Abstractions for Usable Information Flow Control in Aeolus

Winnie Cheng[*] Dan R. K. Ports David Schultz
Victoria Popic[†] Aaron Blankstein[‡] James Cowling
Dorothy Curtis Liuba Shrira[§] Barbara Liskov

MIT CSAIL [*]*IBM Research* [†]*Stanford* [‡]*Princeton* [§]*Brandeis*

Abstract

Despite the increasing importance of protecting confidential data, building secure software remains as challenging as ever. This paper describes *Aeolus*, a new platform for building secure distributed applications. Aeolus uses information flow control to provide confidentiality and data integrity. It differs from previous information flow control systems in a way that we believe makes it easier to understand and use. Aeolus uses a new, simpler security model, the first to combine a standard principal-based scheme for authority management with thread-granularity information flow tracking. The principal hierarchy matches the way developers already reason about authority and access control, and the coarse-grained information flow tracking eases the task of defining a program's security restrictions. In addition, Aeolus provides a number of new mechanisms (authority closures, compound tags, boxes, and shared volatile state) that support common design patterns in secure application design.

1 Introduction

Confidential information, such as credit card numbers and medical records, is increasingly stored online. Keeping this information secure despite malicious attacks and human errors is a high priority, as evidenced by recent regulatory requirements [8, 11]. Building secure software, however, remains as challenging as ever.

Information flow control offers a promising option for construction of secure software. Traditionally, information has been secured through access control, which constrains who is allowed to read and write information. Information flow control complements this by allowing an untrusted entity access to sensitive data as long as it does not reveal the data. It has long been of interest in military systems [21], where having access to "top secret" information does not imply the information can be released to an "unclassified" user.

The *decentralized* information flow control (DIFC) model [15] generalizes the approach and makes it useful for arbitrary applications, by replacing central control with the ability for individuals to define restrictions on the use of their own information. However, despite much recent research on DIFC systems [6, 9, 15–17, 23, 24], information flow control has not been widely used in practice. We believe this is because there are several requirements that are not met effectively by existing systems:

1. Developers require an understandable and flexible authority structure. DIFC depends on the use of authority to determine whether information can be released or trusted. Programmers who use DIFC must be able to understand the authority structure their applications depend on, and they must be able to change this structure, both by establishing new lines of authority and by revoking existing authority.
2. Developers require support for the principle of least privilege, to limit the amount of code that must be verified to ensure security. It must be both possible and *convenient* to run code with reduced privilege.
3. Developers expect a general programming model, including support for distributed programs and concurrent threads with shared variables.
4. Developers need to do all of the above in the context of a familiar programming language.

This paper introduces *Aeolus*, a new DIFC platform developed to satisfy the above requirements. Aeolus is designed for building distributed applications, such as a web service that stores many users' data on multiple servers, or a medical records system accessed from computers in doctors' offices. Aeolus addresses two data security issues: *confidentiality* and *integrity*. Confidentiality ensures that secrets cannot leak except through explicit privileged operations. Integrity ensures that data from untrusted sources is not trusted inadvertently.

Aeolus supports the first requirement by providing a new security model with simple rules based on *principals* and *tags*. Principals represent entities with security interests, and tags allow principals to categorize information. Both are familiar real-world concepts that developers are accustomed to reasoning about. This model allows for structured, fine-grained delegation of authority, through an *authority state*. The authority state allows policies to be specified declaratively and can be used to enforce mandatory *policy constraints* such as separation of duties. Moreover, Aeolus readily supports *revocation*.

Aeolus meets the second requirement by providing new abstractions that support the principle of least privilege [18]. Support for this principle is important for im-

plementing secure applications, but if the mechanisms for limiting privilege are not convenient they will rarely be used in practice – as anyone who has attempted privilege separation on Unix knows all too well. Aeolus's runtime environment provides programmers with the ability to invoke functions with reduced authority, and provides *authority closures*, which allow authority to be delegated to particular programs without fear that that trust can be misused to run other code.

To support the third requirement, Aeolus provides a number of additional abstractions. Aeolus allows programs to run concurrent threads with different levels of contamination. It uses a memory-safe language to isolate threads from each other, while providing a low-overhead *secure shared state* mechanism that allows for efficient sharing while still enforcing information flow restrictions. Aeolus supports distributed programs with a secure RPC mechanism, and provides *boxes*, which allow confidential information to be communicated without contaminating intermediaries that do not observe the information.

Aeolus is a dynamic DIFC system implemented in a set of runtime libraries. Thus, it is OS-independent, and allows programmers to write code in a familiar and conventional programming language, thereby addressing the final requirement. The Aeolus runtime environment runs on all nodes in a distributed system, and allows communication between them while enforcing information flow restrictions. This paper describes our implementation of Aeolus for Java, and shows that it has low performance cost. We have also ported parts of Aeolus to C# and PHP.

Aeolus differs from but is inspired by both streams of current DIFC research: programming languages that enforce static restrictions on information flow [14, 15, 17], and operating systems that track information flow dynamically [6, 9, 23, 24]. Aeolus has more in common with the operating system work, as it tracks information flow dynamically at the granularity of threads rather than requiring programmers to specify restrictions at the level of individual variables. However, because it operates at the language runtime level rather than the OS level, it can provide higher level abstractions for writing secure programs, such as threads with secure shared state. Whereas existing DIFC operating systems manage authority with capabilities, Aeolus uses the authority state. This readily supports revocation and policy constraints, which are difficult to achieve in capability systems.

2 Aeolus Architecture

Aeolus provides information flow control for a distributed computing environment; the architecture is shown in Figure 1. The system consists of many nodes, each of which runs the Aeolus platform and is trusted to enforce Aeolus's information flow rules. Accordingly, only trusted nodes are allowed to enter the system.

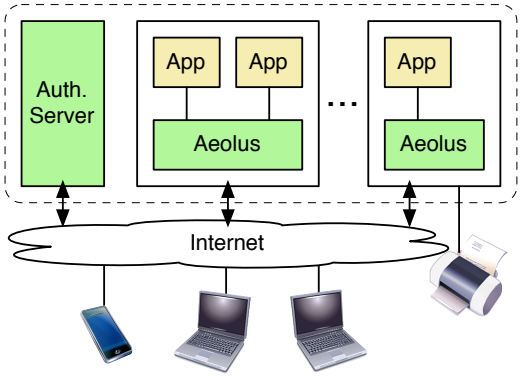

Figure 1: System Architecture

Aeolus tracks information flow within each system node and between nodes. It allows sensitive information to flow between system nodes, but the messages are encrypted and authenticated so that secrecy and integrity are protected. Aeolus restricts communication with nodes outside the system and to I/O devices, all of which are considered to be untrusted. Information can flow out of the system only if it is uncontaminated, and information arriving from the outside is marked as having no integrity.

Aeolus vests authority in principals, which it represents by principal IDs; each thread in Aeolus runs for some principal. Similarly to other dynamic information flow control systems, Aeolus tracks information used by programs as they execute, and determines whether programs have the authority to perform security-sensitive operations, such as declassification. This requires a way to track the authority of principals. Aeolus does this through *authority state*; this is shown as residing at a single authority server (AS), but a distributed implementation is also possible. The AS also tracks system membership; only nodes registered at the AS are in a deployment.

Threat Model. Aeolus is aimed at preventing errors and malicious behavior from undermining information security. The goal is to allow applications to minimize the amount of their code that needs to be trusted, following the principle of least privilege. Like much prior work on information flow control, we do not attempt to address covert-channel and side-channel attacks by malicious software running in the system, or by users who might transcribe data from the screen. However, we have been careful to ensure that our mechanisms do not introduce additional covert channels.

The trusted computing base (TCB) for our system consists of our platform code, together with the lower layers on which it runs: the OS and hardware. We also require a secure authentication service. In our prototype, we implement our platform atop the Java Virtual Machine (JVM), and therefore the TCB includes this as well.

Because the trusted computing base includes a com-

Principals.

- `createPrincipal()` $\rightarrow P$. Returns a new principal; the creating process's principal acts for P.
- `actsFor`(P_1, P_2). Adds an acts-for link from P_2 to P_1. The process must act for P_1, and the link must not create a cycle.
- `revokeActsFor`(P_1, P_2). Removes the acts-for link from P_2 to P_1 (if one exists); the process must act for P_1.

Tags.

- `makeTag()` $\rightarrow t$. Returns a new tag; the process's principal is authoritative for t.
- `makeSubtag`(t_1) $\rightarrow t_2$. Returns a subtag of a top-level tag t_1. The process must be authoritative for t_1, and becomes authoritative for t_2.
- `grant`(t, P_1, P_2). Adds a delegation link for t from P_1 to P_2. P_1 must be authoritative for t, the process must act for P_1, and the link must not create a cycle.
- `revokeGrant`(t, P_1, P_2). Removes the delegation link for t from P_1 to P_2 (if one exists); the process must act for P_1.

Labels.

- `addSecrecy`(t). Adds t to the secrecy label.
- `declassify`(t). Removes t from the secrecy label. The process must be authoritative for t.
- `removeIntegrity`(t). Removes t from the integrity label.
- `endorse`(t). Adds t to the integrity label. The process must be authoritative for t.

Figure 2: Some operations for principals, tags, and labels.

modity operating system and the JVM, Aeolus has a larger TCB than many DIFC operating systems. Our focus in this work has not been on reducing TCB complexity but on identifying the right abstractions for developers to build secure applications; providing a minimal-TCB implementation of the same abstractions is an interesting direction for future work. In addition, we believe supporting existing operating systems and languages is important for the system to be adopted.

3 Information Flow Model

This section describes the basic concepts and rules of the Aeolus security model. Figure 2 shows part of the API. The complete API is described in the Aeolus reference manual [1].

3.1 Principals, Tags, and Labels

The Aeolus model is based on three key concepts: principals, tags, and labels. *Principals* represent entities with security interests, such as individuals or companies. *Tags* provide a way for principals to categorize their information. For example, a user Bob might define three tags, for his public, financial, and medical information.

Labels are sets of tags and are used to control information flow. Each data object (such as a file) and each thread has two labels: a secrecy label, L_S, which reflects confidentiality of information, and an integrity label, L_I, which reflects the integrity of information. The labels of data objects are immutable: they are assigned when the object is created and cannot be changed. Thread labels are mutable: as a thread executes, its labels can change to reflect the secrecy and integrity of the information the thread has observed, subject to the rules defined below.

Aeolus maintains *security state* for each thread, consisting of its two labels and its associated principal.

3.2 Information Flow Rules

Information flow from a source S to a destination D is allowed only if two rules about their labels are satisfied:

- Secrecy Rule: $S.L_S \subseteq D.L_S$
- Integrity Rule: $S.L_I \supseteq D.L_I$

The secrecy rule ensures that confidentiality is maintained as data propagates, while the integrity rule keeps track of influences of low-integrity entities. These rules are an instantiation of the conventional lattice-based rules [4].

A thread can manipulate its labels by adding and removing tags. Adding a tag to a secrecy label and removing a tag from an integrity label are safe manipulations, since the thread only increases its contamination or reduces its integrity. However, the following *privileged* label manipulations are unsafe because they remove constraints on information flow:

- *Declassification*. Remove a tag from a secrecy label.
- *Endorsement*. Add a tag to an integrity label.

Therefore, a thread can perform privileged label manipulations only when its principal *has authority for* the tag being added or removed. Section 3.3 discusses authority

All label manipulations must be done *explicitly*. Aeolus differs in this respect from existing DIFC operating systems, which declassify automatically when a thread with authority for a tag reads an object with that tag in its label [6,23]. Forcing programmers to be explicit prevents leaks due to unintended uses of authority.

3.3 Authority

Authority determines whether a thread can perform privileged label manipulations (declassification and endorsement). Authority starts with tag creation: when a thread creates a tag, its principal has authority for that tag. Subsequently, authority can be delegated either via *acts-for* relationships or via *grants*. Furthermore, previously delegated authority can be *revoked*.

When a new principal is created, the principal of the thread that creates it acts for it and thus has all authority of the new principal. Subsequently, a thread that acts for a principal can delegate that authority to other principals.

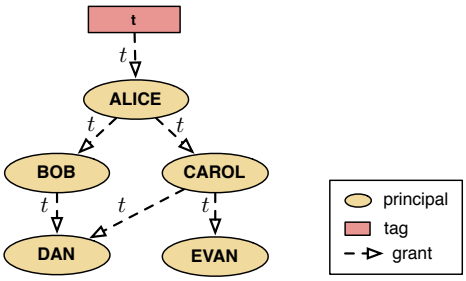

Figure 3: Delegation graph for a tag t

Information about acts-for relations is maintained in the *principal hierarchy*, which is a directed acyclic graph.

The principal hierarchy is useful to capture authority relationships for organizations (groups), and also allows individuals to use different principals for different purposes (roles). A special principal, P_{PUBLIC}, acts for no principals, and can be used to run a computation with no authority. All principals act for P_{PUBLIC}.

Delegation through the principal hierarchy is a blunt instrument, because when one principal acts for another it has *all* the authority of that principal. Grants provide a safer, more controlled, delegation of authority: a principal can grant its authority for a particular tag to another principal. Just as tags allow users to categorize information and provide separate controls for different categories, grants allows users to control authority over those categories.

The delegation hierarchy for a tag is also a directed acyclic graph, as shown in Figure 3; each tag has its own graph. Here, tag t was created by principal ALICE, ALICE has delegated authority for t to both BOB and CAROL, BOB has delegated his authority to DAN, and CAROL has delegated authority to DAN and EVAN. The principals shown in the figure all have authority for t, and so do any principals that act for them either directly or transitively.

Revocation. Revocation of authority is an important concern in real systems. Aeolus allows both acts-for relationships and grants to be revoked. Revocation removes a particular link from the principal hierarchy or delegation graph, and that revocation is transitive. For example, if ALICE revokes her delegation to CAROL in Figure 3, CAROL and EVAN lose authority for t but DAN does not.

Of course, performing a revocation requires authority. If P_2 acts for P_1, that link can be revoked only by a thread that acts for P_1. Similarly, a grant can be revoked only by a thread that acts for the grantor.

3.4 Compound Tags

Applications frequently have sets of tags that are closely related. Capturing such a relationship makes it easier to understand the authority structure of the application and more efficient to run the application.

Aeolus allows developers to express such relationships with *compound tags*. This mechanism allows tags to be grouped statically, as they are created. For example, the medical records system we describe in Section 5.1 has a tag for each patient's data, each a subtag of the ALL-PATIENT-DATA compound tag. Compound tags simplify delegations and label manipulations. Authority for the entire group of tags can be granted by delegating authority for the compound tag. Compound tags reduce label size substantially, since the label only needs to contain the top-level tag, and make declassification inexpensive as only the top level tag must be removed.

3.5 Manipulating Authority State

Aeolus maintains *authority state*, which consists of the principal hierarchy, tags, and their delegations. Applications can modify the authority state by creating principals and tags, or by delegating or revoking authority. These modifications create opportunities for covert channels through the authority state. For example, a malicious application could leak secret information by granting authority for certain tags to a co-conspirator based on the contents of the secret; the co-conspirator then learns about the secret by observing which tags it was granted authority for.

We avoid these covert channels by permitting only threads with null secrecy labels to modify the authority state (i.e., the authority state itself is an object with a null secrecy label). We believe this is a reasonable restriction, as modifications to the principal hierarchy are rare and typically do not occur during normal computation. For instance, authority state is modified when a new user is created but not when that user performs operations. Alternate approaches that allow the principal hierarchy to be modified by contaminated threads are possible, at the cost of increased complexity [19, 20].

3.6 Policy Constraints

An important concern in systems that support dynamic security policies is ensuring that the policies themselves are correct. An example of a correctness requirement is separation of duties between doctors, who can view their patients' sensitive medical data, and administrative assistants, who can view billing and insurance records: no principal ought to be authoritative for both roles.

Such invariants can be enforced by stating *policy constraints*, which are predicates that the authority state must satisfy. Aeolus prevents modifications to the authority state that violate a policy constraint. To define a constraint, a principal must have authority for every principal and tag the constraint covers. The ability to define constraints over an explicit principal hierarchy is an advantage of Aeolus's authority model over capability-based systems.

4 Programming Model

This section describes the programming abstractions Aeolus provides, and explains how they support practical DIFC and the principle of least privilege.

4.1 Threads and Virtual Nodes

Aeolus applications consist of multiple threads, each with its own security state (principal, secrecy label, and integrity label). Aeolus threads can share memory, but their accesses must obey the information flow restrictions. Each thread's memory is private; other threads cannot access it. Threads can share objects only through Aeolus's shared state mechanisms, described in Section 4.2, which enforce the information flow restrictions.

Distributed applications can run on multiple physical machines, and multiple applications can be run on the same physical machine. To support this, each thread is part of a *virtual node*. Each virtual node runs on a single physical node and may contain many threads. Typically, an application will run one virtual node on each physical node it uses. Virtual nodes are used to isolate applications: as we describe below, only threads in the same virtual node can share memory. Threads can also communicate with other virtual nodes via RPC as discussed in Section 4.4, and through Aeolus's distributed file system.

Importantly, shared state, RPCs, and the Aeolus file system are the *only* mechanisms by which Aeolus allows threads to communicate with each other. All these mechanisms check the threads' secrecy and integrity labels. Thus, information cannot leak from one thread to another if it is not permitted by the information flow rules of Section 3.2.

4.2 Shared Objects and Boxes

Aeolus allows threads to securely share state, while enforcing information flow restrictions. The mechanisms ensure that each thread's labels accurately reflect information communicated through shared state.

Aeolus uses two rules to enforce secure sharing. First, each object in shared state has labels, and a thread can only read or modify the object if permitted by the standard flow control rules. Second, shared objects are *encapsulated*: threads cannot obtain pointers to the interior of shared objects, which prevents the threads from bypassing the label checks. Similarly, shared objects cannot contain pointers to the local memory of any thread. To ensure proper encapsulation, Aeolus performs deep copies of arguments and results of calls on shared objects: it recursively follows pointers and copies objects, except for pointers to other encapsulated shared objects.

Users can define shared objects with arbitrary methods; the Aeolus platform adds runtime label checks and copies arguments to ensure encapsulation. Aeolus conservatively assumes that each method of a shared object both reads and writes the object (a flow out and a flow in, respectively). Therefore, calls are allowed only if the labels of the thread match those of the object *exactly*. In addition, the thread's label cannot be changed while executing a shared object method.

Aeolus provides three kinds of built-in shared objects with less restrictive label rules. *Shared queues* provide a form of IPC and *shared locks* provide IFC-aware synchronization among threads. *Boxes* are in-memory containers whose labels reflect the contamination of the information inside the box. A thread can copy data into the box or copy data from the box if its labels and the box's labels allow the flow.

Boxes provide special semantics for RPCs (see Section 4.4): contaminated information can be sent inside a box and the recipient becomes contaminated only when it opens the box to retrieve the content. For example, a web server can receive a box containing a password from a user and pass it to an authentication service without looking at the password itself. Similar control could be achieved by using the file system, by having the caller send the pathname of a file containing the tagged information, but using boxes is simpler and more efficient.

Every virtual node has a *root* object that its threads can use to access shared state. This object has null labels and is typically used to locate other shared objects. A shared object is inaccessible when it cannot be reached from the root or from any thread. Storage for inaccessible shared objects is collected automatically.

4.3 Principle of Least Privilege

The principle of least privilege is essential for building secure applications, because it prevents bugs from becoming critical security failures. Aeolus must make it possible to ensure that each part of the application runs with only the authority it needs. More than that, it must make it *convenient* and *efficient* to do so, lest this principle fall by the wayside in practice. Temporarily dropping authority and regaining it should be as easy as making a function call.

Aeolus supports the principal of least privilege using two mechanisms: *reduced authority calls*, which allow a thread to drop privilege, and *authority closures*, which execute code with previously bound-in authority.

Reduced authority calls are straightforward: the caller specifies a function to run, and the principal to run it with. The caller must act for that principal (i.e. it must reduce authority, not raise it). We expect applications to use these calls frequently to drop privilege they do not need. Aeolus also provides reduced authority forks, which start new threads. During a fork, the arguments to the call must be copied into the memory of the new thread, so that it does not share memory with the old one.

An *authority closure* is an object that is bound to a

principal. The principal is specified when creating the closure; the thread that creates the closure must act for that principal. Thereafter, any thread can call methods of the closure. Calls start running with the labels of the caller, but the authority (principal) of the closure. On return, the caller's labels are merged into the thread's labels – a union for the secrecy labels and an intersection for the integrity labels. This allows the closure to use its authority to remove contamination added during its own execution, but it cannot remove contamination its caller already had.

Programmers create new authority closures by defining subclasses of AeolusClosure. Closure objects are vested with authority when they are instantiated: the AeolusClosure constructor is passed a principal, which the caller must act for. The Aeolus runtime treats closure objects specially: whenever one of their methods is invoked, the system switches the thread's principal to the closure's for the duration of the call and performs the label manipulations described above. An example of an authority closure is shown in Section 5.3.

4.4 RPCs, External I/O, and Files

Aeolus applications use RPCs to communicate between virtual nodes. An application makes a closure available for RPC by binding it to a name. When a remote thread invokes the RPC, a new thread in the closure's virtual node executes the call with the authority of the closure and the labels of the caller, like calling a closure locally.

A thread must have a null secrecy label to bind a closure to an RPC name since otherwise the existence of an RPC with a particular name could be a covert channel. This restriction is not problematic because applications typically register RPCs when they start.

Clients outside the system can send requests to Aeolus nodes using the RPC protocol or by using sockets. Data received from outside the system is given a null integrity label since we cannot vouch for its validity. Replies can only be sent outside the system if the sender's secrecy label is null, since we cannot guarantee confidentiality of data sent to the outside. Furthermore, boxes cannot be sent externally because external nodes are not trusted with the contents of the box.

Finally, Aeolus provides a network file system that enforces DIFC. Files have immutable labels and access is allowed only if the label constraints are satisfied; the rules are similar to those in HiStar [23].

5 Evaluation: Expressive Power

This section evaluates Aeolus with respect to the goals established in the introduction: understandable and flexible authority structure, support for the principle of least privilege, and support for distributed and concurrent programs. We do this by examining three examples, each

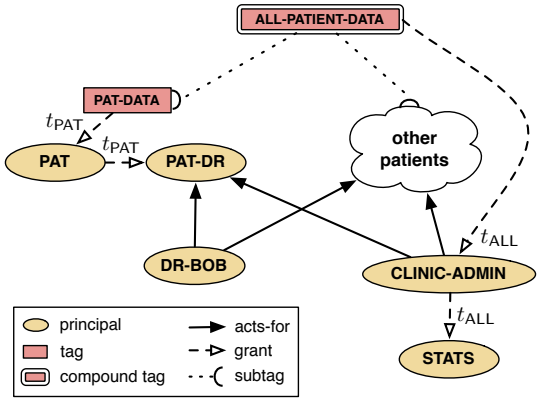

Figure 4: A portion of the authority state for a medical clinic. For each patient, there is a principal for the patient (PAT), a principal representing the patient's doctor (PAT-DR), and a tag (PAT-DATA) applied to the patient's records.

representing a class of similar applications:

- The medical information system (Section 5.1) represents systems in which there are complex security relationships between individuals in an organization.
- Bob and the tax preparer (Section 5.2) represents distributed computations with two parties with different security interests, and demonstrates the use of boxes.
- The online personal finance service (Section 5.3) represents computations with many parties with separate security interests. It illustrates the use of shared state, and of reduced authority calls and authority closures to support the principle of least privilege.

5.1 The Medical Clinic

Ensuring confidentiality and integrity of medical records is a major problem for the health care industry [11]. This example illustrates the authority structure of such a system, requiring delegation, revocation, and compound tags.

Figure 4 shows part of the authority state for a medical clinic. It focuses on a particular patient, Pat. Pat is represented by principal PAT, and has a tag PAT-DATA that is applied to all records containing Pat's protected health information. Accordingly, only threads running as principals with authority for PAT-DATA can declassify and expose Pat's information – for example, to display it on the screen or send a prescription to the pharmacy.

Only doctors caring for Pat should have authority for Pat's information. This constraint is achieved by having Pat grant authority for the PAT-DATA tag to a PAT-DR principal. Then, one or more doctors (in Figure 4, just DR-BOB) can act for the PAT-DR role. Importantly, these delegations can be changed. If Pat gets a new doctor, the CLINIC-ADMIN, who also acts for PAT-DR, can allow the new doctor to act for PAT-DR. If Pat changes doctors, the CLINIC-ADMIN can

revoke the acts-for link between PAT-DR and DR-BOB, thus ensuring that DR-BOB no longer has authority for Pat's data.

In addition to providing doctors access to patient records, the medical clinic also runs a statistical analysis package periodically. This code runs as an authority closure with principal STATS, which is delegated authority for all patient tags; the code is trusted to obfuscate the result. To perform the delegation, the medical clinic uses a compound tag, ALL-PATIENT-DATA; PAT-DATA and the other patients' tags are subtags of this compound tag. Using the compound tag makes the label small (it contains just the compound tag) and allows efficient declassification (only one check is required), even though there may be thousands of patients. The delegation to STATS also does not need to be changed when new patients are added.

Discussion. The medical clinic application benefits from Aeolus's authority model. It is natural to describe the relationship between Pat, Pat's doctor, and Pat's data using principals and tags. By examining the authority structure, one can easily answer the question of who can declassify Pat's medical records. Further, constraints can be enforced, such as requiring that only doctors can act for a doctor role. In existing DIFC operating systems, authority is managed by capabilities, so these policies would require additional application-specific trusted code to control access to the appropriate declassification capability.

This application also benefits from support for revocation: removing DR-BOB's access is simply a matter of revoking a particular delegation. Once this delegation is revoked, threads running on behalf of DR-BOB, or anyone to whom he might have delegated authority, can no longer declassify Pat's data. In systems that use capabilities, revocation is more challenging, because capabilities cannot typically be revoked: threads may already be running with the revoked authority, and most DIFC operating systems also allow capabilities to be stored persistently on disk. Modern capability systems address this problem by building higher-level abstractions atop capabilities [13]; Aeolus provides these abstractions directly.

5.2 Bob and the Tax Preparer

This example, from [15], illustrates the use of distributed computation and boxes. Here, Bob is a client of an online tax preparation service. He submits his financial information to the service, along with billing information that is used to charge Bob for the service. The tax preparer then uses a proprietary database to produce Bob's tax form. There are two secrecy goals: Bob's financial information and tax form are confidential and only Bob should be able to see them; and the tax preparer's database is confidential and should not be disclosed to Bob or anyone else.

In our implementation, Bob uses a client that communicates with the tax preparer via RPC; both run on Aeolus nodes. The RPC is handled by a thread with authority for the TAX-PREP tag. Bob's client places his information in a box with secrecy label {BOB} and sends the box in an RPC to the tax preparer, along with Bob's untagged billing information. The thread in the tax preparer's virtual node that executes the RPC records Bob's billing information. Then it adds tags BOB and TAX-PREP to its secrecy label, allowing it to open the box containing Bob's financial information and read the proprietary tax database. Next, it computes Bob's tax form, uses its authority to remove the TAX-PREP tag, and returns the form to Bob.

Discussion. This example benefits from Aeolus's support for distributed computation. In particular, Bob can send an RPC with arguments that contain labeled data, and rely on the tax preparer's Aeolus runtime to ensure secrecy. Most prior DIFC systems cannot communicate sensitive information over the network. The notable exception is DStar [24], but it forces applications to periodically refresh authority, which is inconvenient.

The use of boxes is essential in this example. The tax preparer does not have authority for BOB. If it became contaminated immediately upon receiving the RPC, it could not record the billing data in a file without BOB in the label. Boxes allow the thread to avoid becoming contaminated until it actually reads Bob's sensitive information.

5.3 Financial Management Service

Our final example, which we use as a benchmark in Section 7.1, is an online personal finance management service inspired by Mint.com. Users provide the system with online banking credentials for their bank accounts, and the service aggregates their transaction histories, computes statistics, and produces a report. The application uses files and shared state to store information securely, and uses authority closures and reduced authority calls to run code with minimal authority. Users and banks are external entities in this system, and do not run the Aeolus platform; all communication with these entities must thus be done with null labels.

There are several clear security requirements. A user's financial data must not be exposed to other users or third-party banks. A user's online banking credentials (username and password) are even more sensitive: they should not be used for any purpose other than to log in to the corresponding bank. They should not even be revealed to the user, in case the user's access to the site is compromised.

To capture these constraints, each user has a separate tag, e.g., ALICE-DATA and a principal, e.g., ALICE, that is authoritative for this tag. These tags are subtags of the ALL-USER-DATA compound tag. There is also a principal for each bank, and an associated tag that is used to protect users' credentials for that bank.

A user's financial data is stored in a file with the appropriate USER-DATA tag in its secrecy label. Each of a user's bank credentials is stored in its own file, with a secrecy label containing both the user's tag and the appropriate BANK tag; this way even the user can't expose the credentials.

Because a user's session might consist of many requests, the service also caches information for active users in a shared hash table that maps user IDs to session state for that user. Within a session-state object are boxes containing bank credentials for each of that user's banks. The labels here mirror those of the files: the secrecy label of the session-state object contains the USER-DATA tag, while each box has both the USER-DATA and the BANK tags in its secrecy label.

When Alice requests to display her recent transactions, the request is received (over a secure channel) by a thread that is authoritative for all users. After validating the user's identity, it makes a reduced authority call, dropping its authority to the ALICE principal.

The thread then locates the session-state object containing her information (assuming it is in the cache). Reading Alice's session-state object requires adding the ALICE-DATA tag to the thread's secrecy label. If a bug in the code caused the thread to read a *different* user's information, it will not be able to remove the corresponding tag and thus cannot leak it outside the system.

The thread then calls an authority closure for each of Alice's banks. This closure takes the box containing Alice's credentials for that bank, and obtains and returns a list of Alice's transactions from the bank; a sketch of the closure code is shown in Figure 5. The closure runs with authority for both its BANK tag and the ALL-USER-DATA tag. It adds the BANK tag to the thread's secrecy label, opens the box to obtain the credentials, uses its authority to declassify, and contacts the remote bank. When the closure returns Alice's transactions, its label is merged with that of its caller, so the user's tag, e.g., ALICE-DATA, is restored to the label. Therefore, we need not be concerned that Alice's data will be exposed to some other user.

After obtaining information from all banks, the thread computes the result. Since this involves invoking untrusted code – for example, an OFX parser or a graph-drawing library – it does so using a reduced authority call to P_{PUBLIC}. Because the thread's label contains the ALICE-DATA tag, the called code can process her financial data but cannot expose it over the network or store it in an unlabeled file. When the reduced authority call returns, the thread removes the ALICE-DATA tag from its secrecy label and returns the result to Alice.

Discussion. Aeolus' reduced authority calls and authority closures make it convenient to run application components with minimal authority, so only a small amount of application code needs to be trusted for security. For example, to ensure the secrecy of a user's bank password, we need only trust the bank closure, as only it runs with the authority to remove the bank's tag. In previous DIFC operating systems, dropping and regaining authority requires use of a different process, which is more difficult to program and imposes higher overhead – making it likely that developers will not bother to do so.

```
public class BankClosure extends AeolusClosure {
    private BankInfo bank;
    private Tag bankTag; // the tag for this bank

    public BankClosure(PID bankPID, Tag t, BankInfo b) {
      super(bankPID); // binds the closure to bankPID
      bankTag = t;
      bank = b;
    }

    // this function runs with bankPID authority
    public Transactions getTxns(Box<Credentials> u) {
        addSecrecy(this.bankTag);
        Credentials c = u.get()
        declassify(this.bankTag);
        declassify(c.userTag);
        Transactions t = download(bank.url,
                                  c.username, c.password);
        return t;
    }
}
```

Figure 5: Example of an authority closure

The application also benefits from shared state since it can cache data in memory rather than resorting to files. Aeolus is the first DIFC system to support secure shared state with labeled application-level objects. An application running on an existing DIFC operating systems could implement a "shared state manager" process that other processes communicate with by IPC, but this is inconvenient, expensive, and risky: that process needs to run with complete authority because it is contaminated by the content of all shared objects.

6 Implementation

This section describes some highlights of our Java-based implementation of Aeolus. Further details, including the complete interface, are available in the Aeolus reference manual [1]. We also have implementations for C# and PHP, but do not describe these here.

Figure 6 shows the structure of the implementation at an Aeolus node. Aeolus runs on top of the JVM within a single OS process, and all Aeolus applications on the node run within the same JVM. The Aeolus runtime runs a special thread (the authority state client) that manages an authority state cache and handles interactions with the authority state server (the AS) on behalf of all threads at the node. Application code accesses Aeolus features via a set of libraries. The AS runs on a separate machine.

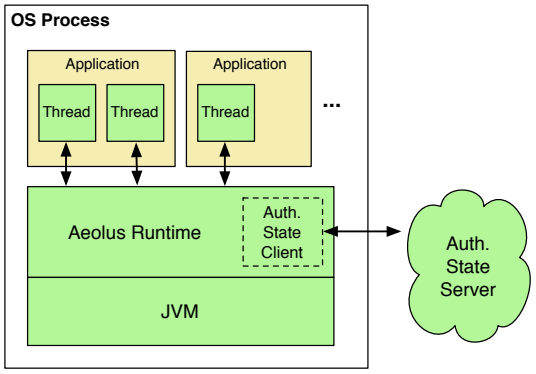

Figure 6: Structure of an Aeolus node

6.1 Threads and Isolation

All Aeolus threads on a node, even those belonging to different virtual nodes, run within the same JVM. All threads and shared state objects share a common heap.

Running all threads in the same JVM provides good performance: for example, it allows us to support fast shared state communication between threads. However, each Aeolus thread must appear to have its own isolated memory, and label checks must be interposed on accesses to shared state and boxes. Aeolus achieves this isolation goal by relying on the memory safety of the JVM: each thread can access only the parts of the heap accessible from its stack. Aeolus ensures that only one thread can have a pointer to any object, except for encapsulated shared objects.

Implementing Copy-Based Isolation. To ensure that only one thread can have a pointer to a non-shared object, objects must be copied before they can be passed to a new thread. Similarly, arguments to and results from shared object methods must be copied to prevent threads from obtaining pointers to the contents of encapsulated shared objects. These deep copies can be expensive; prior work rejected copying entirely, citing prohibitive costs [12]. Aeolus uses a highly optimized, low-level cloning library to reduce the cost of copying.

An important optimization is to avoid copying immutable state, which can be safely shared. Aeolus recursively analyzes classes at load time to determine whether instances could contain mutable (non-final) fields or can refer to any mutable objects via fields, superclasses, and inner and outer classes. For the purposes of this analysis, shared state objects are considered immutable; as we describe below, they are instrumented to perform runtime label checks, so references to them can be shared.

Programmers can use the *AeolusSafe* marker interface to indicate that a type is immutable. If an AeolusSafe type contains or refers to mutable state, a load-time error occurs. Aeolus can safely assume that all subtypes of an AeolusSafe type are immutable. If a type is AeolusSafe (or if it is immutable and final) then the instrumentation to copy values of that type is omitted entirely.

Restrictions. Aeolus requires application code to use a "safe" subset of Java, because certain Java features could be used to circumvent isolation and the information flow rules. Aeolus applications may not use:

- reflection and native code, which can bypass the Java type system
- static variables, which could allow threads to share state in ways that violate the information flow rules[1]
- direct access to underlying Java APIs for I/O and threading; applications must use the Aeolus API

The restrictions are enforced through a combination of load-time bytecode verification implemented with a custom class loader, and runtime checks implemented using Java's SecurityManager framework. Our approach to isolation is similar to that used by other systems that run multiple applications in a single JVM [3, 12].

6.2 Shared Objects and Closures

The Aeolus class loader uses bytecode rewriting to make shared objects and closures safe, convenient, and efficient. Shared state objects and closures can provide any number of public methods; the Aeolus class loader adds instrumentation to them to enforce the information flow rules. Closure methods are instrumented to save the caller's labels and merge them into the thread labels on return. Shared state objects are instrumented to perform the necessary label checks, and also to copy arguments and return values of method and constructor calls (including any thrown exception objects) if necessary to enforce isolation.

Closure objects are required to be immutable (subtypes of AeolusSafe). This prevents closures from keeping state between calls and potentially leaking information to differently labeled callers.

6.3 Management of Authority State

Authority state is stored in a transactional database at a special *authority server* (AS). Each node runs an *authority state client*, which sends all authority updates to the AS. Updates are infrequent, and thus result in minimal overhead, but queries over the authority state are common. The authority state client maintains a local cache of authority state to reduce query latency and AS load.

When there is a miss in the authority cache, Aeolus fetches a *block* of related information. Since the notion of what is "related" depends on the application (e.g., if we need information about a patient in a medical system,

[1] We plan to remove this restriction in the future by providing private versions of static variables for each thread. The technique is similar to that proposed for Java's as-yet-unavailable isolation API [3].

what other information is useful to know?), we provide a way for applications to organize the state into *blocks*.

The cache needs to be managed properly to ensure that it contains correct and timely information. When a block is fetched, the cache also receives and applies all updates that have occurred since its last communication with the AS. This ensures that each query runs on a consistent snapshot of the AS state. In addition, we provide causality by propagating information about the most recent AS update at the sender in its messages; at the receiver, we ensure that the next use of the cache reflects this update or a later state. Finally we provide timeliness, which is especially important for revocation. A cache stops processing queries until it can communicate with the AS if its update information is older than δ seconds; δ is a system parameter and might be on the order of 30 seconds.

The Query Cache. Even if all necessary blocks are present in the cache, determining whether a principal has authority for a certain tag might be computationally expensive if the application has a complex authority state. To avoid this overhead, we also maintain a *query cache*, which stores the results of recent queries. The query cache contains a set of pairs; each indicating that a particular principal has authority for a specific tag, or acts for a specific other principal.

When a client receives updates from the AS, it removes all query cache entries computed from information in blocks that have changed. This approach reduces the metadata needed for each query-cache entry (we store information about blocks rather than specific delegation links) but can lead to unnecessary evictions.

7 Evaluation: Performance

This section demonstrates that a large application running on Aeolus performs about as well as an application running on pure Java that does an equivalent amount of work. We first examine the end-to-end performance of an implementation of the financial management service from Section 5.3, and show that the overhead of adding information flow control with Aeolus is minuscule. To gain further insight into this result, we then explore the sources of overhead, via microbenchmarks.

A key contributor to Aeolus's low overhead is that it tracks information flows at the level of threads, relying on memory safety for isolation. Therefore, protection boundary crossings are much cheaper than in information flow systems that rely on OS processes for isolation [6, 9, 23]. Furthermore, execution is cheaper than in systems that do finer-grained tracking [16, 22] because Aeolus interposes only on communication and not on individual memory accesses. Some of our optimizations, particularly authority query caching, fast copies, and immutability analysis, also contribute substantially to our performance results.

These experiments use our Java implementation of Aeolus on a 2.50 GHz Core 2 Quad system with 4 GB of RAM, running Ubuntu 11.04 with Linux kernel 2.6.38-10. Our Aeolus libraries ran atop the OpenJDK 1.7.0_1 JVM.

7.1 End-to-End Performance

The evaluation of the financial management service prototype is interesting because it tests the performance of a wide range of Aeolus features: the code does label manipulations, makes use of shared state, and uses both authority closures and reduced authority calls. The prototype operates as a single virtual node that receives requests from users, fetches their user information, and retrieves financial data using bank closures. In the experiment, each user has three banks, and we use ten threads to handle user requests (i.e., we can run ten requests at a time). Each bank closure establishes an SSL connection to a user's bank, downloads the user's bank statement in OFX format, and returns the result; lacking real banks to test with, we simulate the network delay by sleeping for 100 ms. When all bank information has arrived, the thread generates a graph of spending habits, using a reduced authority call.

We compared the average request processing time for the implementation of this benchmark to one that does not use Aeolus or DIFC. The Aeolus version required 323.9 ms processing per request, 0.15% greater than the 323.5 ms processing time for the native version. There was a corresponding decrease in throughput from 17.97 req/s to 17.61 req/s. We observed similar overhead when varying the amount of time taken to process financial information retrieved from each bank.

7.2 Microbenchmarks

Applications running on Aeolus have low overhead, as shown above, because they invoke security operations relatively infrequently and mostly do real work. Nevertheless, it is worthwhile to examine the sources of overhead.

Shared State and Copying. Aeolus must check the caller's labels on each call to a method of a shared state object, but this overhead is negligible because labels are small (typically they contain one tag), making shared state a viable option for communication between threads. A trivial call with an empty label and no arguments or return value costs 8.9 ns.

The cost of copying arguments and return values of shared state methods is more significant, however. As discussed in Section 6.1, Aeolus avoids copying these objects if they are immutable; it takes 13 ns to make this determination by looking up the object's type in a hash table. If a formal parameter to a method is known to be immutable at load time (i.e., it is a primitive type, a

Operation	Time (ns)
Reduced authority call to P_{PUBLIC}	7.7
Reduced authority call to P_x	51
Closure call	83
Java method call	4.0

Figure 7: Time to perform a reduced authority call or invoke a closure. The cost of an ordinary Java method call is provided for comparison.

final and immutable class, or a shared or AeolusSafe object), then the copying instrumentation is omitted entirely, reducing the overhead for that argument to zero.

For objects that must be copied, Aeolus uses a fast, low-level mechanism that takes about 93 ns to duplicate an object, plus the time to recursively duplicate its fields. This offers significantly greater performance than the typical implementation of cloning, which serializes the object into a buffer and then deserializes the buffer; this naive approach takes 6.3 μs to copy an empty object, much of it spent on needless validity checks. Most complex objects contain at least some state that is immutable, e.g., Strings, so our optimization of avoiding copies of these fields is also significant.

We found that copying array elements was particularly slow (200 ns). Arrays of immutable elements (such as bytes, integers, or Strings) are common, so we made them a special case. Using `System.arraycopy` reduced the per-element copying cost to 1.5 ns for an array of immutable objects.

Closure and Reduced Authority Calls. The table in Figure 7 shows the overhead involved in performing reduced authority calls and closure calls when the process label is empty and there are no arguments or results. Calls to P_{PUBLIC} incur less overhead because they need not check authority state.

Forks. Aeolus can fork a new thread running with different authority and start running in the new thread in 12 μs. Tracking security state and enforcing isolation adds a small overhead relative to the 4.7 μs it takes to start a thread in Java. (Thread pools are used as an optimization in both cases.) Because Aeolus implements isolation using threads, these figures are much smaller than the cost to fork an OS process, the equivalent operation in a DIFC operating system. For comparison, creating a new OS-level process on the same machine requires 135 μs.

Authority Management. Aeolus adds overhead in the management of authority state. In particular, a thread can only perform a declassification or endorsement on a tag if its principal has authority for that tag. In applications with complex authority structure, checking this may require traversing a number of delegation links.

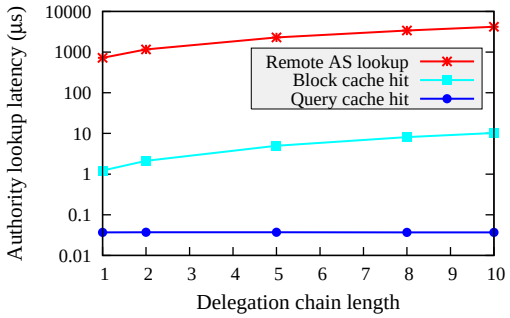

Figure 8: Authority lookup cost, by cache level used. Note the logarithmic y-axis.

Our use of a two-level cache mitigates this cost. Figure 8 shows the average request latency for an authority check, with varying delegation chain lengths. Depending on the length of the chain, it takes 0.7 to 4.2 ms to fetch the relevant state from the authority server to answer a query. If the state is already in the block cache, the answer can be computed locally in 1 to 10 μs. If the answer is in the query cache, it takes only 37 ns, regardless of the length of delegation chains. Here, the authority server is located on the same local network as the client. In a wider-area deployment, latency for uncached requests would be higher and the cache's benefit would be even greater.

File I/O. Aeolus must interpose on accesses to the file system to enforce information flow control. Because it must load and check the file's label, Aeolus imposes an additional cost of 400 μs on the first read to a particular file. Subsequent requests are cached.

8 Related Work

Aeolus builds on work on DIFC in programming languages and operating systems, fusing the concepts developed in these two areas into a new high-level model. It also provides new mechanisms not found in other systems, such as closures, boxes, and shared volatile state.

8.1 Programming Languages

Jif [14, 15] introduced the DIFC model, which formed the basis for many subsequent systems, including Aeolus. Jif embeds information flow policies in labels and makes labels part of the type system; programs that violate the information flow rules are rejected by the compiler. Jif also supports certain kinds of dynamic policies. Aeolus, in contrast, does all label checks at runtime and does not require type annotations.

Some language-based approaches [2, 19, 20] use the concept of a dynamic principal hierarchy to specify security policies, and support precise fine-grained declassification. Aeolus's principal hierarchy draws inspiration

from this work. However, tags allow us to group objects with the same security policy in a more convenient way than is possible in the type-system-based approaches. Our model also operates at a coarser granularity, allowing programmers to focus on the privileges required for different modules of an application, rather than the sensitivity of individual variables and objects.

Fabric [10] adds trust relationships to the principal hierarchy, supporting federated systems with mutually distrustful nodes. Aeolus assumes all nodes in a deployment are trusted, but could be extended to use this approach. Fabric nodes cache the transitive closure of all the acts-for links they know about; similarly, Aeolus's query cache stores edges in the transitive closure of the authority graph, but Aeolus doesn't precompute the entire transitive closure.

Another significant difference between Aeolus and previous language-based approaches is that Aeolus does not require applications to be written in a new language, nor does it require special compiler support.

8.2 Operating Systems

DIFC-based operating systems expose information flow controls to the applications via the operating system API. Asbestos [6] and HiStar [23] are new operating systems that provide DIFC properties using labels and tags. Asbestos tracks information flow at the level of processes exchanging unreliable messages; HiStar acts at the microkernel level of threads, memory segments, and gates (which are somewhat like our authority closures).

Aeolus borrows the notion of tags and labels from this recent work. However, labels in Asbestos and HiStar combine mechanisms for privacy, integrity, authentication, declassification privilege, and access control. Flume [9], like Aeolus, separates information flow labels from authority and access control to make labels easier to understand. Also like Aeolus, Flume runs in user mode on a standard OS. However, Aeolus differs from Flume because it uses an explicit principal hierarchy rather than capabilities.

Aeolus also exposes higher-level abstractions to the application, such as closures and boxes. Whereas Asbestos, HiStar, and Flume require separate processes for privilege separation, Aeolus's abstractions make defining trust boundaries within an application easier and more efficient.

Among these systems, only HiStar provides support for shared memory, by exposing low-level page table protection mechanisms. Aeolus provides more usable sharing at the level of objects applications use. Laminar [16] also aims to provide sharing of application objects, using a hybrid of OS and JVM mechanisms. Laminar tracks information flow at object granularity; however, because fine-grained dynamic tracking is expensive, it only tracks contamination within code blocks called *security regions*. It does not track contamination outside these regions, making it hard to enforce end-to-end security guarantees.

DStar [24] extends HiStar over the network and is the only DIFC system that provides support for revocation. It does this by requiring authorizations to be refreshed periodically. Aeolus uses a similar technique internally but provides a more convenient abstraction to users.

Aeolus applications enforce discretionary information flow policies through carefully controlled use of authority – a notion programmers are familiar with. Other systems have explored alternatives such as data-flow assertions [22] and special-purpose policy specification languages [5].

The Singularity operating system [7], while not an information flow system, supports language-based isolation like Aeolus. Singularity provides fast IPC via hand-off: threads can exchange shared objects, but at most one thread can have a reference to a given shared object at any time. In contrast, Aeolus shared objects can be accessed by many threads concurrently, and Aeolus uses copying to prevent direct references from crossing the boundary between threads and shared objects.

9 Conclusions and Future Work

Aeolus is a new distributed platform for developing and deploying secure applications using DIFC. It provides a new security model based on tags and a principal hierarchy, which allows fine-grained delegation and revocation of authority. Recording trust relationships in a principal hierarchy makes it possible to define policy constraints.

Aeolus provides abstractions to make writing secure applications easier. Authority closures and reduced authority calls support the principle of least privilege. In addition, Aeolus supports distributed and concurrent programs, providing boxes to limit contamination, IPC, and shared state to allow convenient yet safe use of shared volatile information.

We implemented Aeolus in Java, and used it to develop several applications. Aeolus's features made expressing the desired information flow constraints convenient. Our experiments show that Aeolus has good performance, and its approach for caching authority state is effective. An initial release of the system can be downloaded from `http://pmg.csail.mit.edu/aeolus/`.

We are investigating programming language extensions to make Aeolus even more convenient. For example, it would be useful to have a syntactic extension for running blocks of code with reduced authority, eliminating the need to make reduced authority calls to separate methods. Also, if it were possible to recognize that a method on a shared object did not perform any mutation operations, either directly or indirectly, we could allow user-defined shared objects with relaxed label restrictions; presently, Aeolus conservatively assumes that shared object meth-

ods both read and write the object.

Additionally, we are currently extending the system with support for secure audit trails, and we are developing an approach to integrate databases into the model in a flexible way.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Eddie Kohler, for their helpful feedback. This research was supported by the National Science Foundation under grant CNS-0834239 and by funding from Northrop Grumman Corporation.

References

[1] Aeolus reference manual. Available at http://pmg.csail.mit.edu/aeolus-ref.pdf.

[2] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *USENIX Security 2007*, Boston, MA, Aug. 2007.

[3] G. Czajkowski. Application isolation in the Java virtual machine. In *OOPSLA '00*, Minneapolis, MN, Oct. 2000.

[4] D. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[5] P. Efstathopoulos and E. Kohler. Managable fine-grained information flow. In *EuroSys '08*, Glasgow, UK, Apr. 2008.

[6] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP '05*, Brighton, UK, Oct. 2005.

[7] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys '06*, Leuven, Belgium, Apr. 2006.

[8] J. A. Hall and S. L. Liedtka. The Sarbanes-Oxley act: implications for large-scale IT outsourcing. *Commun. ACM*, 50(3):95–100, 2007.

[9] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP '07*, Stevenson, WA, Oct. 2007.

[10] J. Liu, M. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *SOSP '09*, Big Sky, MT, Oct. 2009.

[11] R. T. Mercuri. The HIPAA-potamus in health care data security. *Commun. ACM*, 47(7):25–28, 2004.

[12] M. Migliavacca, I. Papagiannis, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch. DEFCON: High-performance event processing with information security. In *USENIX '10*, Boston, MA, June 2010.

[13] M. S. Miller and J. S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *ASIAN '03*, Mumbai, India, Dec. 2003.

[14] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL 1999*, San Antonio, TX, Jan. 1999.

[15] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP '97*, Saint Malo, France, Oct. 1997.

[16] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *PLDI '09*, Dublin, Ireland, June 2009.

[17] A. Sabelfeld and A. C. Myers. Language-based information flow security. In *IEEE JSAC*, volume 21, Jan. 2003.

[18] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *SOSP '73*, Yorktown Heights, NY, Oct. 1973.

[19] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *CSFW '06*.

[20] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. *ACM Trans. Program. Lang. Syst.*, 30(1):6, 2007.

[21] U.S. Department of Defense Computer Security Center. Trusted computer system evaluation criteria. DoD 5200.28-STD, Dec. 1985.

[22] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP '09*, Big Sky, MT, Oct. 2009.

[23] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI 2006*, Seattle, WA, Nov. 2006.

[24] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *NSDI 2008*, San Francisco, CA, Apr. 2008.