label changes: most cached results would quickly be invalidated.

Our solution is a balanced tree implementation. This gives label operations on typical large labels costs logarithmic in the size of the inputs and the memory cost of temporary labels is reduced by allowing labels to share substructure. The tree implementation allows us to evaluate labels an order of magnitude larger than our previous work. In our Asbestos Web server, the new implementation with 300,000-tag labels performs better than the original sorted array implementation at almost any size.

Asbestos labels are AVL trees whose leaves are sorted *chunks* of tagnode pointer and level pairs (Figure 5). We ensure that tagnodes are 8-byte aligned, allowing us to store a tagnode pointer and a level in a single 32-bit slot. Labels are reference counted and updated copy-on-write, so multiple entities can share label memory when appropriate. Additionally, because the only state that an AVL tree needs in each node is the node’s height in the tree, internal and leaf nodes can be shared among labels. Empirically, we had the best performance when storing a maximum of 24 label components per chunk.

Each tree node contains its height in the tree, to facilitate balancing; the range of tagnode pointers in the subtree, to enable subtree comparisons; the set of levels in the subtree; and other data structure maintenance fields. The set of levels helps optimize operations on labels that mostly contain privilege. For example, Figure 6 shows a pseudocode sketch for comparing two labels with the $\sqsubseteq$ operator. Ignoring default levels, if all of the a node’s levels are less than

```
bool label_leq(label_node $a$, label_node $b$) {
    if ($a$.default_level > $b$.default_level)
        return false;
    else if (!range_overlap($a$.range, $b$.range)
            || levelset_max($a$.levels) < levelset_min($b$.levels))
        return levelset_max($a$.levels) $\leq$ $b$.default_level
                && levelset_min($b$.levels) $\geq$ $a$.default_level;
    else if ($a$.height == 0 && $b$.height == 0)
        return label_chunk_leq($a$, $b$);
    else if ($a$.height > $b$.height)
        return label_leq($a$.right, $b$) && label_leq($a$.left, $b$);
    else
        return label_leq($a$, $b$.right) && label_leq($a$, $b$.left);
}
```

Fig. 6. Pseudocode for checking whether $a \sqsubseteq b$. Entire subtree comparisons can be skipped if all of the a subtree's levels are less than any of the b subtree's levels. Some computations with default levels are omitted.

any of the b node's levels, then $a \sqsubseteq b$. In the common case where a sending process's label consists mostly of privileged tags, this allows the comparison operator to skip the vast majority of a label's nodes.

6. LABEL PERSISTENCE

Any general purpose operating system must support persistent storage. In systems that support information flow control, the storage system must uphold the constraints imposed by the configuration of labels in place when the data is stored. However, stored data is usually accessed after the system label state has changed, and often after it has been completely cleared by one or more reboots. For example, if a file contains data that is marked secret with a label of $\{t\ \mathbf{3}, \mathbf{1}\}$, an application must have the appropriate clearance to read the data. After a reboot, there are no processes that have or can acquire that privilege. The storage system itself must thus preserve the privilege needed to access the data it stores.

Our file system semantics resemble those of HiStar [Zeldovich et al. 2006] except for the way privilege is stored. HiStar and other systems such as EROS [Shapiro and Hardy 2002] preserve privilege by introducing a single-level store. Rebooting returns the system to a checkpointed state, and a process's tags and capabilities are stored along with its virtual memory. One consequence of this design is that privilege is tied to process lifetime: after the last process with privilege for a tag t dies, there is no way to recover that privilege.

Asbestos introduces an alternate technique called *pickling* that preserves privilege within a more conventional stable storage design. A pickle is a special type of file that stores privilege for a single tag. A special *unpickle* operation allows a process to recover privilege for the tag if various constraints are satisfied. Pickles simplify the process of recovering privilege, whether it be after application restart or reboot, and fit well into existing file system designs. The Asbestos file server preserves privilege and upholds information flow invariants while avoiding covert channels through file metadata such as names and labels. While these properties might be easy to provide if the file server could

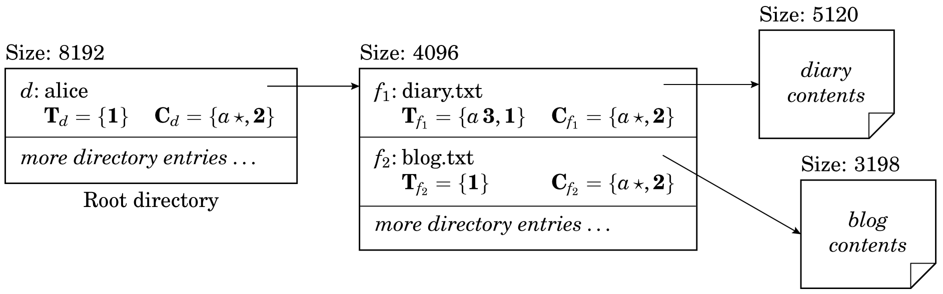
Size: 8192
d: alice
$\mathbf{T}_d = \{\mathbf{1}\}$ $\mathbf{C}_d = \{a\star, \mathbf{2}\}$
more directory entries ...
Root directory
Size: 4096
f_1: diary.txt
$\mathbf{T}_{f_1} = \{a\,\mathbf{3}, \mathbf{1}\}$ $\mathbf{C}_{f_1} = \{a\star, \mathbf{2}\}$
f_2: blog.txt
$\mathbf{T}_{f_2} = \{\mathbf{1}\}$ $\mathbf{C}_{f_2} = \{a\star, \mathbf{2}\}$
more directory entries ...
Size: 5120
diary contents
Size: 3198
blog contents

Fig. 7. File and directory labels. User Alice owns a publicly readable directory */alice*, a publicly readable file */alice/blog.txt*, and a private file */alice/diary.txt*. Only processes with privilege tag $a\star$ may modify her files.

arbitrarily create privilege and allocate tags with specific values, for higher assurance, the Asbestos file server operates within the same rules as ordinary Asbestos processes.

6.1 File System Semantics

The label rules for file operations are similar to the label rules for processes. Each file f in the Asbestos file system has a tracking label $\mathbf{T}_f$ and a clearance label $\mathbf{C}_f$. The tracking label resembles a process's tracking label. It records the information flow corresponding to the file's contents; when a process reads from f, the file server's reply sets $\mathbf{T}^+ = \mathbf{T}_f$ to preserve information flow constraints. The clearance label constrains processes attempting to write to the file. A process P with tracking label $\mathbf{T}_P$ may only write to a file f if $\mathbf{T}_P \sqsubseteq \mathbf{C}_f$. For example, in Figure 7, if user Alice has privilege for tag a and creates a file *blog.txt* with clearance label $\mathbf{C}_{\mathit{blog.txt}} = \{a\star, \mathbf{1}\}$, then the only processes that may modify *blog.txt* are processes to which Alice grants the privilege $a\star$ (an instance of the capability idiom).

Directories have tracking and clearance labels exactly like regular files. If process P lists directory d, its tracking label $\mathbf{T}_P$ will be updated to reflect the $\mathbf{T}_d$ label. Operations that modify a directory, such as creating or removing files, are treated as writes to the directory. For example, if a process P is to create a file in directory d, its tracking label $\mathbf{T}_P$ must obey $\mathbf{T}_P \sqsubseteq \mathbf{C}_d$.

Unlike process labels, file and directory labels are immutable; a file meant to hold a secret must be created with an appropriate label. The file server requires processes to supply the immutable tracking and clearance labels at file creation time. It also checks some constraints on those initial labels. The new file's tracking label must be at least as high as the tracking label of the creating process ($\mathbf{T}_P \sqsubseteq \mathbf{T}_f$), maintaining valid information flow. Furthermore, the file's clearance label must be less than or equal to its tracking label ($\mathbf{C}_f \sqsubseteq \mathbf{T}_f$). This is the reverse of the rule for processes due to the different meaning of clearance in the file context. The immutable label design, which was influenced by HiStar, simplifies certain information flow guarantees. Designs that allow a file's tracking label to change are either more complex or leak information. Although immutable labels may appear cumbersome, in practice it has not

Operation	**Requirements**	**Results**
read(f)	$\mathbf{T}_f \sqsubseteq \mathbf{C}_P$	$\mathbf{T}_P \leftarrow \mathbf{T}_P \sqcup \mathbf{T}_f$
write(f)	$\mathbf{T}_P \sqsubseteq \mathbf{C}_f$	
create$(f, d, \mathbf{T}, \mathbf{C})$	$\mathbf{T}_P \sqsubseteq \mathbf{C}_d,\ \mathbf{T}_d \sqsubseteq \mathbf{C}_P,\ \mathbf{T}_P \sqsubseteq \mathbf{T}, \mathbf{C} \sqsubseteq \mathbf{T}$	$\mathbf{T}_P \leftarrow \mathbf{T}_P \sqcup \mathbf{T}_d, \mathbf{T}_f \leftarrow \mathbf{T}, \mathbf{C}_f \leftarrow \mathbf{C}$
pickle$(f, d, \mathbf{T}, \mathbf{C}, t, \ell, pass)$	$\mathbf{T}_P \sqsubseteq \mathbf{C}_d,\ \mathbf{T}_d \sqsubseteq \mathbf{C}_P,\ \mathbf{T}_P \sqsubseteq \mathbf{T}, \mathbf{C} \sqsubseteq \mathbf{T},$ $\mathbf{T}_P(t) = \star,\ \mathbf{T}_{FS}(t) = \star$	$\mathbf{T}_P \leftarrow \mathbf{T}_P \sqcup \mathbf{T}_d, \mathbf{T}_f \leftarrow \mathbf{T}, \mathbf{C}_f \leftarrow \mathbf{C},$ $t_f \leftarrow t, \ell_f \leftarrow \ell, pass_f \leftarrow pass$
unpickle$(f, \mathbf{3}, pass)$	$\mathbf{T}_f \sqsubseteq \mathbf{C}_P$	$\mathbf{T}_P \leftarrow \mathbf{T}_P \sqcup \mathbf{T}_f$
unpickle$(f, \ell, pass)$ where $\ell < \mathbf{3}$	$\mathbf{T}_P \sqsubseteq \mathbf{C}_f, \mathbf{T}_f \sqsubseteq \mathbf{C}_P, \ell \geq \ell_f,$ $pass = pass_f$	$\mathbf{T}_P \leftarrow (\mathbf{T}_P \sqcup \mathbf{T}_f) \sqcap \{t_f\, \ell, \mathbf{3}\}$

Fig. 8. File operations on file f in directory d by process P.

been difficult to determine a file's intended label at creation time. Figure 8 summarizes the label rules for file system operations.

The immutability of file labels and the semantics of the create operation allow directory read operations to return most metadata, including file names, file labels, and inode numbers. This is possible because directory read operations do not return state that can be affected by processes with tracking labels higher than the directory's. By definition, a process P can create a file in directory d only if $\mathbf{T}_P \sqsubseteq \mathbf{C}_d$ and $\mathbf{C}_d \sqsubseteq \mathbf{T}_d$; thus the file name and file labels were specified by, and are no more secret than, a process with $\mathbf{T}_P \sqsubseteq \mathbf{T}_d$. The ability to freely read a file's label without being influenced by it allows a process to safely determine how its labels would change as a result of reading a file. Operations that return mutable file attributes, such as file size, must update the process's tracking label with the file's tracking label to maintain information flow constraints, since these attributes can be modified by processes that have too many secrets to write to the directory. For example, a directory with $\mathbf{C}_d = \{\mathbf{1}\}$ might contain a file with $\mathbf{T}_f = \{t\, \mathbf{3}, \mathbf{1}\}$. A process with $\mathbf{T}_P = \mathbf{T}_f$ could not write to the directory, but could write to the file, changing its size.

6.2 Preserving Privilege with Pickles

In order to preserve privilege, the Asbestos file server must serialize labels in a way that allows it to find equivalent tags later, possibly after a reboot, despite the complication that tag names are nonpersistent and randomly generated. Therefore, the Asbestos file system directly expresses tag serialization with pickle files. When unpickling, depending on the parameters used to create a pickle file, a process may gain some amount of privilege for a pickle's tag via the unpickle operation.

To create a pickle file for tag t, process P sends a request to the file system containing t, an optional password, and the maximum privilege the file system should grant as the result of an unpickle operation. This maximum privilege is specified as a minimum level ℓ. If a password is specified during pickling, the same password must be used when obtaining privilege by unpickling. Since a pickle is a file, P also specifies a pathname and file labels so the file server can perform all the normal file creation checks. The file server also confirms that P has privilege for t and that the file server itself has been granted privilege for t. (In order to implement pickles and to correctly express file information flow,

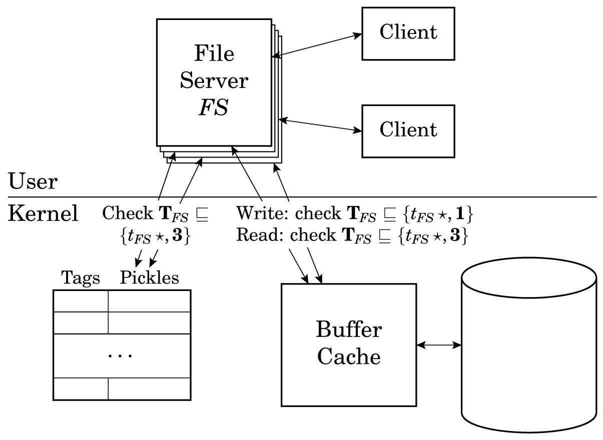
File Server FS
Client
Client
User
Kernel
Check $\mathbf{T}_{FS} \sqsubseteq \{t_{FS}\star, \mathbf{3}\}$
Write: check $\mathbf{T}_{FS} \sqsubseteq \{t_{FS}\star, \mathbf{1}\}$
Read: check $\mathbf{T}_{FS} \sqsubseteq \{t_{FS}\star, \mathbf{3}\}$
Tags
Pickles
...
Buffer Cache

Fig. 9. File server architecture. Each client talks to a unique file server event process. The event processes can access pages from the buffer cache as well as modify the pickle/tag mapping table.

the file server must have privilege for all serialized tags.) Once these checks succeed, the file system can create the pickle file.

There are two kinds of unpickle operations. Unpickling with a level of **3** requests no additional privilege, so the file server simply replies with the value of the tag for that pickle; the correct password is not required. Unpickling with a different level acquires a pickle's stored privilege. A process Q making such a request supplies the pathname of the pickle, the password (if any), and the desired privilege level, which must be greater than or equal to the level stored in the pickle. The file system then checks whether Q passes the normal file system checks for reading and writing the pickle file. For pickle files, write permission has been repurposed to indicate who can extract privilege. If Q passes these checks, the file system tells Q the tag value of the pickle and uses $\mathbf{T}^-$ to grant privilege for the tag at the desired level.

The file server can recover privilege after a reboot by simply creating a new tag for each pickle. Although the new tag values will likely not equal the values from before the reboot, applications will not be able to tell the difference as long as they store references to pickle files instead of storing raw tag values. Within a boot, however, the file system must remember the specific tag mapped to each pickle.

6.3 File Server Implementation

The Asbestos storage system is composed of a user-level file server and two kernel components, a buffer cache and a pickle/tag mapping table (Figure 9). Processes access the file system by communicating with the file server, which accesses the disk, buffer cache, and mapping table through special kernel interfaces.

To write data to the disk, the file server first makes sure all the contamination, taint, and integrity associated with the data has already been serialized:

that is, all tags at a level other than **1** have pickle equivalents. This ensures that the file server can use the data it is about to write in a consistent way after a reboot.

The file server has read and write privileges to the raw disk blocks, so it is effectively trusted with all data in the file system and all pickled tags. However, the file server is not completely privileged and is not trusted with any tags or data outside the file system. Specifically, the kernel requires that the file server's tracking label be less than or equal to {**1**} when writing to the disk. This means that the file server must have privilege for all tags on any data going to disk. As a consequence, processes may protect especially secret data from ever being written to disk using a tag that is never granted to the file server. In essence, the file system is only as privileged as it is trusted by the processes that store data in it. This differs from HiStar's single-level-store design, in which the equivalent of the file system is trusted for all tags.

The file server allows arbitrarily labeled processes to read from the file system, although they cannot write to it. This increases flexibility. The file server avoids contamination from these tainted clients by isolating their interactions using event processes (Section 7). However, in order to provide a consistent view of the file system, the file server must coordinate these event processes' states. This coordination is made safe from storage channels via two trusted kernel components, a buffer cache and a pickle/tag mapping table. These trusted components are relatively simple (about a quarter the code length of the untrusted components). They are accessible only to processes with privilege for a special tag, which the kernel initially grants to the file server. The buffer cache follows a conventional design. Only processes with tracking label at or below {**1**} can write to the buffer cache, but tainted processes can read from the buffer cache since reading doesn't alter the system in a way observable from other processes (except by timing). When a page is written to the buffer cache, the kernel notifies any other process that had previously read the page in order to maintain consistency. The pickle/tag mapping table maintains the definitive correspondence between pickles and tags, and is an instance of a general tag mapping interface available for other uses. The mapping table's interface was designed to be free of covert channels. For example, the table maps tags to pickles via the single request "give me a tag for this pickle value." This may create a new mapping (and a new tag) or return an existing mapping, but because tag values are random, there is no way to determine which of the two cases has occurred and therefore no way for the event process to use the mechanism to transmit information. Interfaces such as "check whether this pickle has a tag" would expose information and are therefore not provided.

6.4 Discussion

In order to obtain privilege from a pickle, the process must be able to read and write the pickle as well as read the directory structure down to where the pickle is stored. These requirements can effectively restrict the tracking and clearance labels of any process acquiring the stored privilege. For example, if $\mathbf{T}_P$ includes

$t\star, u\star$, then P can pickle u with $\mathbf{C}_u = \{t\star, \mathbf{3}\}$. This means that another process Q may only acquire privilege for u if it has $\mathbf{T}_Q(t) = \star$, another instance of the capability idiom.

Password protected pickles enable applications to construct their own privilege hierarchies. In order to create an independent privilege hierarchy, an application would first create a password protected pickle with an empty tracking and clearance label to root its hierarchy. It would then create a directory and pickle tree, protected by the first pickle, to match the privilege hierarchy it desires. After a system reboot, the application can recover the entire privilege hierarchy by unpickling the root pickle and using it to unpickle the subsequent levels of the hierarchy.

7. EVENT PROCESSES

Asbestos application designers must manage how information flows through their applications. Server-like processes that handle multiple users' private data present a challenging information flow pattern; absent any active management, they would accumulate, and therefore spread, the tags used to protect users' privacy as they handled different users' requests. The standard approaches to managing this pattern of information flow have several disadvantages. One standard approach would isolate users by forking the server on a per-user or per-request basis. Unfortunately, this could be resource intensive, requiring hundreds or thousands of page tables, labels, and so forth, and would require that the initial request for each flow of communication (the one that causes the server to fork) be sent with no tags protecting it, revealing at least when and how many requests were made to the server. Another direct approach is to have the server cleanse itself of tags after handling each request. With Asbestos labels, there are two ways to remove tags: have privilege for the tags so the contamination doesn't stick in the first place, or ask a privileged process to declassify the server process. Both of these options leave the process over-trusted (either completely privileged with respect to the data, or trusted to "forget" data) and therefore vulnerable to disclosing many users' data. To handle the information flow pattern found in server-like processes, we need a mechanism that provides data isolation at an acceptable resource cost.

Examining how servers use the data that needs to be isolated helped us develop a mechanism that isolates different flows of communication. Many efficient servers [Pai et al. 1999; von Behren et al. 2003; Krohn 2004] are driven by a simple dispatch loop:

```
while (1) {
    event = get_next_event();
    user = lookup_user(event);
    if (user_is_new(user))
        user.state = create_state();
    process_event(event, user);
}
```

```
ep_checkpoint(&msg);
if (!state.initialized) {
    initialize_state(state);
    state.reply = new_port();
}
process_msg(msg, state);
ep_yield();
```

Fig. 10. Core loop for a typical event process-based server.

This arrangement is efficient, since only one process is involved, and there is little space overhead beyond the minimum memory required to hold each user's state. Furthermore, the process only accesses the state of one user at a time, so if some mechanism made it impossible to access more than one user's data at a time, the server would hardly notice the difference. That mechanism would have to isolate the state of different users and ensure that the process's labels would correspond to the accessible user state.

7.1 The Event Process Abstraction

The Asbestos *event process* abstraction bundles the process state that is specific to a particular flow of communication into an addressable entity. This allows a process to isolate a user's data and privilege by handling each user's requests in a different event process. Conceptually, an event process is a forked copy of the process's address space and any state specific to the address space, such as labels. The event process mechanism automatically creates new event processes for new communication flows and delivers messages for existing flows to the appropriate event processes. Automatic forking is used because explicitly creating event processes for new flows would be fraught with covert channels.

Event processes are usually inactive and only run when they are explicitly handling a request. This usage pattern is evident in the event process interface: event processes always start running by receiving a message via the **ep_checkpoint** system call; after processing the request, they call **ep_yield** to sleep until receiving the next request. The code for a typical event process-based server resembles an event-driven dispatch loop; see Figure 10. Many event process entities effectively share the event loop, each with its own isolated state. The two **ep** system calls manage control flow transfer between events. The "state" variable refers to a different user in each event process.

A process enters the event process realm the first time it calls **ep_checkpoint**, after which the process itself will never run again. Instead, event processes derived from the *base process* can run. The process remains idle until a message arrives on a port for which the base process, or any of its event processes, holds receive rights. If the base process holds the port's receive rights, the kernel creates a new event process and delivers the message in the new event process's context. The new event process starts with tracking and clearance labels and memory state copied from the base process and no ports of its own. If, on the other hand, an existing event process holds receive rights for the destination port, the message is delivered in that event process's context, including its possibly modified labels and memory. In either case, any label changes performed by the event process or due to message delivery only apply to that event process's labels, receive rights for created ports are granted to the running event process, and changes to memory only apply to the event process that made them. After it finishes processing a message, an event process calls the **ep_yield** system call. This call saves any changes to labels, memory, and so forth, and then suspends the process, just as when the base process first called **ep_checkpoint**. No event process will run until another message is delivered, at which point execution again resumes at the code directly after the call to **ep_checkpoint**.

We expect event processes to perform tasks that change relatively small portions of the address space, and take advantage of this to save memory. All event processes with the same base process share the same page table. When an event process runs, we change the page table to reflect the changes the event process has made to its address space. Furthermore, event processes often make temporary modifications to memory (such as the stack) that are only useful while handling the current event. To keep the kernel from unnecessarily saving such memory modifications across **ep_yield**, an event process can call **ep_clean** to undo all memory changes within the specified address range. Later, when an event process is done, it can free all its resources with the **ep_exit** system call. The kernel state for each event process consists of tracking and clearance labels (which may share subtrees with the base process's labels), receive rights for ports, a little kernel bookkeeping information, and pages of memory that differ from the base process.

Usage. Messages addressed to a base process port typically correspond to new client processes or new client network connections, exactly the situations in which it is appropriate to create application state for a user. An event process can discover its own creation by checking and setting a memory location that the base process initialized to zero; a new event process inherits the zero, while a re-activated event process will see its previous nonzero write to that location. A new event process will typically allocate a new port on which to receive messages, as in line 4 of Figure 10. Messages to this port will always be delivered to the current event process (as long as it does not transfer receive rights for the port). Consequentially, event processes can send queries to the file server or other processes on behalf of the current user and later receive replies on the new port. When the event can be handled without waiting for any replies, an event process can exit without creating a port, though that particular event process will never run again.

The base process does not explicitly create event processes, nor does it know of their existence. In fact, once it calls **ep_checkpoint**, the base process never directly executes again, nor is there a way to change its memory. Different event processes are also unaware of each other's existence except possibly through message-based communication, which preserves the independence and isolation represented by per-event process labels. It would be possible to design mechanisms for event processes to selectively share memory subject to label checks.

The file server shows a use of event processes in practice. Recall that the file server is only privileged for tags entrusted to it by applications. Furthermore, it can only write data to disk when it is able to use that privilege to completely declassify the data. However, the file server should still be able to read data on behalf of processes with information flow that the file server can't declassify. Event processes let the file server serve these contaminated processes without spreading contamination to other clients. Each client of the file server communicates with a unique event process. Since each event process acts like a fork of the base process from the perspective of information flow, nonpickled tags are not spread among the clients. Therefore, using event processes allows our file server to effectively serve reads to contaminated clients.

7.2 Discussion

Typically, programs scatter users' data across the stack in addition to various places on the heap. This would lead to a relatively large number of pages that are unnecessarily specific to each event process. Some of these pages merely hold temporary variables, but others must persist across the processing of several messages. Storing the nontemporary data in a contained data structure on the heap can minimize the number of persistent pages required. This data management technique is much more natural in event-driven programming, which the event process system calls symbiotically encourage. Thus, event processes tend to use minimal private memory, and the optimization of only storing page table differences is profitable in practice (1.4 pages per event process for our test application).

Our current implementation of event processes uses a single thread of execution per base process. This implementation detail violates the isolation of individual event processes. For instance, an event process may block indefinitely in **recv**, blocking all event processes that belong to the corresponding base process. We believe that it would be possible to independently schedule event processes without much performance impact, eliminating the associated channels.

8. WEB SERVER DESIGN

The Asbestos Web server is based on the OKWS system for Unix [Krohn 2004]. In the original OKWS design, a demultiplexer, *ok-demux*, accepts each incoming TCP connection and parses its HTTP headers to determine which service the remote client is requesting. It then hands the connection off to a *worker process* specialized to provide that service. Unix OKWS attempts to isolate services so that one compromised service cannot affect others. The Asbestos Web server also isolates services, but additionally isolates user state *within* workers via event processes; a worker, even if compromised, cannot leak one user's information to other users.

8.1 Startup

The Asbestos Web server is started by a launcher process. The launcher spawns *ok-demux*, site-specific workers requested by the site operator, and

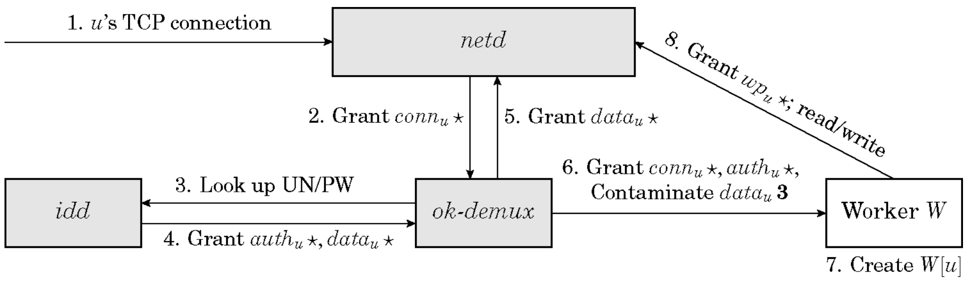
1. u's TCP connection
$netd$
8. Grant $wp_u \star$; read/write
2. Grant $conn_u \star$
5. Grant $data_u \star$
6. Grant $conn_u \star$, $auth_u \star$, Contaminate $data_u$ **3**
idd
3. Look up UN/PW
4. Grant $auth_u \star$, $data_u \star$
ok-$demux$
Worker W
7. Create $W[u]$

Fig. 11. OKWS message flow for handling user u's Web request.

two other processes seen in Figure 1, *idd* and *ok-dbproxy*. The processes exchange and inherit ports to establish the communication paths seen in Figure 11.

The *ok-demux* process must be certain that it is communicating with the worker processes that the launcher started and therefore can't trust workers to identify themselves correctly. Thus, the launcher grants privilege for a process-specific *verification tag* to each process it starts. *Ok-demux* collects these tag values from the launcher. When a worker identifies itself to *ok-demux*, it must provide a **V** label containing its verification tag at level **0**, allowing *ok-demux* to verify that it speaks for the relevant process. Other designs, such as having the launcher mediate initial communication with the workers or providing a verified sender field in the IPC mechanism, could also solve this problem.

In the current system, a process crash would necessitate a restart of the whole process suite, though a more mature version of launcher could restart dead processes (as in OKWS on Unix).

8.2 Basic Connection Handling

We now describe the data path of a simple Web request for the Asbestos Web server; Figure 11 shows the steps. When a user u makes an HTTP connection:

(0) During initialization, *ok-demux* registered with the user-level network server, *netd*, to listen for incoming TCP connections on the machine's Web port.

(1) Once *netd* accepts u's connection, it allocates a new port $conn_u$ to act as a "socket." Processes will send READ and WRITE messages to this port to communicate over the network. The port label, $\mathbf{PC}_{conn_u}$, is set to $\{conn_u\ \mathbf{0}, \mathbf{2}\}$ by the kernel. Section 8.7 describes *netd* further.

(2) *Netd* notifies *ok-demux* of the new connection and uses $\mathbf{T}^- = \{conn_u \star, \mathbf{3}\}$ to grant it the capability to send to the socket.

(3) *Ok-demux* reads network data from u via port $conn_u$ until it can authenticate the user. Currently, the Web server uses a simple username and password pair. *Ok-demux* sends u's username and password to the Web server's identity server, *idd*, which is described in Section 8.4.

(4) If u provided a valid login, *idd* uses $\mathbf{T}^-$ to grant *ok-demux* privilege for two tags corresponding to u, a *secrecy* tag $data_u$ and an *authority* tag $auth_u$, both at level $\star$.
(5) *Ok-demux* grants $data_u\,\star$ to *netd*, which then raises its clearance label to contain $data_u\,\mathbf{3}$ and raises $\mathbf{PC}_{conn_u}$ to $\{conn_u\,\mathbf{0}, data_u\,\mathbf{3}, \mathbf{2}\}$. These changes implement the secrecy idiom with a declassifier: data at $data_u\,\mathbf{3}$ can escape over the network, via $conn_u$. From now on, *netd* will respond to all messages sent to $conn_u$ using $\mathbf{T}^+ = \{data_u\,\mathbf{3}, \star\}$.
(6) When *ok-demux* read TCP payload bytes from *netd* in Step 3, it also noted which service u requested. Assuming worker W provides the service, *ok-demux* grants the $conn_u$ and $auth_u$ capabilities to W—allowing W to send data to u's connection and to act with u's authority—and also raises W's tracking label to $data_u\,\mathbf{3}$ with $\mathbf{T}^+ = \mathbf{C}^+ = \{data_u\,\mathbf{3}, \star\}$, enforcing u's secrecy.
(7) Worker W returns from **ep_checkpoint** into a new event process $W[u]$, receiving $conn_u\,\star$, $auth_u\,\star$, and $data_u\,\mathbf{3}$.
(8) Event process $W[u]$ makes a new port wp_u and grants the capability to send to that port to *netd*. It then reads u's request by sending read requests to *netd*'s port $conn_u$, yielding, and reading *netd*'s replies to wp_u upon wakeup. After reading and parsing u's entire request, the event process formulates a reply and writes it to $conn_u$.
(9) $W[u]$ calls **ep_exit** after completing the request.

We briefly argue that the worker $W[u]$ can communicate with u as intended. $W[u]$'s tracking label is contaminated with $data_u\,\mathbf{3}$ in Step 6, but *netd*'s clearance label was changed in Step 5 to accommodate that contamination. Consequently, the kernel will allow $W[u]$ to send data over $conn_u$ to *netd* and across the network to u.

The protocol is secure because any process or event process that accesses u's data either is trusted and has $data_u\,\star$ in its tracking label, or is not trusted and has $data_u\,\mathbf{3}$. In this example, *netd*, *idd* and *ok-demux* have $data_u\,\star$, but we assume them uncompromised (see Section 8.8 for further discussion). The event process $W[u]$ and its descendants have had the opportunity to see user u's private data, and therefore have $data_u\,\mathbf{3}$ in their tracking labels. All other processes, such as those working on behalf of a different user x, do not have the necessary clearance to receive messages from $W[u]$ or its descendants, preventing them from receiving u's data. Even if such data theft were possible, *netd* would not allow its traffic over the network: any process with $data_u\,\mathbf{3}, data_x\,\mathbf{3}$ in its tracking label cannot send data to *netd* because no *netd* port has the clearance to receive that contamination.

While the kernel enforces security policies that isolate user data flows from one another, the Web server's concept of a user is opaque to the operating system. Having declassification privilege for a Web server user's tag, such as $data_u$, implies nothing about access to sensitive system resources, such as the kernel disk image or the system password file. An Asbestos application like the Web server has no need for "superuser" access.

8.3 Web Sessions

Although HTTP is stateless, many Web servers store *session data* that persists over multiple HTTP connections. The Asbestos Web server can securely store per-user server-side state with simple additions to the above protocol. When supporting sessions, the *ok-demux* process stores a table of all recently active user-worker pairs. In Step 6, if user u requests service from worker W, *ok-demux* looks in this table for a port to the event process $W[u]$. If it finds such a port, it forwards $conn_u$ directly to $W[u]$. If it does not, it forwards u's connection as normal, causing a new event process to be created. When the new event process allocates port wp_u in Step 7, it notifies *ok-demux*, which then inserts the value into its session table for future use.

The worker event process writes session data to memory as normal. To preserve this state across connections, it must call **ep_yield** instead of **ep_exit** in Step 9. Because *ok-demux* sends u's requests for W directly to $W[u]$, those requests will see any previous changes to session state. At the end of the event loop, event processes typically call **ep_clean** before yielding to discard all pages modified since the checkpoint that do not hold session data; this will typically include the stack. When a user u's session times out or u explicitly logs off, u's worker event processes call **ep_exit** and *ok-demux* cleans u's entries out of its session table.

8.4 Managing Identities

In Steps 3 and 4, *ok-demux* verifies user u's login credentials by querying an identity server *idd*. This server associates persistent user identification data, such as username, user ID, and user password, with the more temporary authority and secrecy tags $data_u$ and $auth_u$. When *idd* answers a successful login query in Step 4, it either generates new $data_u$ and $auth_u$ tags (if u has not logged in recently) or returns cached $data_u$ and $auth_u$ tags. In the current implementation, *idd* stores user information in a relational database (see Section 8.5) and never cleans its cache. The identity server has special capability-based access through *ok-dbproxy* to this password database, which other processes such as workers cannot access directly. Thus, the lookup in Step 3 will result in a database query per first-time login.

8.5 Database Interaction

Asbestos offers preliminary database support to worker processes through a port of the Unix package SQLite [SQLite]. A process called *ok-dbproxy* interposes on all Web server database accesses, converting Asbestos labels and security policies to data types and functions native to standard SQLite. With database access, the Web server can easily extend its label-based security policy to one that persists across system reboots. In our current implementation, *ok-dbproxy* is both privileged and trusted: it is trusted to check and set labels to ensure integrity and secrecy respectively, and is privileged in that *idd* grants it all Web server users' secrecy tags at level $\star$.

Out database implementation is intended mostly to make the benchmark and demonstration application more realistic, and does not currently support the full generality of labels. Instead it supports tainting rows as belonging to a specific user, at either level **3** or **1**. *Ok-dbproxy* accomplishes this by adding a "user ID" column to the table definition of every table accessed by Web server workers. The workers themselves cannot access or change this column. When *ok-dbproxy* receives INSERTs, UPDATEs, or other SQL queries that write to the database, it first checks that the request came with a valid username u and a **V** label bounded by $\{data_u\,\mathbf{3}, auth_u\,\mathbf{0}, \mathbf{2}\}$. The **V** label conveys two important facts. First, the sender's tracking label does not contain tags other than $data_u$ at level **3**, and therefore the sender has only been contaminated by her own data. Second, because the **V** label contains $auth_u$ at level **0**, the sender was granted the authority to write data for u. After checking the given **V** label, *ok-dbproxy* queries *idd* to affirm the binding between user u and the two tags $data_u$ and $auth_u$. If all checks pass, *ok-dbproxy* rewrites u's request so that every row written will have u's user ID in the private "user ID" column.

Whenever a worker process fetches data from the database via SELECT, the *ok-dbproxy* process reapplies the appropriate contamination to returned rows. If a row's user ID column contains u's ID, then *ok-dbproxy* returns the row's data using $\mathbf{T}^+ = \{data_u\,\mathbf{3}, \star\}$; it queries *idd* for $data_u$ if it does not have this mapping cached. Each row is returned as a separate message with appropriate $\mathbf{T}^+$ labels, and to finish the request, *ok-dbproxy* sends a message without a $\mathbf{T}^+$ label indicating that all data has been returned. Since each worker's clearance label is limited to receiving at most one user's secrets, a worker will receive only rows meant for its user, and cannot tell how many other rows were sent.

Our current prototype retrofits a standard database with a subset of Asbestos's security features. Database systems built specifically for Asbestos or other distributed information flow control systems would incorporate labels and event processes in a deeper way.

8.6 Decentralized Declassification

When a process, such as user x's worker event process, asks to read user u's data from the database, the database will contaminate the response with $data_u\,\mathbf{3}$. User x's event process will fail to receive this message since its clearance label is not high enough. Even if it were to receive the message, its tracking label would be too high to send data back to x over the connection $conn_x$. But user u may *want* to share some data with x, such as her public profile. That is, user u sometimes needs to declassify private data for public access.

The Asbestos Web server supports semi-trusted decentralized declassifiers for this purpose. To trust a worker as a declassifier, the launcher tells *ok-demux* that a particular worker is a declassifier. Then, when *ok-demux* hands a connection off to the declassifier worker D in Step 6, it grants D the tag $data_u\,\star$ instead of contaminating it with $data_u\,\mathbf{3}$. Privilege for $data_u$ gives the worker the privilege to declassify u's secret data. Thus, when D contacts the database to SELECT u's secret data, *ok-dbproxy*'s response does not affect D's tracking label. The declassifier can now write declassified data to the database, providing a **V** label of $\{\mathbf{1}\}$ to prove it has this right. Internally, *ok-dbproxy* flags a data row as declassified by setting the row's user ID column to zero. When *ok-dbproxy* reads data with zeroed user IDs back out of the database, it does not add an $\mathbf{T}^+$ label, and user x's worker process can safely read u's declassified data out of the database without affecting its tracking label. This declassification is decentralized since it does not directly involve *idd*, the creator of tag $data_u$. Furthermore, the $D[u]$ event process is trusted only by u and cannot declassify any other user's data. An attack on a declassifier worker can expose more of u's data than intended, but cannot otherwise affect the system's information flow.

8.7 Network Server

All access to the network in Asbestos happens through one process, *netd*, which implements the TCP/IP stack (using a port of LWIP [Dunkels 2003]), manages network devices (using a version of Intel's Linux driver for the E1000 gigabit Ethernet card), and creates connections for other processes. As the single interface to the network, *netd* has a privileged role and must properly apply restrictions to connections it manages. An application can send a message to *netd* to request a outgoing connection to a remote host or to listen for incoming connections. In either case, *netd* wraps a new connection with an Asbestos port for which it grants privilege to the requesting application in the manner of the capability idiom. Once a process has a port corresponding to an open connection, it may perform READ and WRITE operations to transfer data, CONTROL operations to close the connection or change the socket's low-water mark, and SELECT operations to determine available buffer space. On a listening socket, a process may perform READ operations to accept incoming connections and CONTROL operations to close the socket. In order to apply labeling to network connections, *netd* optionally maintains a secrecy tag for each connection. When a process tells *netd* to add a secrecy tag to a connection, later messages sent in response to operations on that connection will be contaminated with the secrecy tag at level **3** (a more general implementation would apply a general $\mathbf{T}^+$ label with tags at any level). In the Web server, for example, *netd* contaminates all data read from user u's connection with $data_u$ **3** at *ok-demux*'s behest.

8.8 Trust and Privilege in the Asbestos Web Server

Currently, all Asbestos Web server components are trusted and/or privileged except for worker processes. We claim that for typical Web sites, worker processes correspond to the most vulnerable and error-prone parts of the computing base. They are vulnerable because they read, write, store, and manipulate sensitive data both from the network and from the database. They are error-prone for several software engineering reasons. First, worker code typically does not face external code audit, both because it varies greatly from site to site and because many sites' intellectual property controls discourage open review. Second, load on Web sites can fluctuate wildly: with unexpected load spikes come emergency performance fixes that can accidentally circumvent security mechanisms. Third, large Web sites can run hundreds of thousands of lines of Web code, and since writing Web service code that functions correctly (produces the correct result for honest users) seems simple, it is often assigned to junior programmers without stringent oversight. Experience has shown, however, that writing *secure* Web services is challenging indeed. We believe securing Web applications with automatic, kernel-enforced mechanisms to be a significant step for Web security.

To show that Asbestos can secure worker processes, we built two workers with intentional security flaws. One allows the user to inject arbitrary SQL to run against a database and the other allows the user to start an arbitrary process. Using these workers we logged in as a particular user and then tried to access data belonging to another user, data that should be inaccessible. In all cases, the inappropriate data flow was prevented. By adding messages to the kernel we were able to confirm that the label components that should have prevented the inappropriate data flow were in fact preventing that flow.

More Web server components could be moved out of the trusted or privileged computing base. For instance, *netd* could be decomposed into a simple trusted and privileged component and an event process-based workhorse. The trusted front end would classify incoming packets and firewall outgoing packets based on discretionary label rules; it would therefore be privileged with respect to all Web server user tags $data_u$, as *netd* is now. It would forward packets, once classified, to the appropriate event processes of an untrusted *netd* back end, which would manage the specifics of TCP buffering and flow control. Each back-end event process would be contaminated with respect to the user on whose behalf it speaks, much like worker processes in the current system. Similarly, the database might be decomposed into a trusted, privileged indexer and an event process-based back end that would manage caching and stable storage.

9. COVERT CHANNELS

Asbestos labels prevent processes from explicitly transmitting sensitive information to unauthorized parties. However, supposedly isolated processes can still communicate information through covert channels. Our goal is not to eliminate covert channels—an impossible task—but rather to make it significantly harder to leak information than on systems used as Internet servers today. While high-grade military systems are required to quantify the rates of all covert channels, for Asbestos we content ourselves with enumerating the channels.

Broadly speaking, covert channels can be categorized as either timing or storage channels. A process A conveys information to B with a timing channel if it modulates its use of system resources in a way that observably affects B's response time. For instance, A might flush the processor cache or cause the disk arm to be moved farther from a subsequent request. We are less concerned with timing channels than with storage channels, as to some extent timing channels can be mitigated by limiting processes' ability to measure time precisely [Hu 1991]. Asbestos offers no such feature, however, and the problem becomes harder in the presence of network communication.

Storage channels are caused by any state that can be modified by process A and observed by B when A is not supposed to transmit information to B. As it is difficult if not impossible to eliminate all storage channels, it was a goal to avoid storage channels that could be exploited within a single process, so that at least two cooperating processes are required to communicate information in violation of a label policy.

The Asbestos design contains two inherent storage channels, the program counter and process labels. The **ep_yield** system call potentially affects the program counter of a differently tagged event process by causing it to run. Two colluding, identically labeled event processes can transmit a bit of information by the order in which they call **ep_yield** if the next scheduled event processes have lesser taint. This channel is roughly equivalent to the covert channel intentionally included by the drop-on-exec feature of IX [McIlroy and Reeds 1992].

The **send** system call potentially raises the value of the recipient's tracking label to an unanticipated value. This is also a storage channel, as labels can be observed through lack of communication. Consider a contaminated process A attempting to communicate a bit of sensitive information to an uncontaminated process C. An attacker might construct two uncontaminated processes (either before A becomes contaminated or by controlling some other uncontaminated process). These processes, B_0 and B_1, both repeatedly send heartbeat messages to C. By sending a message that contaminates process B_i, A can communicate the value i to C. Such storage channels are inherent to any system in which a contaminated process can change the labels of a less contaminated process. HiStar [Zeldovich et al. 2006], a successor system to Asbestos, avoids these channels by requiring that processes explicitly choose to contaminate themselves: a HiStar thread actively changes its label before reading contaminated data. A similar technique could eliminate the analogous channel in Asbestos's message passing architecture.

Other Asbestos kernel data structures have been carefully designed to avoid exploitable storage channels. Tags are generated by incrementing a 61-bit counter, which would be a storage channel. However, since the kernel encrypts the counter value to produce tags, the user-visible sequence of tags does not convey exploitable information.

The current implementation still has several other storage channels that could be closed or limited. In particular, Asbestos does not yet deal gracefully with resource exhaustion.

10. EVALUATION

Asbestos's information flow and isolation features add moderate overhead to Asbestos applications. The additional memory per user required to support user isolation on our Asbestos Web server is small. The server's throughput and latency are competitive with those of the well-known and well-tuned Apache server running on Linux. For each metric, the Asbestos Web server's performance lies between the performance of an Apache server with CGI-based workers and an Apache server with worker code linked in as an integrated module.

Performance measurements were conducted on a gigabit Ethernet-based local network with a Linux HTTP client generating requests. The Asbestos server is a 2.8 GHz Pentium 4 with 1 GB of RAM. Our experiments made as few file

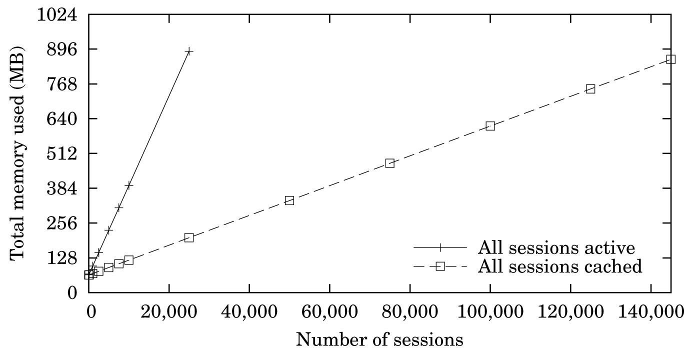

Fig. 12. Memory used by active and cached Web sessions as a function of the number of sessions. Includes all memory allocated by both kernel and user programs.

system accesses as possible; we disabled all Web access logging and ran all databases in memory.

10.1 Memory Use

In Section 7, we argued the merits of event processes over more traditional fork-accept designs. Our hypothesis was that each additional protection domain might consume only one additional page of memory. Our measurements roughly support this claim.

Web sites often cache dynamic data to lighten database load and to avoid latency. As discussed in Section 8.3, the Asbestos Web server uses event processes to cache dynamic data while maintaining isolation among users. Our test program uses event processes that persist for the lifetime of a Web session, and thus span multiple HTTP connections At the end of an HTTP connection, the worker uses **ep_clean** to release all allocated memory except for session data. A cleaned event process holding only session data is called a *cached session*. An event process that is processing an HTTP request uses more memory than a cached session, since it stores temporary variables and buffers; such an event process is called an *active session*. A typical Web server has many more cached sessions than active ones.

Our experiments measured the amount of allocated memory after creating different numbers of Web sessions, including space for kernel data structures. In all of our memory measurements, we ran the server with one toy Web service that stores data from a user's HTTP request and returns it to the user in the subsequent request. The size of the response is about 1 KB.

The system uses approximately 1.4 four-kilobyte pages per cached session; Figure 12 graphs the results. One complete page is due to state maintained in the worker's event process. The remainder of the memory is for kernel data structures—event processes, labels, and tags—as well as memory in other user processes, such as *idd*.

To determine the memory cost of active sessions, we repeated the previous experiment but modified the worker so that it never unmaps memory via **ep_clean** or **ep_exit**. This method produces worst-case behavior, capturing the maximum amount of memory consumed by our simple worker. The experiment shows that an additional seven pages of memory are used by each active session. Two of those pages are stack and exception stack pages, one is for the event process's message queue, and the remaining four comprise the modified heap and pages with modified global variables.

10.2 Web Server Performance

Our throughput and latency experiments tested an even simpler Web application: a chargen service that responds with a string of characters whose length depends on the client's parameters. We compared the Asbestos Web server to the Apache Web server, version 1.3.33 [Apache] (which outperformed version 2.0.54 in our tests). We implemented our test application both as a standard CGI process written in C and as an Apache module written in C [Apache API]. In both cases, Apache keeps a pool of pre-forked processes to answer requests. Apache with CGI processes additionally forks and executes the CGI binary for each request. Apache with the module version of the service, which we call "Mod-Apache," does not fork for each request. Mod-Apache is efficient but provides no isolation. Apache with CGI processes does provide some isolation, but at a significant cost when compared to Mod-Apache, since each request is handled in a forked process. However, at least in its default configuration, Apache does not run CGI processes in a chroot jail, so a malicious user that takes over an exploitable CGI process can easily attack any other local vulnerabilities accessible to the corresponding Unix user. In contrast, the Asbestos server provides isolation both between services and between users within a service.

Throughput. To test throughput for the Asbestos server relative to Apache and Mod-Apache, we first varied concurrency to maximize completed connections per second. For Apache, 400 concurrent connections maximized throughput; for Mod-Apache 16 concurrent connections was the sweet spot. Asbestos's network stack is based on LWIP [Dunkels 2003], which was chiefly designed to conserve resources and is not tuned for performance under load. Sixteen concurrent connections gave maximum throughput. For the Asbestos server, we then varied the number of cached sessions in the system. In all tests, the server responded with 144 bytes of HTTP data, 133 bytes of which were in headers. Larger responses only exercise the network stack.

The Asbestos server's isolated users were authenticated and workers were run in different event processes as usual. We measured performance with the session support described in Section 8.3: once authenticated to the system, future requests were serviced by the event process created in the authentication step. The Asbestos throughput results thus contain data both for forwarding messages to existing event processes and for creating new event processes, which is slower. (Creating a new event process involves communication with the database and some kernel overhead.) In our benchmark, for 1000 user sessions and more, each user connected to its session exactly four times; a workload with a different ratio of new sessions to existing sessions would perform differently. Because the number of sessions affects the size of labels on some components, we expect performance to change with the number of cached sessions. Neither

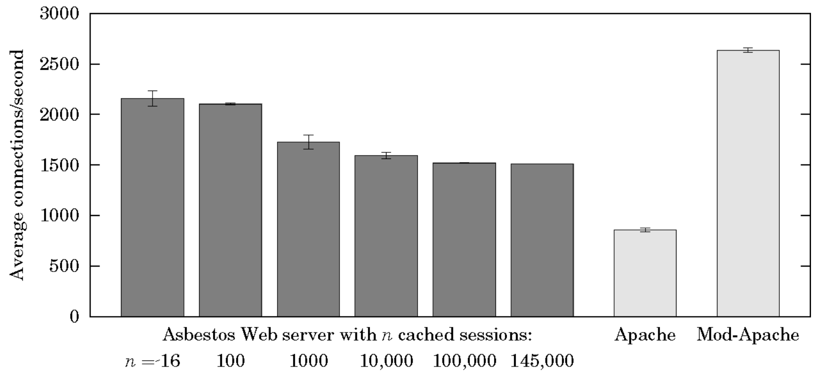

Fig. 13. Throughput for various numbers of cached sessions in the Asbestos Web server, compared with Apache and Mod-Apache.

	Latency (μs)	
Server	**Median**	**90th percentile**
Mod-Apache	999	1,015
Apache	3,374	5,262
Asbestos, 16 sessions	1,917	2,382
Asbestos, 145,000 sessions	2,542	3,082

Fig. 14. The median and 90th percentile latencies of requests to various server configurations.

Apache nor Mod-Apache isolates users, so no attempt to authenticate users is made for those tests.

Figure 13 shows that, with sixteen sessions, the Asbestos server performs about 250% better than Apache, or about eighty percent as well as Mod-Apache. However, for most of the tests, the Asbestos server performs around twice as well as Apache and half as well as Mod-Apache. Section 10.3 further discusses the factors that reduce the Asbestos server's performance as the number of sessions increase.

Latency. This section compares the per-request latency of the Asbestos Web server with Apache and Mod-Apache. We measured the latency of all three servers with a concurrency of only four simultaneous connections; larger concurrency values pressure Asbestos's LWIP stack. Mod-Apache, which processes each request within a single process, responds to most requests with very low latency. This is to be expected of a server that can handle Web requests with simple library calls. Unlike Mod-Apache, Apache with CGI pays performance penalties for forking and IPC, responding to most requests with three to five times the latency. As shown in Figure 14, the Asbestos server with sixteen users has a smaller median latency than Apache, as well as a smaller 90th percentile. Scheduling affects Asbestos to a lesser extent because there is no parallelism for requests to choose from. All requests must sequentially traverse *netd*, *okdemux*, worker, and then *netd* again, which does not give the option for any request to be temporarily starved. Asbestos with 145,000 cached sessions has latencies which are a bit more than sixteen users.

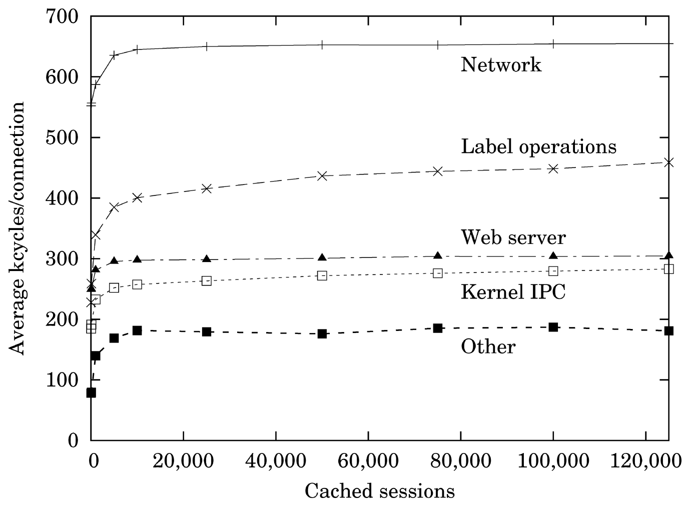

Fig. 15. The average cost of various Asbestos components in kcycles/connection as the number of cached sessions increases.

10.3 Label Costs

Ideally, varying the number of sessions should have no effect on throughput or latency. However, the size of various labels in the system will increase with the number of sessions. Figure 15 shows the costs of various components in the system in thousands of CPU cycles per connection as the number of cached sessions increases. The "Network" and "Web server" lines represent the time spent in *netd* and the server code itself, respectively. The "Kernel IPC" line includes time spent in processing the **send** and **recv** system calls, minus time spent dealing with label operations. All time spent doing label operations anywhere in the system is accounted for in the "Label operations" line. Finally, all other time, including time spent in the database and time dealing with page faults, is included in the "Other" line.

Most of the components use the same amount of time regardless of the number of sessions. Networking consumes the largest fraction of time. Label operations, on the other hand, use more time per connection as the number of sessions increases. Since the Asbestos Web server uses two tags to isolate a user, 145,000 cached sessions implies *idd* and *ok-dbproxy*'s tracking labels will contain more than 290,000 tags, the vast majority at level $\star$; *netd*'s clearance label will have accumulated 145,000 tags in order to accept secrecy tags; and *ok-demux* will hold at least 145,000 tags for open worker sessions. Furthermore, some of these large labels must be updated to include a capability for each new TCP connection, and then to release that capability when the connection is passed to an event process or closed. The AVL tree structure currently used for labels supports these frequent label operations with only logarithmic overhead. As a result, Asbestos labels and event processes can practically isolate user state even on a server storing data for thousands of users.

11. CONCLUSION

The Asbestos operating system makes nondiscretionary access control mechanisms available to unprivileged users, giving them fine-grained, end-to-end control over the dissemination of information. Asbestos provides protection through a new labeling scheme that, unlike schemes in previous operating systems, allows data to be declassified by individual users within categories they control. The categories, called tags, use the same namespace as communication endpoints, making them a kind of generalization of capabilities. As in a capability system, processes can dynamically generate new tags and distribute them independently. Processes can specify temporary label restrictions on sent messages to avoid the unintentional use of privilege.

Asbestos's new process abstraction, the event process, efficiently supports server processes that must spin off many versions inhabiting distinct security compartments. Event processes impose less overhead on the operating system than forked address spaces, so many thousands of them can coexist without resource strain. A prototype Web server manipulates labeled data so that even software bugs in the high-risk worker code cannot cause one user to receive another's private data. The system requires only 1.4 pages of memory per cached Web session and exhibits performance comparable to Unix systems that provide weaker isolation.

ACKNOWLEDGMENTS

The authors thank the following people for their comments and technical contributions: Lee Badger, Emin Gün Sirer, and the anonymous reviewers; Mike Mammarella and Chris Frost for network stack integration; Michelle Osborne for work on an earlier version of the system; and the contributors to the LWIP project, including Adam Dunkels and Leon Woestenberg.

REFERENCES

APACHE. The Apache HTTP Server Project. `http://httpd.apache.org`.

APACHE API NOTES. Apache API module notes: `http://httpd.apache.org/docs/1.3/misc/API.html`.

BELL, D. E. AND LA PADULA, L. 1976. Secure computer system: Unified exposition and Multics interpretation. Tech. Rep. MTR-2997, Rev. 1, MITRE Corp., Bedford, MA.

BERSTIS, V. 1980. Security and protection of data in the IBM System/38. In *Proceedings of the 7th Annual Symposium on Computer Architecture (ISCA)*. 245–252.

BRANSTAD, M., TAJALLI, H., MAYER, F., AND DALVA, D. 1989. Access mediation in a message passing kernel. In *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland, CA, 66–72.

CHERITON, D. R. 1988. The V distributed system. *J. ACM 31*, 3, 314–33.

DENNING, D. E. 1976. A lattice model of secure information flow. *Commun. ACM 19*, 5, 236–243.

DENNING, D. E. AND DENNING, P. J. 1977. Certification of programs for secure information flow. *Commun. ACM 20*, 7, 504–513.

DEPARTMENT OF DEFENSE. 1985. *Trusted Computer System Evaluation Criteria (Orange Book)*. Department of Defense. DoD 5200.28-STD.

DUNKELS, A. 2003. Full TCP/IP for 8-bit architectures. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MOBISYS)*. San Francisco, CA.

EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. 2005. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. Brighton, England.

FRASER, T. 2000. LOMAC: Low water-mark integrity protection for COTS environments. In *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland, CA, 230–245.

GOLDBERG, R. P. 1973. Architecture of virtual machines. In *Proceedings of the AFIPS National Computer Conference*. Vol. 42. 309–318.

HARDY, N. 1988. The confused deputy (or why capabilities might have been invented). *Operat. Syst. Rev. 22*, 4, 36–38.

HU, W.-M. 1991. Reducing timing channels with fuzzy time. In *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland, CA, 8–20.

JAEGER, T., PRAKASH, A., LIEDTKE, J., AND ISLAM, N. 1999. Flexible control of downloaded executable content. *ACM Trans. Inform. Syst. Secur. 2*, 2, 177–228.

KARGER, P. A. 1987. Limiting the damage potential of discretionary Trojan horses. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 32–37.

KARGER, P. A. AND HERBERT, A. J. 1984. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland, CA, 2–12.

KARGER, P. A., ZURKO, M. E., BONIN, D. W., MASON, A. H., AND KAHN, C. E. 1990. A VMM security kernel for the VAX architecture. In *Proceedings of the IEEE Symposium on Security and Privacy.* Oakland, CA, 2–19.

KEY LOGIC. 1989. The KeyKOS/KeySAFE system design. Key Logic. Tech. Rep. SEC009-01. `http://www.cis.upenn.edu/~KeyKOS/`.

KING, S. T. AND CHEN, P. M. 2003. Operating system support for virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, San Antonio, TX.

KROHN, M. 2004. Building secure high-performance web services with OKWS. In *Proceedings of the USENIX Annual Technical Conference*. Boston, MA, 185–198.

KROHN, M., EFSTATHOPOULOS, P., FREY, C., KAASHOEK, F., KOHLER, E., MAZIÈRES, D., MORRIS, R., OSBORNE, M., VANDEBOGART, S., AND ZIEGLER, D. 2005. Make least privilege a right (not a privilege). In *Proceedings of the 10th Hot Topics in Operating Systems Symposium (HotOS-X)*. Santa Fe, NM.

LANDWEHR, C. E. 1981. Formal models for computer security. *ACM Comput. Surv. 13*, 3 (Sept.), 247–278.

LEMOS, R. 2005. News.com: Payroll site closes on security worries, Feb. 25, 2005. `http://news.com.com/2102-1029_3-5587859.html`.

LIEDTKE, J. 1995. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain Resort, CO.

LOSCOCCO, P. AND SMALLEY, S. 2001. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the USENIX Annual Technical Conference—FREENIX Track*. 29–40.

MACMILLAN, K., BRINDLE, J., MAYER, F., CAPLAN, D., AND TANG, J. 2006. Design and implementation of the SELinux policy management server. In *Proceedings of the Security Enhanced Linux Symposium*. Baltimore, MD.

MCCOLLUM, C. J., MESSING, J. R., AND NOTARGIACOMO, L. 1990. Beyond the pale of MAC and DAC—defining new forms of access control. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 190–200.

MCILROY, M. D. AND REEDS, J. A. 1992. Multilevel security in the UNIX tradition. *Softw.—Pract. Exper. 22*, 8, 673–694.

MITCHELL, J. G., GIBBONS, J., HAMILTON, G., KESSLER, P. B., KHALIDI, Y. Y. A., KOUGIOURIS, P., MADANY, P., NELSON, M. N., POWELL, M. L., AND RADIA, S. R. 1994. An overview of the Spring system. In *Proceedings of COMPCON 1994*. 122–131.

MYERS, A. C. AND LISKOV, B. 2000. Protecting privacy using the decentralized label model. *ACM Trans. Comput. Syst. 9*, 4 Oct., 410–442.

NEWS10. 2005. Hacker accesses thousands of personal data files at CSU Chico, March 17, 2005. `http://www.news10.net/display_story.aspx?storyid=9784`.

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999. Flash: An efficient and portable Web server. In *Proceedings of the USENIX Annual Technical Conference*. Monterey, CA, 199–212.

PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. 1995. Plan 9 from Bell Labs. *Comput. Syst. 8*, 3, 221–254.

RASHID, R. F. AND ROBERTSON, G. G. 1981. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*. Pacific Grove, CA, 64–75.

ROZIER, M., ABROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMANN, F., KAISER, C., LANGLOIS, S., LEONARD, P., AND NEUHAUSER, W. 1988. CHORUS distributed operating system. *Comput. Syst. 1*, 305–370.

SALTZER, J. H. AND SCHROEDER, M. D. 1975. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9, 1278–1308.

SCHNEIER, B. 1993. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *Proceedings of Fast Software Encryption, Cambridge Security Workshop*. Springer-Verlag, 191–204.

SHAPIRO, J. S. AND HARDY, N. 2002. EROS: A principle-driven operating system from the ground up. *IEEE Softw. 19*, 1, 26–33.

SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. 1999. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. Kiawah Island, SC, 170–185.

SQLITE. `http://www.sqlite.org`. Version 3.2.1.

TANENBAUM, A. S., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G. J., MULLENDER, S. J., JANSEN, J., AND VAN ROSSUM, G. 1990. Experiences with the Amoeba distributed operating system. *Commun. ACM 33*, 12, 46–63.

TROUNSON, R. 2006. Major breach of UCLA's computer files. *Los Angeles Times*, Dec. 12, 2006. `http://www.latimes.com/news/local/la-me-ucla12dec12,0,7111141.story`.

VMWARE. 2000. VMware and the National Security Agency team to build advanced secure computer systems. *Tech Trend Notes 9*, 4, 3–11. `http://www.vmware.com/pdf/TechTrendNotes.pdf`.

VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. 2003. Capriccio: Scalable threads for Internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, Lake George, NY, 268–281.

WATSON, R., MORRISON, W., VANCE, C., AND FELDMAN, B. 2003. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proceedings of the USENIX Annual Technical Conference*, San Antonio, TX, 285–296.

WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. 2002. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. Boston, MA, 195–210.

ZELDOVICH, N. B., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. 2006. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. Seattle, WA.

Received April 2007; revised October 2007; accepted October 2007