Labels and Event Processes in the Asbestos Operating System

Petros Efstathopoulos* Maxwell Krohn† Steve VanDeBogart*
Cliff Frey† David Ziegler† Eddie Kohler* David Mazières‡ Frans Kaashoek† Robert Morris†
*UCLA †MIT ‡Stanford/NYU
http://asbestos.cs.ucla.edu/

ABSTRACT

Asbestos, a new prototype 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 label mechanism, including controls on inter-process communication and system-wide information flow. A new event process abstraction provides lightweight, isolated contexts within a single process, allowing the same process to act on behalf of multiple users while preventing it from leaking any single user's data to any other user. A Web server that uses Asbestos labels to isolate user data requires about 1.5 memory pages per user, 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

Keywords: labels, mandatory access control, information flow, event processes, secure Web servers

1 INTRODUCTION

Breaches of Web servers and other networked systems routinely divulge private information on a massive scale [23, 32]. The kinds of exploitable software flaws that enable these breaches will persist, but all is not lost if we design systems that limit the possible impact of most exploits. A powerful tool to contain exploits is the principle of least privilege [37], which directs that each system component should have the minimum privilege required to accomplish its task. A corresponding policy would prevent a server acting for one principal from accessing data belonging to another principal through direct or indirect channels. A least privilege policy, enforced by the operating system at the behest of a small, trusted part of the application, would defang classes of exploits from SQL injection to buffer overruns, making servers much safer in practice.

Unfortunately, current operating systems cannot enforce least privilege. Even the much weaker goal of isolating services from one another (without isolating principal state inside each service) requires fiddly and error-prone abuse of primitives designed for other purposes [20]. Most servers instead revert to the most insecure design, monolithic code running with many privileges. It should come as no surprise that high-impact breaches continue.

New operating system primitives are needed [21], and the best place to explore candidates is the unconstrained context of a new OS. Hence the Asbestos operating system, which can enforce strict application-defined security policies even on efficient, unprivileged servers.

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, they can be used to enumerate positive rights, such as the right to send to the network. Unlike traditional capability systems, however, Asbestos labels can also track and limit the flow of information within system- and application-defined compartments. These complementary security models are linked by a key observation: the ability to declassify data in a single compartment is analogous to possession of a discretionary capability. The resulting system supports capability-like and traditional MLS [9] policies, as well as application-specific isolation policies with *decentralized declassification*, 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 data belonging to multiple users and lose the ability to respond to anyone. One fix is a forked server model, in which each active user has a forked copy of the server process; unfortunately, this resource-heavy architecture burdens the OS with many thousands of processes that need memory allocated and CPU time scheduled. Event processes allow a single process to keep private state for multiple users, but isolate that state so that an exploit affects only one user's data. A group of event processes is almost as efficient as a single ordinary process. The event process discipline encourages efficient server construction, and in our experiments, servers can cache thousands of user sessions with low storage costs.

Measurements on an x86 PC show that an Asbestos Web server can support comprehensive user isolation at a cost of about 1.5 memory pages per user. Furthermore, although our prototype label implementation impacts performance, an Asbestos Web server storing isolated data for thousands of users is in some ways 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.

2 APPLICATION GOAL

We evaluated Asbestos by implementing a secure application that we could not build on current systems, namely a dynamic-content Web server that isolates user data. Our goal, in a nutshell:

SOSP'05, October 23–26, 2005, Brighton, United Kingdom.

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.

The rest of this section expands and clarifies this goal. Although the goal refers to server applications, Asbestos mechanisms should aid in the construction of other types of software. For example, email readers could use related 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 additionally want to *isolate* different users' data from one another, so that any security breaches are contained. The Asbestos event process mechanism aims to satisfy this requirement.

By *unprivileged*, we mean that the system administrator has granted the application the minimum privilege required to complete its job, and this minimum privilege is much less than all privilege. Thus, the system follows the principle of least privilege.

Users are *application-defined*, meaning each application can define its own notion of principal and its own set of principals. One application's users can be distinct from another's, or the user populations can 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 *sharing* among users for data that need not be isolated. 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 can 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 even totally compromised processes from violating the policy; for example, they should be unable to launder data through non-compromised services and applications. Thus, isolation policies can restrict *information flow* among processes that may be ignorant of the policies. Unfortunately, any system that controls information flow through run-time checks can inappropriately divulge information when those checks fail [31]; in effect, kernel data structures for tracking information flow provide a covert storage channel. We aim to eliminate storage channels that can be exploited without multiple processes, so that a later, hardened version of Asbestos can improve security by limiting process creation rates. Section 8 discusses this issue in depth.

In summary, Asbestos must support a form of *mandatory access control*, which transitively isolates processes by tracking and limiting the flow of information. Unprivileged applications define their own isolation policies and decide what information need not be isolated. Furthermore, OS mechanisms for labeling processes must support highly concurrent server applications.

These Asbestos ideas achieve full expression in the design and implementation of the Asbestos OK Web server, a much improved version of the original OKWS for Unix [20]. The server implements a Web site with multiple dynamic *workers*—one each for logging in, retrieving data, and changing a password, for example. Each worker is its own process; the *ok-demux* process analyzes incoming connection requests and forwards them to the relevant worker. Each worker 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 SQL database access (table rows are labeled as belonging to particular users) and declassifiers (selected workers that can export user data to the public). The workers are untrusted, meaning that worker compromise cannot violate the user isolation policy. Trusted components 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. Declassifier workers are semi-trusted: 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.

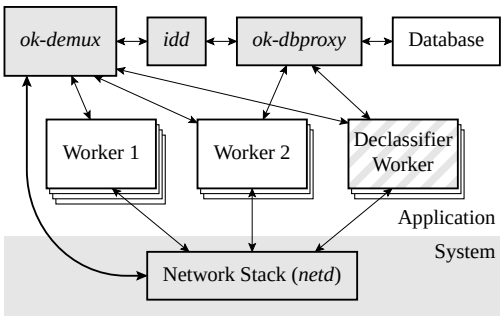

Figure 1: Processes of the Asbestos OK Web server. Grey boxes are trusted. Worker processes contain one event process per user session.

3 Related Work

Mandatory access control (MAC) systems provide end-to-end enforcement of security policies by transitively following causal links between processes. Operating systems have long expressed and enforced these policies using *labels* [9]. Labels assign each subject and object a security level, which traditionally consists of a hierarchical sensitivity classification (such as *unclassified*, *secret*, *top-secret*) and 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 [25] and FreeBSD [44].

MAC systems generally aspire to achieve some variant of the $*$*-property* [3]: 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* rather than mandatorily enforcing it. Of course, real operating systems do provide some way to declassify or "downgrade" data—for example, as a special privilege afforded certain users if they press the secure attention key [17]—but this lies outside the main security model.

Most MAC systems are geared towards military settings, which require labels to specify at least 16 hierarchical sensitivity classifications and 64 categories [9]. This label format determines 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 users and groups might enforce access control within the secret, nuclear level.

Asbestos labels differ significantly from those of previous operating systems in that Asbestos lets any process dynamically create label categories, or *compartments*. Moreover, a process can partially bypass the $*$-property by declassifying information or raising the security clearance of other processes—but only with respect to certain compartments, such as the ones it creates. Asbestos tracks information flow by dynamically adjusting labels, but a new *event process* abstraction lets a single, unprivileged process separately handle data from multiple compartments without accumulating restrictions. As described later, the Asbestos system call interface provides a number of other novel features that facilitate the use of labels, including temporary voluntary restrictions and split send/receive labels with different defaults.

The idea of dynamically adjusting labels to track potential information flow dates back to the High-Water-Mark security model [22] of the ADEPT-50 in the late 1960s. Numerous systems have incorporated such mechanisms, including IX [28] and LOMAC [10]. The ORAC model [27] supported the idea of individual originators placing accumulating restrictions on data, somewhat like creating compartments, except that data can still only be declassified by users with the privileged Downgrader role.

Asbestos labels more closely resemble language-level flow control mechanisms. Jif [31], in particular, was an inspiration for Asbestos because of its support for decentralized declassification through separate ownership of different label components. Because it is a programming language, Jif has the advantage of being able to perform most of its label checks statically, at compile time. Runtime checks can affect control flow on failure, thereby creating implicit information flows [8]. However, compared to Asbestos, Jif requires a centralized principal hierarchy and has no equivalent to split label defaults, which Asbestos uses to support policies such as preventing one process from talking to another.

Asbestos uses communication ports similar to those of previous message-passing operating systems [6, 24, 30, 35, 36, 41], some of which can confine executable content [14], others of which have had full-fledged mandatory access control implementations [5]. Asbestos uses the same namespace—*handles*—for both ports and compartments, allowing labels to emulate a wide range of security mechanisms from discretionary capabilities to multi-level security.

In theory, capabilities alone suffice to implement mandatory access control. For instance, KeyKOS [18] achieved military-grade security by isolating processes into compartments and interposing reference monitors to control use of capabilities across compartment boundaries. EROS [39] later successfully realized the principles behind KeyKOS on modern hardware. Psychologically, however, people have not accepted pure capability-based confinement [29], perhaps from fear that if just one inappropriate capability escapes, the security of the whole system may be compromised. As a result, a number of designs have combined capabilities with authority checks [4], interposition [15], or even labels [16].

Mandatory access control can also be achieved with unmodified traditional operating systems through virtual machines [11, 17]. For example, the NetTop project [42] uses VMware for multi-level security. Virtual machines have two principal limitations, however: performance [19, 46] 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.

4 Asbestos Overview

Asbestos IPC resembles that of microkernels such as Mach. Processes communicate using messages sent to *ports*. 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 below.

Asbestos messaging is asynchronous and, 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 the process's labels can change to prevent delivery—or to allow it. More seriously, given reliable delivery notification, a process could 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 messages to be dropped, careful compartment 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 [34].

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. Therefore, communication is generally bootstrapped using environment variables that specify the port names services are currently using.

Asbestos contains system calls for allocating, remapping, and freeing memory at particular virtual addresses, for creating and destroying processes, for creating and dissociating ports, for sending and receiving messages, for bootstrapping, and for debugging, in addition to calls supporting label and event process functionality.

5 Asbestos Labels

Asbestos labels support *decentralized compartments* that any process can dynamically create and manipulate. In order to allow nonprivileged programs to craft their own MAC security schemes, Asbestos labels combine both mandatory and discretionary access controls. Asbestos gives a program that creates a new compartment a *discretionary* right to declassify data in that compartment: the program can give that right away, making the right similar to a capability. The program will typically launch other processes, restricting their labels so that they can reveal data only to processes in the compartment. It may also give the right to declassify to programs trusted to sanitize data; these programs can then release tainted data outside the compartment. Programs can use the same discretionary rights to establish identity and integrity, and to protect the right to send messages to a port—that is, to implement a send capability.

Three features of the Asbestos label design are particularly important for decentralized compartments. First, a special sensitivity level, $\star$, represents declassification privilege with respect to a compartment. Second, when sending a message, a process can supply additionally restrictive *discretionary labels* on top of the process

labels maintained by the kernel; some of these labels are transmitted to the receiving application for possible analysis. Finally, Asbestos processes have separate send and receive labels with *different defaults* for future compartments, allowing policies that transitively prevent two processes from communicating without unduly restricting either process's ability to communicate with the rest of the system.

5.1 Label basics

In general, information flow labels form a *lattice*, a partial order in which any finite set of labels has unique least upper and greatest lower bounds [7]. The partial order $\sqsubseteq$ determines whether one label is dominated by another. The least-upper-bound operator $\sqcup$ is used to combine security classes—when a process reads objects with different classes, for instance. The greatest-lower-bound operator $\sqcap$, unusual in other label systems, is used in Asbestos for declassification.

In Asbestos, each process P has two labels, a *send label* P_S and a *receive label* P_R (somewhat analogous to IX's current and maximum labels). The send label represents the process's current contamination, the receive label the maximum contamination it is able to accept from others. To first order, P may send to Q if

$$P_S \sqsubseteq Q_R, \tag{1}$$

which means that Q is able to receive messages from processes at P's current contamination level, and also that Q is willing to accept contamination at P's level. When the message is delivered, Q's send label is contaminated by P's send label, since information flows from P to Q. Again to first order,

$$Q_S \leftarrow Q_S \sqcup P_S, \tag{2}$$

the least upper bound on the two send labels.

Asbestos compartments are named by *handles*, which are 61-bit numbers. Any process can create a compartment with the **new_handle** system call, which returns a previously-unused handle and, as explained below, grants the calling process privilege for that handle. Handle values are unique since boot time. Thus, unlike a file descriptor value, a given handle value refers to the same handle in all contexts. The 61-bit namespace is large enough that allocating handles at a rate of 1 billion per second would require 73 years to exhaust all values. The kernel generates handles by encrypting a counter with a 61-bit block cipher (derived from Blowfish [38]), resulting in an unpredictable but non-repeating sequence of values; the unpredictability closes certain covert channels by concealing the number of handles that have been created at any given time. However, handles are not in any way self-authenticating [41]—simply knowing a handle's value confers no additional privilege.

Handle privileges are represented by *levels*, which are members of the ordered set $[\star, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}]$; in send labels, $\star$ is the lowest or most privileged level, and **3** is the highest or least privileged level. The default levels lie in between; they are **1** for send labels and **2** for receive labels. The reasons for this difference are explained below.

A label, then, is just a function from handles to levels. We write them as functions, and also using set notation, such as $\{h_1\ \mathbf{0}, h_2\ \mathbf{1}, \mathbf{2}\}$; the default level, which appears without a handle at the end of the list, applies to all handles not mentioned explicitly. To compare two labels, we compare each of their components:

$$L_1 \sqsubseteq L_2 \ \text{ iff } \ L_1(h) \leq L_2(h) \ \text{ for all } h.$$

With this ordering, the least-upper-bound and greatest-lower-bound operators, $\sqcup$ and $\sqcap$, are $(L_1 \sqcup L_2)(h) = \max(L_1(h), L_2(h))$ and $(L_1 \sqcap L_2)(h) = \min(L_1(h), L_2(h))$.

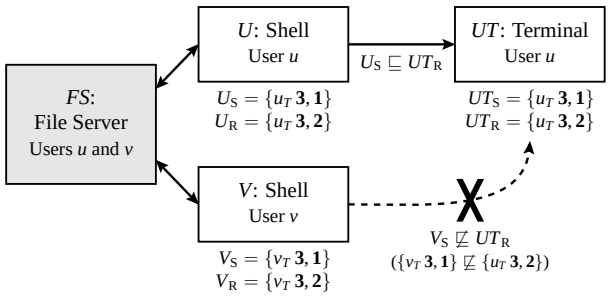

Figure 2: Simplified process communication with labels. The file server is trusted.

5.2 Privacy

We now examine how Asbestos labels can provide privacy through information flow control, using a simple four-process example: a trusted multi-user file server, user shells for users u and v, and a terminal to which user u is logged in. The system's goal is to allow user u's information to pass freely over the terminal while preventing other users' information from escaping. We first assume that process labels are assigned out of band; the next section shows how they are assigned in a decentralized fashion.

Each user needs a security compartment, so we assign each user u a *taint handle* u_T. The next step is to differentiate processes that have seen u's private data from those that have not. We will use send labels for this purpose, since they track the flow of messages by raising receivers' levels with $\sqcup$. We mark the send label of any process that sees u's private data by setting u_T's level higher then the default of **1**. If we choose level **3** for user taints, a process with $P_S(u_T) = \mathbf{1}$ (the default send level) hasn't seen u's data, while a process with $P_S(u_T) = \mathbf{3}$ has.

Now for receive labels. By default, processes have $P_R(u_T) = \mathbf{2}$. This is below the user taint level of **3**, so a process's receive label must be explicitly raised, to $u_T\ \mathbf{3}$, to allow it to receive u's data. Raising receive labels makes the system more permissive, so in Asbestos, it requires special privilege: processes are not free to raise their receive labels arbitrarily.

Figure 2 shows the resulting system. The shell processes U and V are tainted with u_T and v_T (that is, $U_S(u_T) = \mathbf{3}$ and $V_R(v_T) = \mathbf{3}$), and their receive labels allow them to receive the data of their respective users. Any processes they create or communicate with will have the same characteristics. User u's terminal, UT, has the same labels as U. U can send messages to UT, since $U_S \sqsubseteq UT_R$, but V cannot, since $V_S(v_T) > UT_R(v_T)$, and neither can any other process that has seen v's data.

Discretionary contamination Consider the file server FS in Figure 2. To maintain the system's information flow properties, the file server must label files: a process that reads user u's file must become tainted with $u_T\ \mathbf{3}$. (We worry about writes later.) The file server must be able to taint different users' processes in different ways, so it cannot simply use Equation 2 to taint processes.

In Asbestos, the file server can *selectively* taint messages with the appropriate handle by providing an optional *contamination label* C_S when sending a message. This label raises the sender's send label to a new *effective send label* $E_S = P_S \sqcup C_S$. The effective label, not the true send label, is used to check information flow and to contaminate the receiver's send label. Equations (1) and (2) thus become

$$E_S \sqsubseteq Q_R \quad \text{and} \tag{3}$$

$$Q_S \leftarrow Q_S \sqcup E_S. \tag{4}$$

Since contamination only *restricts* information flow, it requires no special privilege; processes can arbitrarily contaminate the messages they send. The default contamination label is $\{\star\}$, which, as the lowest possible label, adds no additional contamination.

This is our first example of an optional, or discretionary, use of the label system. The idea is simple: when processes can control their interactions with the label system—in ways that don't violate basic information flow properties, of course—the label system can implement more security policies, potentially including all access control interactions needed in an operating system.

The four levels The label assignment above prevents *v*'s data from reaching any processes except for an explicitly initialized set, those with $P_{\mathrm{R}}(v_T) = \mathbf{3}$. But Asbestos's four levels **0**–**3**, and its different defaults for send and receive labels, allow other policies as well. Say, for example, we represent user taint by $u_T\,\mathbf{2}$, rather than $u_T\,\mathbf{3}$. Then *U* and *V* could communicate with each other, as well as with other processes in the system; and we could still prevent privacy violations via *UT*, by *lowering* its receive label, to $\{v_T\,\mathbf{1}, \mathbf{2}\}$. *UT* could communicate with *U*, but still not with *V*, since $V_{\mathrm{S}}(v_T) = \mathbf{2}$; and if *U* received a message from *V*, its send label would rise to $v_T\,\mathbf{2}$, preventing further communication with *UT*.

Thus, when user taint uses level **3**, the system defaults to denying user-tainted messages, and the compartment manager must explicitly raise the receive level of each process allowed to receive user data. When user taint uses level **2**, the system defaults to allowing communication, and those processes that *shouldn't* receive user data must have their receive labels explicitly *lowered*. (An application of the latter might be allowing anyone to read a file so long as they don't send the contents to the network daemon.) Different send and receive defaults make it easy to select either model, whereas implementing the latter model in a traditional information flow system would require changing every label in the system.

This also explains why Asbestos labels have levels **0**–**3**. We need two levels for send and receive defaults, and levels above and below each of these defaults. In send labels, **1** usually corresponds to the absence of taint; **2** to a "partial taint", as in the latter model, where most communication remains unimpeded; and **3** to full taint, where most communication is prevented. Similarly, in receive labels, **3** indicates the right to be tainted arbitrarily; **2** is the default; and **1** prevents communication with any tainted process. **0** is used for integrity and capabilities, as we'll see below.

Multi-level policies requiring hierarchical sensitivity classification can be emulated in Asbestos using multiple compartments. For instance, to support *unclassified*, *secret*, and *top-secret* levels, the security administrator can use two compartments: one for *secret*, *s*, and one for *top-secret*, *t*. A process's receive label then reflects its owner's security clearance: $\{\mathbf{2}\}$ for *unclassified*, $\{s\,\mathbf{3}, \mathbf{2}\}$ for *secret*, and $\{s\,\mathbf{3}, t\,\mathbf{3}, \mathbf{2}\}$ for *top-secret*. Similarly, send labels reflect the highest level of data a process has actually seen: $\{\mathbf{1}\}$ for *unclassified*, $\{s\,\mathbf{3}, \mathbf{1}\}$ for *secret*, and $\{s\,\mathbf{3}, t\,\mathbf{3}, \mathbf{1}\}$ for *top-secret*.

Odd label values, such as a send label of $\{t\,\mathbf{3}, \mathbf{1}\}$, are also possible. Though this has no direct mapping to a security level, a process with such a send label will only be able to send to processes with *top-secret* clearance, so the desired information flow properties are preserved. In general, however, the Asbestos design is streamlined for large numbers of non-hierarchical compartments rather than traditional, military-style sensitivity classifications. In particular, we believe that scalability to many compartments is a requirement for MAC to protect user data in today's Internet applications.

Receive labels and dynamic taint Asbestos receive labels limit the taint that processes may receive, and thus the effects of taint accumulation. For example, the send labels in Figure 2 will not change with respect to u_T and v_T, absent intervention by some privileged process. Asbestos labels can, however, support a range of other policies. For example, U_{R} and V_{R} could both be set to $\{u_T\,\mathbf{3}, v_T\,\mathbf{3}, \mathbf{2}\}$, allowing either shell to read either user's information. Once *U* reads *v*'s data, it will lose the ability to send messages to *UT*—but, unfortunately, might still be able to convey some information by exploiting covert channels. Following the principle of least privilege, it is better not to raise *U*'s receive label if it doesn't need access to *v*'s data, but this policy choice is up to the application designer. Like Figure 2, our Web server sets receive labels to prevent dynamic taint except where specifically needed.

5.3 Declassification privileges

Asbestos decentralizes declassification using the special $\star$ level: a process with $P_{\mathrm{S}}(h) = \star$ has declassification privilege with respect to *h*, or equivalently, is said to *control* compartment *h*. This privilege concretely means that other processes cannot contaminate *P* with respect to *h*. Even if *P* receives a message from a process *Q* with $Q_{\mathrm{S}}(h) = \mathbf{3}$, $P_{\mathrm{S}}(h)$ remains $\star$, the lowest level. *P* can thus forward data from *Q* to less tainted processes, thereby *declassifying* information with respect to *h*. In notation, define

$$L^\star = \begin{cases} \star & \text{if } L(h) = \star \\ \mathbf{3} & \text{otherwise.} \end{cases}$$

The contamination step from Equation (4) then becomes

$$Q_{\mathrm{S}} \leftarrow Q_{\mathrm{S}} \sqcup (E_{\mathrm{S}} \sqcap Q_{\mathrm{S}}^\star); \tag{5}$$

the $E_{\mathrm{S}} \sqcap Q_{\mathrm{S}}^\star$ term gives $\star$ levels in Q_{S} precedence over contamination from E_{S}. Only a process itself can remove $\star$ levels from its send label, using a special variant of the **send** system call.

In our example, the file server, which is trusted by both users to store their files, and which should apply a minimal taint to any file data it returns (rather than being tainted indefinitely high), has privilege with respect to both u_T and v_T:

$$FS_{\mathrm{S}} = \{u_T \star, v_T \star, \mathbf{1}\},$$
$$FS_{\mathrm{R}} = \{u_T\,\mathbf{3}, v_T\,\mathbf{3}, \mathbf{2}\}.$$

The receive label allows *FS* to receive messages tainted arbitrarily with respect to u_T or v_T; but regardless of the taints it receives, its send label will stay the same for u_T and v_T.

Decontamination A process initially has privilege for every handle it creates: the **new_handle** system call sets $P_{\mathrm{S}}(h) = \star$ for every handle it returns. Since *h* was previously unused, all other processes start with $Q_{\mathrm{S}}(h) \geq \mathbf{1}$ (the default send level). Normal message exchange with *P* will not change this situation. However, Asbestos allows a process with privilege to explicitly distribute privilege to other processes, either by forking or using a mechanism called *decontamination*. This adds flexibility but, since a privileged process could already decontaminate and forward data, doesn't fundamentally change the system's information flow properties. This dynamic compartment creation and privilege manipulation differs from systems such as Jif, which has a fixed hierarchy of users controlling various I/O channels and code.

A process with declassification privilege for handle *h* can decontaminate other processes' labels with respect to *h* by lowering their send labels and raising their receive labels. This uses two more optional label arguments to the **send** system call, namely a *decontaminate-send label* D_{S} and a *decontaminate-receive label* D_{R}. The decontaminate-send label is used to lower the receiver's send label, and the decontaminate-receive label to *raise* the receiver's *receive* label. Both of these operations make the system more permissive, and thus require special privilege with respect to

the handle involved—the privilege represented by $\star$. In notation, Equations (3) and (5) become

$$E_S \sqsubseteq Q_R \sqcup D_R \quad \text{and} \tag{6}$$

$$Q_S \leftarrow (Q_S \sqcap D_S) \sqcup (E_S \sqcap Q_S^\star), \quad Q_R \leftarrow Q_R \sqcup D_R. \tag{7}$$

The system must also check that whenever a decontamination label might change the receiver's labels, the sender controls the relevant compartments: that is, that $P_S(h) = \star$ whenever $D_S(h) < \mathbf{3}$ or $D_R(h) > \star$.

5.4 Integrity

The file server can thus accept requests from any user without fear of contamination and can declassify user data as appropriate. Of course, a useful file server must also implement an *integrity* policy to prevent arbitrary processes from overwriting users' data. An integrity policy can either be mandatory—transitively blocking any flow of low-integrity data into a user's files—or discretionary. Let us first consider a discretionary policy, in which only processes that *speak for* user u can write to u's files, but their writes are free to incorporate data from less trusted sources.

Speaking for u is a positive right, not a taint, and whether a process speaks for u is unrelated to whether or not it has read any of u's secret data. We thus need a new compartment to represent speaking for u, represented by u_G, user u's *grant handle*. A process can speak for u only if $P_S(u_G) \leq \mathbf{0}$. Hence, our file server must verify $P_S(u_G) \leq \mathbf{0}$ before accepting a write to u's file from P.

Asbestos supports such integrity checks with a fourth (and final) optional label argument to **send**, the *verification label* V. The verification label temporarily *lowers*—restricts—the receiver's effective receive label. Thus, the sender proves with V that its labels are below a constraint independent of the receive label. Concretely, the label check from Equation (6) becomes

$$E_S \sqsubseteq (Q_R \sqcup D_R) \sqcap V. \tag{8}$$

Since $E_S = P_S \sqcup C_S$, this implies that $P_S \sqsubseteq V$, and for the check to succeed, the verification label must be an upper bound on the sender's send label. Unlike the other optional labels C_S, D_S, and D_R, the verification label is also passed up to the receiving application when the message is received. Thus, the application knows an upper bound on the sender's send label. In our file server example, a process writing u's file must supply $V = \{u_G\ \mathbf{0}, \mathbf{3}\}$ to prove it speaks for u. The file server, in turn, verifies the process speaks for u by checking $V(u_G) \leq \mathbf{0}$ before accepting a write to u's file.

An alternative design might eliminate V and just supply message recipients with a copy of the sender's send 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, a pitfall known as the *confused deputy* problem [12]. In our example, a process that speaks for multiple users must explicitly name the credentials it intends to exercise for each write.

Level 0 and mandatory integrity The **0** level permits the construction of mandatory integrity policies. For example, a process P with $P_S(u_G) = \mathbf{0}$ can speak for u, but since **0** is less than the default send level of **1**, it cannot further disseminate the privilege: the minute P receives a message from a process Q that does not speak for u ($Q_S(u_G) \geq \mathbf{1}$), P_S will become tainted and P will lose its ability to speak for u. Thus, P cannot act for Q and relay low-integrity data into u's files.

As with secrecy, different defaults in send and receive labels allow targeted exclusion of particular processes. An example is preventing system files from being corrupted from the network. The file server can allocate a compartment, s, and require $V(s) \leq \mathbf{1}$ for writes to system files. Setting the network daemon's send label to $\{s\ \mathbf{2}, \mathbf{1}\}$ then ensures that no process contaminated with data from the network can overwrite system files.

5.5 Capabilities and preventing contamination

The discretionary verification label can be used to implement many application-defined security policies, but it is limited in one important way: An application can choose to ignore a message after examining V, but since the message was already delivered (to allow V to be examined), the application's labels have already been contaminated with the message's taint. In general this taint cannot be undone. Thus, V can flexibly verify integrity but cannot prevent inappropriate contamination. Imagine, for example, a mail reader that starts an untrusted program to read an attachment. The mail reader can, and should, accept contamination from other system processes, such as the file system; but though it needs to communicate with the attachment program, it doesn't want to accept contamination from it. A compromised attachment that develops a high taint should lose the ability to send to the mail reader.

What is needed is a way to shift a simple form of message filtering into the kernel. Asbestos supports this in a straightforward way by integrating communication ports with the label system. The result not only prevents undesired contamination but also ends up providing the semantics of capability-based send rights.

First, the port namespace is the same as the handle value space, so port names can be used as label compartments. Second, every port p is associated with a *port receive label* or *port label* p_R. This label is used to lower, or restrict, the process's receive label, but only for messages delivered to that port. It thus acts like a verification label imposed by the *receiver*, rather than the sender. For a message sent to port p, the label check from Equation (8) becomes

$$E_S \sqsubseteq (Q_R \sqcup D_R) \sqcap V \sqcap p_R. \tag{9}$$

The port label furthermore restricts how much a receive label can be decontaminated. A process that controls a compartment can grant another process the right to receive tainted messages with D_R, and simultaneously taint its send label with C_S. This idiom is common in practice; our Web server uses it, for example, to contaminate worker processes with the relevant user taint u_T. Some processes, such as long-running system servers, may want to avoid undesired taint, however. They do so by setting their port labels to low values (which prevent contamination). The kernel will reject any message that attempts to decontaminate a receive label beyond what is allowed by the port label; specifically, it checks that $D_R \sqsubseteq p_R$.

Port labels, like verification labels, are entirely discretionary. Each process solely controls the port labels for all ports for which it has receive rights, and neither lowering nor raising a port label requires special privilege. Processes supply an initial port label when creating a port; most often this is $\{\mathbf{3}\}$, which adds no restrictions relative to the process's receive label, but it can be $\{\mathbf{2}\}$ or anything else. As a convenience, the kernel modifies this port label by setting $p_R(p) \leftarrow \mathbf{0}$ before returning the new port. Since all other processes in the system initially have $P_S(p) \geq \mathbf{1}$ (the default send level), no other process can send to p until P explicitly grants access. However, the **set_port_label** system call, which changes a port's label, doesn't modify its input. By resetting the port label to $\{\mathbf{3}\}$ (with no exception for p itself), the process can allow anyone in the system to send messages to p, subject only to the process receive label's restrictions.

Capabilities The resulting port label system supports capability-like send rights. When process P first creates port p, no one else can send to p. P can grant the right to send to p by decontaminating another process's send label with respect to p; that is, it can send

P, Q	Processes
p, h	Ports, handles
$\star$, $\mathbf{0}$, $\mathbf{1}$, $\mathbf{2}$, $\mathbf{3}$	Label levels, in increasing order
L, C, D, V, E	Labels (functions from handles to levels)
P_S	Process P's send label
P_R	Process P's receive label
p_R	Port p's receive label
$L_1 \sqsubseteq L_2$	Label comparison: true iff $\forall h, L_1(h) \leq L_2(h)$
$L_1 \sqcup L_2$	Least-upper-bound label: $(L_1 \sqcup L_2)(h) = \max(L_1(h), L_2(h))$
$L_1 \sqcap L_2$	Greatest-lower-bound label: $(L_1 \sqcap L_2)(h) = \min(L_1(h), L_2(h))$
$L^\star$	Stars-only label: $L^\star(h) = \begin{cases} \star & \text{if } L(h) = \star, \\ \mathbf{3} & \text{otherwise} \end{cases}$

Figure 3: Notation.

send$(p, \text{data}, C_S, D_S, V, D_R)$ // Send message to port p
Let Q be the process with receive rights for p
Let $E_S = P_S \sqcup C_S$
Requirements:
(1) $E_S \sqsubseteq (Q_R \sqcup D_R) \sqcap V \sqcap p_R$
(2) If $D_S(h) < \mathbf{3}$, then $P_S(h) = \star$
(3) If $D_R(h) > \star$, then $P_S(h) = \star$
(4) $D_R \sqsubseteq p_R$
Effects:
Grant D_S and contaminate with E_S, but preserve Q_S's $\star$ handles
$Q_S \leftarrow (Q_S \sqcap D_S) \sqcup (E_S \sqcap Q_S^\star)$
$Q_R \leftarrow Q_R \sqcup D_R$

new_port(L)
Let p be an unused port
Effects:
$p_R \leftarrow L$
$p_R(p) \leftarrow \mathbf{0}$
$P_S(p) \leftarrow \star$
Return p

set_port_label(p, L)
Requirement:
P has receive rights for p
Effect:
$p_R \leftarrow L$

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

Q a message with $D_S = \{p \star, \mathbf{3}\}$. Q can then further redistribute that send right. Note that it is primarily the port receive label, p_R, rather than the process label, P_R, that prevents arbitrary processes from sending to p. A process can create many ports with different receive labels and, just like capabilities, separately distribute the right to send to each port.

5.6 Summary and implementation

Figure 3 summarizes the notation developed in earlier sections, and Figure 4 gives the final versions of the label operations associated with the **send**, **new_port**, and **set_port_label** system calls.

In user space, a label is represented as an array of handle values plus a default level. A 64-bit number can represent a label entry: the upper 61 bits are the handle value, the lower 3 bits encode its level in that label.

In kernel space, each active handle corresponds to a 64-byte data structure called a *vnode*. For port handles, this structure includes the port label and a reference to the process with receive rights. A hash table maps handle values to vnodes. Vnodes are reference counted; when all kernel references to a vnode disappear, the kernel may reuse its memory.

Since a series of label operations accompanies every IPC, the kernel label implementation has major impact on performance and memory usage. In our current design, a label points to a sorted array of *chunks*, each of which is a sorted array of up to 64 vnode pointers. Since these pointers are 8-byte aligned, their lower 3 bits are again available for the corresponding levels. Labels are reference counted and updated copy-on-write, so multiple entities can share label memory when appropriate. Additionally, chunks are reference counted and updated copy-on-write, and multiple labels can share chunks. Each chunk is marked with the minimum and maximum of its vnodes' levels, as is each label. This helps optimize certain common operations; for example, if L_2's maximum level is no larger than L_1's minimum level, then $L_1 \sqcup L_2 = L_1$ by definition. In the worst case, of course, operations like $\sqsubseteq$, $\sqcap$, and $\sqcup$ are linear in the size of their input labels. Optimization opportunities remain, for example when most of label's handle levels are $\star$, and we plan to improve the label implementation for future work. The smallest label is about 300 bytes long, including space for one chunk.

6 Event Processes

Labels alone don't work well for processes that handle multiple users' private data. To avoid accumulating contamination, such processes would have to be trusted with declassification privilege by each relevant user, leaving them over-trusted and vulnerable. Existing OS abstractions are no help. On the one hand, user-level threads are efficient but share an address space, and therefore do not provide isolation. On the other, forking a separate process per user provides isolation, but may have low performance due to operating system overheads, such as memory. What's needed is a new abstraction that combines the performance benefits of cooperative user threads with the isolation benefits of forking new processes.

Many efficient servers [20, 33, 43, 45] use an implementation pattern that suggests a solution to this problem. All server work is driven by a simple event-driven dispatch loop:

```
while (1) {
    event = get_next_event();
    user = lookup_user(event);
    if (user not yet seen)
        user.state = create_state();
    process_event(event, user);
}
```

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. The missing piece is a way to isolate the state of different users, and to ensure that the process's labels are set correctly while executing on behalf of each user.

6.1 The event process abstraction

An Asbestos *event process* abstracts the notion of a subset of process state belonging to a single user. As with processes, the kernel restricts an event process's privileges while it handles incoming messages for a user, and isolates different event processes' state; but as with user-level threads, event processes limit concurrency and impose low space and scheduling overheads. Each event process is associated with one conventional *base process*, from which its initial state is drawn. The event process's kernel state consists only of a send label, a receive label, receive rights for ports, and a set of private memory pages, plus some bookkeeping information, altogether occupying 44 bytes of Asbestos kernel memory. For comparison, Asbestos's minimal process structure takes 320 bytes.

The code for a typical event process-based server resembles the event-driven dispatch loop above:

```
1.  ep_checkpoint(&msg);
2.  if (!state.initialized) {
3.      initialize_state(state);
4.      state.reply = new_port();
5.  }
6.  process_msg(msg, state);
7.  ep_yield();
```

Intuitively, many event process entities now share the event loop, each with its own isolated state. The two **ep_** system calls manage control flow transfer between events. The single "state" variable refers to a different user in each event process.

When the base process first calls the **ep_checkpoint** system call, it enters the event process realm, and the base process itself will never run again. The process is de-scheduled until a message arrives on a port for which the base process, *or any of its event processes*, holds receive rights. The kernel then schedules an event process to receive that message. If a particular event process holds receive rights for the relevant port, then **ep_checkpoint** returns in that event process's context, restoring its private labels, receive rights, and memory. If, on the other hand, the *base* process holds the port's receive rights, the kernel *creates a new event process* and returns in that event process's context. The event process starts with send and receive labels copied from the base process's labels, no receive rights, and no private memory pages.

When **ep_checkpoint** returns a message in an event process's context, the label contamination and declassification rules are applied to that event process's labels. The kernel makes event process memory writes private by marking all shared pages copy-on-write, and an event process gets receive rights for any ports it creates.

After it finishes processing its message, the event process calls the **ep_yield** system call. This call saves any changes to the event process's labels, receive rights, and memory and then suspends the whole process, just as when the base process first called **ep_checkpoint**. No event process will run until another message is available for delivery.

Event processes often make temporary modifications to memory that are useful only for the current event. To keep the kernel from saving such memory modifications across **ep_yield**s, an event process can call **ep_clean** to revert a specified memory range to the base process's state. An event process frees all its resources, including its kernel-maintained state, with the **ep_exit** system call.

Event processes can execute most system calls, including sending and receiving messages, allocating memory, and so forth. Event processes' execution states, unlike their memory states, are not isolated: an event process may block indefinitely in **recv**, blocking the entire process, or even exit via the process-wide **exit** system call.

Usage Messages delivered to a base process handle typically correspond to the advent of new client processes or new client network connections, exactly the situations in which it is appropriate to create a new event process. An event process can tell it is new by checking and setting a memory location that the base process initializes to zero; a new event process inherits the zero, while a re-activation of an existing event process will see its previous nonzero write to that location. A new event process will typically allocate itself a new port on which to receive messages, as in line 4 of the above code sample. The system ensures that messages to this port will be delivered to the current event process, which can thus send queries to file or database servers on behalf of the current user and later receive any replies. For some applications, newly created event processes might exit immediately without creating a handle; though this cannot store state between messages, it does avoid accumulating taint.

An event process has all the power of an ordinary process to restrict its labels, for example to reflect the fact that it is processing a specific user's private data. In the multi-user file server example in Section 5.2, the file server would end up contaminating an event process's send label with the user's u_C handle, correctly reflecting that just the event process was contaminated.

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 executes again in user space, and there is no way to change its memory. Different event processes are also unaware of each other's existence except possibly through message-based communication, preserving the independence and isolation represented by per-event process labels. We plan for future work to investigate mechanisms for event processes to selectively share memory, subject to label checks.

6.2 Implementation

Event processes are efficient for two reasons. First, the kernel scheduling cost is little higher than that of a single process, even with many event processes. Second, the memory overhead of an event process can be as low as a single page of memory—to hold the event process's user-level state—plus a few hundred bytes of kernel state.

While event process memory acts like a copy-on-write copy of the base process's memory, the implementation is optimized for event processes that modify very little memory. The memory state of each dormant event process includes just a list of modified pages and the modified pages themselves. That is, event processes do not keep their own page tables. A running event process borrows the base process's page table data structure in the kernel, changing it in exactly those places that differ.

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, others must persist across the processing of several messages. Storing the non-temporary 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.

7 Web Server Design

The Asbestos Web server is based on the OKWS system for UNIX [20]. In the original OKWS design, a *demultiplexer*, *ok-demux*, accepts each incoming TCP connection and parses its HTTP headers to determine what service the remote client is requesting. It then hands the connection off to a *worker process* specialized for providing that service. OKWS's goal is to isolate services, so that one compromised service cannot affect others. Like its UNIX predecessor, OKWS on Asbestos also isolates services in different worker processes, but it additionally enforces user isolation *within* workers via event processes: a compromised worker cannot leak one user's information to other users.

7.1 Startup

OKWS is started by a launcher process. The launcher spawns *ok-demux*, site-specific workers requested by the site operator, and two other processes seen in Figure 1, *idd* and *ok-dbproxy*. The processes

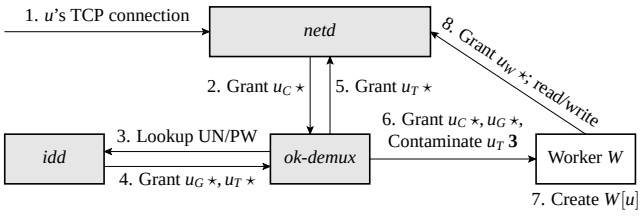

Figure 5: OKWS message flow for handling user *u*'s Web request.

exchange and inherit handles to establish the communication paths seen in the diagram.

The *ok-demux* must be certain that it is communicating with the worker processes that the launcher started and wants to avoid trusting workers to identify themselves correctly. Thus, the launcher grants a process-specific *verification handle* to each process it starts. The *ok-demux* collects these handle values from the launcher. When a worker identifies itself to the *ok-demux*, it must provide a verification label V containing its verification handle at level **0**, allowing the *ok-demux* to verify that it speaks for the relevant process. Other designs, such as having the launcher mediate initial communication with the workers, would also be possible.

In the current prototype, 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).

7.2 Basic connection handling

We now describe the data path of a simple Web request to OKWS running on Asbestos; Figure 5 shows the steps. When a user *u* makes an HTTP connection:

1. The user-level network server *netd* accepts incoming TCP packets from the network. Once *netd* has accepted *u*'s connection, it allocates a new port u_C to act as a "socket" to which processes can send READ and WRITE messages. The port label, u_{CR}, is set to $\{u_C\ \mathbf{0}, \mathbf{2}\}$, so that no process can initially send to it. Section 7.7 describes *netd* further.
2. As it started up, *ok-demux* registered with *netd* to listen for incoming TCP connections on the machine's Web port. Therefore, *netd* notifies *ok-demux* of the new connection by granting it u_C at level $\star$.
3. *ok-demux* reads network data from *u* over port u_C until it can authenticate *u*. Currently, OKWS uses a simple username and password pair, though more sophisticated handshakes are possible. It sends *u*'s username and password to OKWS's identity server, *idd*, which is described in Section 7.4.
4. If *u* provided a valid login, *idd* grants *ok-demux* two handles corresponding to *u*, a *taint handle* u_T and a *grant handle* u_G, both at level $\star$. These handles function like the similarly-named handles in Section 5.
5. *ok-demux* grants $u_T \star$ to *netd*, which then raises its receive label to contain $u_T\ \mathbf{3}$ and raises u_{CR} to $\{u_C\ \mathbf{0}, u_T\ \mathbf{3}, \mathbf{2}\}$. These changes allow u_T-tainted data to escape over the network, but only via u_C. From now on, *netd* will respond to all messages on u_C (such as READs) with replies contaminated with $u_T\ \mathbf{3}$.
6. When *ok-demux* read TCP payload bytes from *netd* in Step 3, it also noted which service *u* is requesting. If this service exists, *ok-demux* forwards u_C to the worker *W* that provides the service, simultaneously contaminating *W*'s send label with $u_T\ \mathbf{3}$ and granting it $u_G \star$.
7. Worker *W* returns from **ep_checkpoint** into a new event process $W[u]$, receiving $u_C \star$, $u_G \star$, and the contamination $u_T\ \mathbf{3}$.
8. Event process $W[u]$ makes a new port u_W and grants it to *netd* at level $\star$. It then reads *u*'s request by sending read requests to *netd*'s port u_C, yielding, and reading *netd*'s replies to u_W upon wakeup. After reading and parsing *u*'s entire request, the event process formulates a reply and writes it to u_C.
9. $W[u]$ calls **ep_exit**.

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

The security of the protocol comes because any process or event process that accesses *u*'s data either is trusted and has $u_T \star$ in its send label, or is not trusted and has $u_T\ \mathbf{3}$. In this example, *netd*, *idd* and *ok-demux* have $u_T \star$, but we assume them uncompromised (see Section 7.8 for further discussion). The event process $W[u]$ and its descendents have had the opportunity to see user *u*'s private data, and therefore have $u_T\ \mathbf{3}$ in their send labels. All other processes, such as those working on behalf of a different user *v*, cannot receive messages from $W[u]$ or its descendents, and therefore cannot receive *u*'s data. Even if such data theft were possible, *netd* would not allow its traffic over the network: any process with $\{u_T\ \mathbf{3}, v_T\ \mathbf{3}\}$ in its send label cannot send data to u_C due to that port's port label restrictions.

Note that while the kernel enforces security policies that isolate user data flows from one another, OKWS's concept of user is opaque to the operating system. Having declassification privilege for an OKWS user's handle, such as u_T, implies nothing about access to sensitive system resources, such as the kernel disk image or the system password file. An Asbestos application like OKWS has no need for "superuser" access, with all of its attendant dangers.

7.3 Web sessions

Since HTTP is stateless, many Web servers support storage of *session data* that persists over multiple HTTP connections. OKWS 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 u_C directly to $W[u]$. If it does not, it forwards *u*'s connection as normal, causing a new event process to be forked. When the new event process allocates port u_W in Step 7, it grants it to *ok-demux*, which then inserts it 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 should 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 user-worker pairs out of its session table.

7.4 Managing identities

In Steps 3 and 4, the *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 grant and taint handles u_T and u_G. When *idd* answers a successful login query in Step 4, it either generates new u_T and u_G handles (if *u* has not logged in recently), or

returns cached u_T and u_G handles if available. In the current implementation, *idd* stores user information in a relational database (see Section 7.5) and never cleans its cache. The identity server has special 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.

7.5 Database interaction

Asbestos offers preliminary database support to worker processes through a port of the Unix package SQLite [40]. A separate process called *ok-dbproxy* interposes on all OKWS database accesses, converting Asbestos labels and security policies to data types and functions native to standard SQLite. With database access, OKWS can 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 contaminate and check labels to ensure secrecy and integrity respectively, and is privileged in that *idd* grants it all user taint handles at level $\star$.

ok-dbproxy adds a "user ID" column to the table definition of every table accessed by OKWS 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 verify label *V* bounded above by $\{u_T\ \mathbf{3}, u_G\ \mathbf{0}, \mathbf{2}\}$. This verify label conveys two important facts. First, the sender's send label does not contain handles other than u_T at level **3**, and therefore the sender has not been contaminated by any data aside from his own. Second, because the verify label contains u_G at level **0**, the sender was granted the ability to write data for *u*, either by *idd* or one of its proxies (i.e., *ok-demux*). After approving the given verify label, *ok-dbproxy* queries *idd* to affirm the binding between user *u* and the two handles u_T and u_G. 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 contaminated with $u_T\ \mathbf{3}$; it queries *idd* for u_T if it does not have this mapping in cache. Each row is returned as a separate message with a separate taint, and to finish the request, *ok-dbproxy* sends an untainted message indicating that all data has been returned. Since each worker's receive label is limited to receiving at most one user's taint, a worker will receive only rows meant for its user, and cannot tell how many other rows were sent. A more relaxed policy could allow workers to be tainted with multiple users' data; but like any dynamic tainting mechanism, this would open a storage channel to the database through worker process labels.

Our current prototype retrofits a standard database with a subset of Asbestos's security features; for instance, rows can only be contaminated with one handle. We envision future database systems built specifically for Asbestos that incorporate labels and event processes in a deeper way.

7.6 Decentralized declassification

Finally, the OKWS prototype supports *decentralized declassification*. As stated above, if any process, such as user *v*'s worker event process, wishes to read user *u*'s data from the database, the database will contaminate the response with $u_T\ \mathbf{3}$. User *v*'s event process will fail to receive this message, since its receive label is not high enough. Even if it were to receive the message, its send label would be too contaminated to send data back to *v* over the connection v_C. But user *u* may *want* to share some data with *v*, such as his public profile. That is, user *u* sometimes needs to *declassify* his private data for public access.

OKWS supports semi-trusted declassifiers for this purpose. A declassifier *D* within OKWS is a worker like any other, except that the launcher tells *ok-demux* of its declassifier status. When *ok-demux* hands a connection off to a declassifier worker *D* in Step 6, it *grants D* the handle $u_T \star$ instead of contaminating it with $u_T\ \mathbf{3}$. With u_T at $\star$, the worker *D* has the privilege to declassify data marked with user *u*'s taint. Thus, when *D* contacts the database to SELECT *u*'s private data, *ok-dbproxy*'s response does not affect *D*'s send label. The declassifier can now write declassified data to the database, providing a verification label of $u_T \star$ to prove it has this right. Internally, *ok-dbproxy* flags a data row as declassified by setting its user ID entry to zero. When *ok-dbproxy* reads data with zeroed user IDs back out of the database, it does not apply any contamination, and user *v*'s worker process can safely read *u*'s declassified data out of the database without affecting its send label. This declassification is decentralized, since it does not directly involve *idd*, the creator of handle u_T. Furthermore, the $D[u]$ event process is trusted only by *u*; it cannot declassify any other user's data, since *ok-demux* only granted it $u_T \star$. An attack on a declassifier worker can expose more of *u*'s data than intended, but cannot otherwise affect the system's information flow.

7.7 Network server

All access to the network in Asbestos is through one process, *netd*, which implements the TCP/IP stack (using a port of LWIP [26]), manages network devices (using a version of Intel's Linux driver for the E1000 gigabit 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, which it grants at level $\star$ to the requesting application. Once a process has a port to an open connection, it may perform READ and WRITE operations to transfer data, CONTROL operations to close the connection or change the 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 taint handle for each connection. When a process tells *netd* to add a taint handle to a connection, later messages sent in response to operations on that connection will be contaminated with the taint handle at level **3**. In OKWS, for example, *netd* contaminates all data read from user *u*'s connection with $u_T\ \mathbf{3}$ at *ok-demux*'s behest.

7.8 Trust and privilege in OKWS

Currently, all OKWS components are trusted and/or privileged except for the worker processes. We claim that for typical Web sites, the 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 this practice. 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.

In the future, we plan to move more OKWS components 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 handles u_T, 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.

8 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 [13]. (Asbestos offers no such feature, however, and the problem admittedly 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. 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 labels. The **ep_yield** system call potentially affects the program counter of a differently tainted 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 [28].

The **send** system call potentially raises the value of the recipient's send label to an unanticipated value. This is also a storage channel, as labels can be observed through lack of communication. Consider a tainted process A attempting to communicate a bit of sensitive information to an untainted process C. An attacker might construct two untainted processes, B_0 and B_1, both of which 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 with run-time checking of dynamic labels [31].

Both of the above channels require at least two processes, which means they can be mitigated by restricting the ability to fork. This illustrates one advantage of the event process abstraction as compared to a more traditional one-label-per-process architecture. Event processes reduce concurrency, thereby also reducing the number of send labels and program counters available as storage channels at any given time.

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

The current implementation still has several other storage channels we intend to close or limit, but we believe these can be mitigated without affecting the claims of the paper. In particular, Asbestos does not yet deal gracefully with certain forms of resource exhaustion.

9 EVALUATION

The goal of this section is to show that OKWS on Asbestos provides a useful level of performance. First, we show the amount of additional memory required to support user isolation in OKWS is small. We then compare OKWS's throughput and latency to that of Apache running on Linux. A prototype OS like Asbestos can hardly be expected to compete with mature, well-tuned systems like these. Our experiments nevertheless show OKWS on Asbestos can already compete with these systems for some scenarios—and indicate fertile ground for future performance improvements.

The performance measurements were conducted on a gigabit local network with a Linux HTTP client generating requests. The Asbestos server is a 2.8GHz Pentium 4 with 1GB of RAM, but Asbestos currently only uses 256MB of RAM. Our experiments made as few file system accesses as possible; we disabled all Web access logging and ran all databases in memory.

9.1 Memory use

In Section 6, 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 7.3, OKWS uses event processes to cache dynamic data while maintaining isolation among users. An event process persists over the lifetime of a Web session, which typically spans multiple HTTP connections. At the end of an HTTP connection, the worker uses **ep_clean** to release all memory allocated, except for the session data. A cleaned event process, with just 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 OKWS with one toy Web service, which stores data from a user's

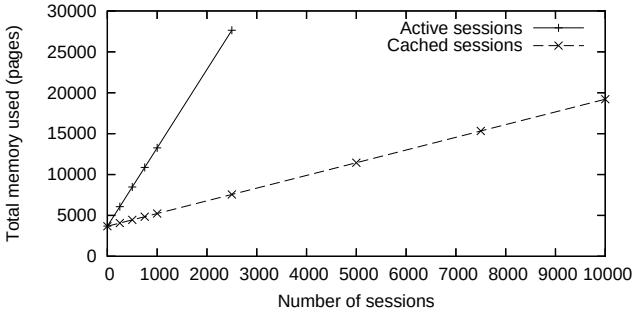

Figure 6: Memory used by active and cached Web sessions as a function of the number of sessions. Includes all memory allocated by both kernel and user programs.

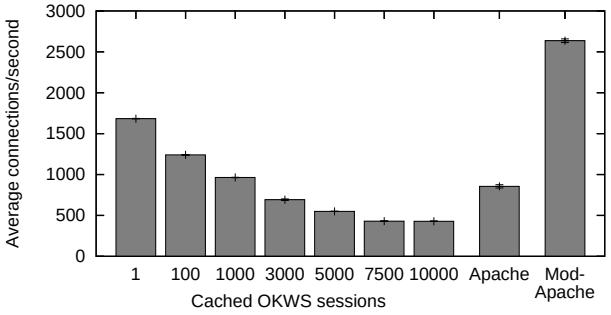

Figure 7: Throughput for various numbers of cached sessions in OKWS, compared with Apache and Mod-Apache.

Server	Latency (μs) Median	90th Percentile
Mod-Apache	999	1,015
Apache	3,374	5,262
OKWS, 1 session	1,875	2,384
OKWS, 1000 sessions	3,414	6,767

Figure 8: The median and 90th percentile latencies of requests to various server configurations.

HTTP request and returns it to the user in the subsequent request. The size of the response is about 1K.

The system uses approximately 1.5 4KB pages per cached session, as seen in Figure 6. 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 handles—as well as potentially memory in other user processes, such as *netd*. The memory required for kernel data structures is around twice as much as we expected, probably due to internal fragmentation or a small memory leak.

To determine the memory cost of active sessions, we repeated the previous experiment but modified the worker so that it does not ever unmap memory, call **ep_clean** or call **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 eight 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 five comprise the modified heap and pages with modified global variables.

9.2 Web server performance

We examined two aspects of Web server performance: throughput and latency. In these experiments, we tested an even simpler Web application, which simply responds with a string of characters whose length depends on the client's parameters. We compared OKWS on Asbestos to the Apache Web Server, version 1.3.33 [2] (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 [1]. 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 if the CGI is exploitable, any vulnerabilities exploitable by a UNIX user on the system are accessible. In contrast, as discussed previously, OKWS provides isolation both between services and between users within a service.

9.2.1 Throughput

To test throughput for OKWS relative to Apache and Mod-Apache, we first varied concurrency to maximize completed connections per second. For Apache, 400 concurrent connections maximized performance; for Mod-Apache 16 concurrent connections was the sweet spot. Asbestos's network stack is based on LWIP [26], which was chiefly designed to conserve resources and does not offer good performance under load; sixteen concurrent connections gave maximum throughput. For OKWS, 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.

Since OKWS isolates users, they were authenticated and run in different event processes as usual. We measured performance with the session support described in Section 7.3: once authenticated to the system, future requests were serviced by the event process created in the authentication step. The OKWS throughput results thus contain data both for forwarding messages to existing event processes and for creating new event processes, which is slower—it 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 somewhat 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 Apache nor Mod-Apache isolates users, so no attempt to authenticate them is made in this test.

Figure 7 shows that, with one session, OKWS performs better than Apache, and a bit over half as well as Mod-Apache. OKWS performs better than Apache until somewhere over one thousand sessions are cached in the system, but even with 10,000 separate event processes, each holding isolated memory state, it performs approximately half as well as Apache. Section 9.3 further discusses the factors that reduce OKWS's performance as sessions increase.

9.2.2 Latency

This section compares the per-request latency of OKWS on Asbestos with Apache and Mod-Apache. Stuck with low-concurrency Asbestos, we measured the latency of all three servers with a concurrency of only four simultaneous connections. 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. Un-

like 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 8, OKWS with one user has a smaller median latency than Apache, as well as a smaller variance. Scheduling affects OKWS to a lesser extent because there is no parallelism for requests to choose from. All requests must sequentially traverse *netd*, *ok-demux*, worker, and then *netd* again, which doesn't give the option for any request to be temporarily starved. OKWS with 1000 cached sessions has latencies which are just a bit worse then those of Apache.

9.3 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 9 shows the costs of various components in the system in thousands of CPU cycles per connection as the number of cached sessions increases. The OKWS and Network lines represent the time spent in OKWS and *netd* code, respectively. The Kernel IPC line includes all time spent in processing **send** and **recv**, which includes most of the system's label operations; time spent in other label operations is included as well. The OKDB line represents time spent in the SQLite database looking up usernames and passwords, and any remaining processing time is accounted for in the Other line.

With one session in the system, most of the processing time is in OKWS and the network stack. As the number of sessions increases, database overhead incurred by user authentication quickly becomes significant. This may simply represent another cost of using unoptimized system components, in this case SQLite. However, label and other kernel IPC operations also take significantly more time as sessions increase. Since OKWS uses two handles to isolate a user, 10,000 cached sessions implies *idd* and *ok-dbproxy*'s send labels will contain more than 20,000 handles; *netd*'s receive label will have accumulated 10,000 declassifications with respect to taint handles; and *ok-demux* will hold at least 10,000 handles 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. Around 3,000 sessions, time spent doing IPC and label operations surpasses time spent in the network stack. By 7,500 sessions, it equals the work being done in all of OKWS. As expected, linear scaling factors in our label implementation lead to linear performance degradation as labels increase in size. Further optimization opportunities are under investigation, as is clearly required. However, we are pleased that the degradation is relatively mild, with no obviously quadratic or exponential factors. As we hypothesized, Asbestos labels and event processes make it practical to isolate user state even on a server storing data for thousands of users.

10 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, which, unlike schemes in previous operating systems, allows data to be sanitized or *declassified* by individual users within categories they control. The categories, called handles, use the same names as communication endpoints, making them a kind of generalization of capabilities. As in a capability system, processes can dynamically generate new handles and distribute them independently, and processes can specify temporary label restrictions on sent messages to avoid the unintentional use of privilege.

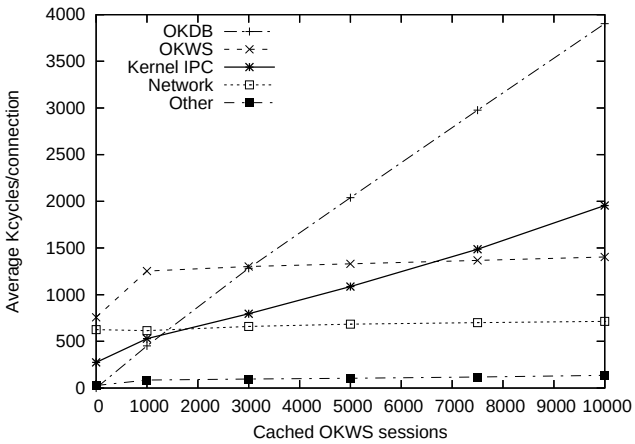

Figure 9: The average cost in Kcycles/connection of various Asbestos components, as the number of cached sessions increases.

Asbestos also introduces a new process abstraction, event processes, which allow a server process to inhabit disjoint security compartments without either privilege or contamination. Event processes impose less overhead on the operating system than forked address spaces, so many thousands of them can theoretically 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 1.5 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; Chris Frost and Mike Mammarella for network stack integration; Michelle Osborne for work on an earlier version of the system; the contributors to the LWIP project, including Adam Dunkels and Leon Woestenberg; the anonymous reviewers; and our shepherd Emin Gün Sirer.

This work was supported by DARPA grants MDA972-03-P-0015 and FA8750-04-1-0090, and by joint NSF Cybertrust/DARPA grant CNS-0430425. David Mazières and Robert Morris are supported by Sloan fellowships.

References

[1] Apache API notes. `http://httpd.apache.org/docs/1.3/misc/API.html`.

[2] Apache HTTP server project. `http://httpd.apache.org`.

[3] David E. Bell and Leonard La Padula. Secure computer system: Unifi ed exposition and Multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, March 1976.

[4] Viktors Berstis. Security and protection of data in the IBM System/38. In *Proc. 7th Annual Symposium on Computer Architecture (ISCA '80)*, pp. 245–252, May 1980.

[5] M. Branstad, Homayoon Tajalli, Frank Mayer, and David Dalva. Access mediation in a message passing kernel. In *Proc. 1989 IEEE Symposium on Security and Privacy*, pp. 66–72, Oakland, CA, May 1989.

[6] David R. Cheriton. The V distributed system. *Journal of the ACM*, 31(3):314–33, March 1988.

[7] Dorothy E. Denning. A lattice model of secure information fbw. *Communications of the ACM*, 19(5):236–243, May 1976.

[8] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[9] Department of Defense. *Trusted Computer System Evaluation Criteria (Orange Book)*, December 1985. DoD 5200.28-STD.

[10] Timothy Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proc. 2000 IEEE Symposium on Security and Privacy*, pp. 230–245, Oakland, CA, May 2000.

[11] R. P. Goldberg. Architecture of virtual machines. In *Proc. AFIPS National Computer Conference*, Vol. 42, pp. 309–318, June 1973.

[12] Norman Hardy. The confused deputy (or why capabilities might have been invented). *Operating Systems Review*, 22(4):36–38, October 1988.

[13] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *Proc. 1991 IEEE Symposium on Security and Privacy*, pp. 8–20, Oakland, CA, May 1991.

[14] Trent Jaeger, Atul Prakash, Jochen Liedtke, and Nayeem Islam. Flexible control of downloaded executable content. *ACM Transactions on Information and System Security*, 2(2):177–228, May 1999.

[15] Paul A. Karger. Limiting the damage potential of discretionary Trojan horses. In *Proc. 1987 IEEE Symposium on Security and Privacy*, pp. 32–37, Oakland, CA, April 1987.

[16] Paul A. Karger and Andrew J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proc. 1984 IEEE Symposium on Security and Privacy*, pp. 2–12, Oakland, CA, April 1984.

[17] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A VMM security kernel for the VAX architecture. In *Proc. 1990 IEEE Symposium on Security and Privacy*, pp. 2–19, Oakland, CA, May 1990.

[18] Key Logic. *The KeyKOS/KeySAFE System Design*, March 1989. SEC009-01. `http://www.agorics.com/Library/KeyKos/keysafe/Keysafe.html`.

[19] Samuel T. King and Peter M. Chen. Operating system support for virtual machines. In *Proc. 2003 USENIX Annual Technical Conference*, San Antonio, TX, June 2003.

[20] Maxwell Krohn. Building secure high-performance web services with OKWS. In *Proc. 2004 USENIX Annual Technical Conference*, pp. 185–198, Boston, MA, June 2004.

[21] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proc. 10th Hot Topics in Operating Systems Symposium (HotOS-X)*, Santa Fe, NM, June 2005.

[22] Carl E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[23] Robert Lemos. Payroll site closes on security worries, February 2005. `http://news.com.com/2102-1029_3-5587859.html`.

[24] Jochen Liedtke. On microkernel construction. In *Proc. 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, December 1995.

[25] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. 2001 USENIX Annual Technical Conference—FREENIX Track*, pp. 29–40, June 2001.

[26] LWIP. `http://savannah.nongnu.org/projects/lwip/`.

[27] Catherine Jensen McCollum, Judith R. Messing, and LouAnna Notargiacomo. Beyond the pale of MAC and DAC—defining new forms of access control. In *Proc. 1990 IEEE Symposium on Security and Privacy*, pp. 190–200, Oakland, CA, May 1990.

[28] M. Douglas McIlroy and James A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, August 1992.

[29] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. `http://www.erights.org/elib/capability/duals/`.

[30] James G. Mitchell, Jonathan Gibbons, Graham Hamilton, Peter B. Kessler, Yousef Y. A. Khalidi, Panos Kougiouris, Peter Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia. An overview of the Spring system. In *Proc. COMPCON 1994*, pp. 122–131, February 1994.

[31] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, October 2000.

[32] News10. Hacker accesses thousands of personal data files at CSU Chico, March 2005. `http://www.news10.net/storyfull1.asp?id=9784`.

[33] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proc. 1999 USENIX Annual Technical Conference*, pp. 199–212, Monterey, CA, June 1999.

[34] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[35] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proc. 8th ACM Symposium on Operating Systems Principles*, pp. 64–75, Pacific Grove, CA, December 1981.

[36] Marc Rozier, Vadim Abrossimov, François Armand, I. Boule, Michel Gien, M. Guillemont, F. Herrmann, Claude Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS distributed operating system. *Computing Systems*, 1:305–370, Fall 1988.

[37] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278–1308, September 1975.

[38] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *Proc. Fast Software Encryption, Cambridge Security Workshop*, pp. 191–204. Springer-Verlag, December 1993. LNCS 809.

[39] Jonathan S. Shapiro, Jonathan Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pp. 170–185, Kiawah Island, SC, December 1999.

[40] SQLite. `http://www.sqlite.org`.

[41] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[42] VMware. VMware and the National Security Agency team to build advanced secure computer systems, January 2001. `http://www.vmware.com/pdf/TechTrendNotes.pdf`.

[43] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for Internet services. In *Proc. 19th ACM Symposium on Operating Systems Principles*, pp. 268–281, Bolton Landing, Lake George, NY, October 2003.

[44] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proc. 2003 USENIX Annual Technical Conference*, pp. 285–296, San Antonio, TX, June 2003.

[45] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pp. 230–243, Château Lake Louise, Alberta, Canada, October 2001.

[46] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pp. 195–210, Boston, MA, December 2002.