Manageable Fine-Grained Information Flow

Petros Efstathopoulos and Eddie Kohler
UCLA Computer Science Department
{pefstath, kohler}@cs.ucla.edu

ABSTRACT

The continuing frequency and seriousness of security incidents underlines the importance of application security. Decentralized information flow control (DIFC), a promising tool for improving application security, gives application developers fine-grained control over security policy and privilege management. DIFC developers can partition much application functionality into untrusted components bound by a kernel- or language-enforced security policy. Unless a (usually smaller and less exposed) trusted component is exploited, the effects of an application compromise are contained by the policy.

Although system-based DIFC can simultaneously achieve high performance and effective isolation, it offers a challenging programming model. Fine-grained policy specifications are spread over several application pieces. Common programming errors may be indistinguishable from policy exploit attempts; the system cannot expose developers to information about these errors, complicating debugging. Static checking (as in language-based DIFC) and new system primitives can reduce these problems, but for dynamic applications like web servers, they do not eliminate them.

In this paper we propose subsystems that make decentralized information flow more manageable. First, a *policy description language* specifies an application-wide security policy in one localized place; communication restrictions are compiled into lower-level labels. Second, information flow-safe *debugging mechanisms* let developers debug DIFC applications without violating security policies. Although these mechanisms are preliminary, we demonstrate their effectiveness using applications similar to those developed for Asbestos and other DIFC systems.

Categories and Subject Descriptors:
D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls, Access controls*; D.2.5 [**Testing and Debugging**]: Debugging Aids

General Terms: Security, Design, Management

Keywords: decentralized information flow control, labels, policy language, debugging

1 INTRODUCTION

Information flow control, or IFC, improves system security by enforcing mandatory policy restrictions. Bugs outside the trusted security kernel cannot violate the information flow policy [6, 7, 9, 15]. IFC's *label* formalism can implement security policies such as secrecy protection (preventing protected information from escaping a system) and integrity protection (preventing external information from corrupting a system). Classical centralized IFC concentrates all *privilege*—defined as the right to relabel information independent of IFC policy—in the security kernel; all other subsystems are completely constrained. This has security benefits, but important application policies often require a form of privilege. For example, consider a web server that responds to requests from different application-defined users. The part of the web server that parses user passwords is necessarily trusted by all users. However, we might like to constrain other parts by a secrecy policy to prevent large-scale password theft. A truly centralized IFC system would seem to require either including the password parser in the system-wide security kernel, or leaving user passwords unprotected.

Decentralized information flow control, or DIFC, addresses this problem by decentralizing the notion of privilege [8, 11, 12, 14, 16]. No special privilege is required to create a new security policy; code is privileged with respect to the policies it creates, while remaining constrained by other policies' information flow rules. This allows applications to split themselves into privileged and unprivileged pieces, and brings the security benefits of information flow control to challenging applications like servers. While a conventionally-designed application occupies a single security domain—a bug anywhere in the application can provide access to the application's full rights—the unprivileged parts of a DIFC application execute in restricted domains, and are thus less security critical.

Unfortunately, privileged application code must still create the relevant security policies by manipulating labels, and any application must be debugged. In our experience with Asbestos's system-based DIFC [14], these management problems are difficult enough to hamper adoption. Labels concisely express an application's information flow constraints and privileges, but a label, which combines the effects of *all* policies active on a process, is created piecemeal using per-policy primitives like "transfer privilege" or "selectively mark a message as secret." These primitives are spread throughout the code—in a privilege-separated application, most communication crosses security domains. This diffuses the individual policies and obscures their combined effect. Policy debugging is also daunting. A mistake in policy definition or implementation often causes a process to have less privilege than it needs. When it attempts to exercise this nonexistent privilege, the system sees an attempted security policy violation indistinguishable from an actual exploit. Debugging such a problem requires extracting information from the process, but the bug itself prevents the process from exporting this information: all process state is subject to information flow rules.

In this paper, we describe enhancements to Asbestos that make privilege-separated applications easier to design, build, and debug. First, a new *policy description language* defines label-based security policies via allowed process communication patterns ("*A* can communicate with *B*, but not *C*"). This concentrates policy specification into one place, making policies easier to write and to reason about. A parser translates the policy into labels, and optionally launches applications with the labels already in place. The language can specify even complex application requirements, such as

Asbestos event processes (a process abstraction that combines isolation and low memory overhead). Although communication patterns are not perfect abstractions for information flow—for example, non-transitive communication patterns have no mandatorily-enforced Asbestos equivalents—they improve programmability for our target applications, and provide a useful point for further research. Second, debugging is supported by *debug domains*, which safely extend the notion of privilege to include application debugging. When a debug domain is given privilege for a given policy, problems involving that policy may be forwarded to a separate debugging process, no matter where those problems occur. Debug domains combine information flow safety—debug domains do not expose information they aren't allowed to see—with usability. Our evaluation shows that the development of previously-proposed DIFC applications is simplified by our system management tools.

This work represents one of several approaches to improving DIFC's manageability. The static checking characteristic of language-based DIFC [4, 11, 12] simplifies application development by reporting most errors at compile time. However, policies are still spread throughout the application; the development process is complicated by requirements to design information flow-safe abstractions and to avoid unsafe idioms; and run-time checks required for dynamic server applications can reintroduce manageability problems [4]. A localized policy, possibly combined with a more extensive label inference, might improve manageability. On the other hand, Asbestos's successors [8, 16] reduce the classes of errors that cannot be reported to applications, but some errors remain unsafe to report, and our debugging primitives would still benefit programmers by collecting all problem reports in one place. A fuller DIFC system would combine aspects of all these.

The rest of this paper is organized as follows: Section 2 presents related work while Section 3 briefly describes some of the properties of Asbestos. Section 4 presents our policy description language and its implementation. Section 5 introduces our approach to debugging. Section 6 presents our experiences with the proposed mechanisms, while Section 7 summarizes our plans for future work.

2 RELATED WORK

Many operating systems that implement Mandatory Access Control (MAC) have used label variants [6] to enforce security policies, including variants of mainstream operating systems such as SELinux [9] and FreeBSD [15], and research operating systems like Asbestos [14] and HiStar [16]. The strict isolation enforced by these systems introduces system management challenges such as those addressed here.

SELinux and TrustedBSD are two of the most popular operating systems that support multilevel security (MLS). They are both based on implementations of the Flask MAC architecture [13], which employs a special kernel component called the Security Server (SS). Policies in these systems are defined in multiple files using a policy description language that is compiled to a binary format before it is passed on to the SS for enforcement. These policies have system-wide effect and are centrally managed by security officers that are able to dynamically insert them into the kernel. Debugging in this context is performed mainly through privileged access to special system files (such as *sysfs* files) and the console. This is very different from the decentralized IFC implemented by Asbestos, where untrusted application developers may create and manage their own set of compartments and implement policies that will have to be enforced (by the kernel) within the context of the application. This work attempts to facilitate system management tasks such as policy description and debugging without requiring extra system privilege.

The system management challenges we faced while developing Asbestos applications such as the Asbestos web server [14] were the main motivation for our work. HiStar mitigates some programming issues by shifting the responsibility for most process label changes to the process itself. This lets HiStar safely report most errors to the calling application. However, errors are still generated by widely separated application fragments, and policy specification management remains a challenge. Although HiStar implements an untrusted, user-level Unix emulation layer, applications running on this layer are subject to Unix-type security policies; introducing more complex policies requires interacting with HiStar abstractions. For instance, running the ClamAV anti-virus application in a HiStar isolated domain involves a special wrapper/launcher that implements the necessary label initialization. Our work attempts to provide a general solution to this wrapper/launcher problem, allowing developers to express even complex security policies at a higher level.

The Flume system [8] implements decentralized IFC for Linux using a reference monitor. Flume abstractions go beyond HiStar's in supporting management. The promise of the system is to provide IFC guarantees for Linux applications without requiring extensive changes to application code. A wiki application using Flume was implemented using a separate launcher application module for policy initialization. A policy language like the one we present could replace such purpose-built launchers.

Jif [12] and JFlow [11] provide language level information flow control by annotating (labeling) source code at the granularity of variables and functions. These systems allow developers to express very fine-grained policies through widespread code annotations, whereas Asbestos labels enforce policies at the process level. Our policy framework uses communication-based annotations, as opposed to label annotations, to define policies in one piece, minimizing the dispersion of trusted policy implementation code. Jif policy checking and debugging is largely performed statically at compile time, and the generated code is guaranteed to comply with the policy. This simplifies debugging. However, more complex server-type applications, such as those in the more recent Jif projects SIF [4] and the Swift framework [3], involve dynamic decisions by design, and SIF and Swift allow for the dynamic creation of objects that are labeled appropriately to adhere to the application policy. Debugging the resulting runtime behavior would seem to face similar management challenges, and could benefit from ideas like our debug domains.

3 DIFC MANAGEMENT CHALLENGES

In this section, we use a concrete example derived from the Asbestos web server (AWS) [14] architecture to further motivate the system management challenges addressed in this work, and briefly introduce the Asbestos system.

Defining a DIFC policy involves deciding on the allowed and forbidden information flows among application modules and the rest of the system, identifying the amount of privilege each operation requires, and granting privilege to application modules accordingly. It also often involves specifying a policy for dynamically changing application elements, such as new processes created as users log in and out of a server. Correctly defining and implementing the application policy is of critical importance. With a too-broad policy, most of the application has privilege, and the benefits of DIFC are not achieved; with a too-narrow policy, an application will generally not function (e.g., system components will not be able to communicate as needed).

Figure 1 presents a policy similar to that of the Asbestos web server, which provides multiple users with services implemented

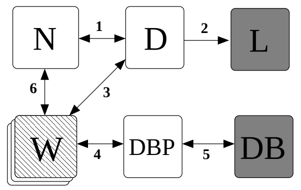

Figure 1: A representation of the explicit communication requirements in an AWS-like policy. Arrows represent expected communication patterns; single arrows denote one-way communication. In the absence of arrows, a per-component default communication rule applies: white components (N, D, DBP) are able to freely send and receive information by default, lined components (W) are "receive only" by default, and shaded components (L, DB) are isolated, unable to send or receive by default.

by isolated *worker processes*. Each box in the figure represents a labeled application component, and arrows represent communication between components required for the application to be secure and functional. The most important aspect of the policy in this example is the requirement for confidentiality: a user's information must not escape to some other user, even in the presence of bugs in worker processes. (Most of the figure's other processes are part of the server's trusted computing base.) We examine how each of the application processes should (not) communicate with the rest of the system, and define the application policy accordingly.

The N process is the network daemon, a system process with permission to access the network card. N is considered trusted by any application accessing the network. When a new user connects to the service through the network daemon, the connection is forwarded by N to the *demux* process D (arrow 1 in Figure 1). This process examines the connection, including possibly a username and password, and associates the connection with a corresponding user. As a result, the demux, like the network daemon, is trusted.

After D has identified the incoming user (e.g. Alice), it notifies N of the user's identity. This allows N to mark the corresponding connection with Alice's identity, which both marks incoming data as Alice-confidential and allows Alice-confidential data to escape on that connection. Then D informs the system logger L of the new connection. For security reasons, L is deliberately isolated and only needs to receive information from D (arrow 2).

Once the connection has been logged, D forwards the request to one of the untrusted *worker* processes W for execution. W thus resembles a CGI script. W can access Alice's data through the database front-end DBP. In order to enforce our confidentiality policy and prevent Alice's data from leaking—even if W is buggy—we require that W can send information *only* to components that are privileged to receive data belonging to Alice, such as N, D, and the database front end DBP. Given that information flowing towards W does not pose any threat to our confidentiality policy, we need to ensure that W may have two-way communication with N, D, and DBP (arrows 3, 4 and 6) and receive-only communication otherwise.

The database front end DBP is responsible for marshalling all requests going in and out of the database: it sanitizes requests, checks privilege for writes, and labels, or *contaminates*, all outgoing data according to DIFC requirements. Since DBP handles all user data it is considered a trusted component. In our application DBP needs to have two-way communication with W and the database DB (arrows 4 and 5), but there is no fundamental policy reason to restrict DBP's communication with other processes—given that it is a trusted component, anyone needs to interface with in order to access the database.

To ensure that all database accesses are marshalled we need to isolate the database (DB) from the rest of the system, making it accessible only through DBP.

An attempt by Alice to obtain Bob's data through DBP will fail since DBP marks Bob's data in a way that Alice's worker cannot receive. Similarly, if Alice's worker is buggy and tries to leak Alice's information through Bob's network connection, the attempt will fail since D has arranged that Bob's network connection can export Bob-confidential data, but not Alice-confidential data.

The arrows in Figure 1 represent the expected communication patterns among application components. A complete policy must also model the communication behavior of each component with any processes not mentioned explicitly in the diagram. A simple such model is a *default* rule that applies to any process pair not explicitly mentioned. Since N, D, and DBP are trusted components, their communication behavior does not need to be limited by default in this high-level policy. (These processes may, and do, limit their own communication behavior to enforce finer-grained policies—for instance, N may restrict access to certain communication endpoints, requiring privilege to send information over to them.) This is represented in Figure 1 by placing N, D, and DBP in white boxes. In the case of L and DB an "isolated" default applies, represented by the shaded box; these processes should be prevented from communicating with other unprivileged processes. In the case of W a "receive-only" default applies, represented by the box with diagonal lines. A "send-only" default rule is also possible. In the absence of relevant explicit rules, the communication between two processes with contrasting defaults is governed by the most restrictive default. For instance, although D's default rule is "unrestricted communication", D may not communicate with DB, since DB's default rule is "isolated".

The IPC analogy underlying communication constraint diagrams like Figure 1 is easy to understand. However, a direct implementation of such a diagram might generalize poorly to communication over shared resources, such as files, or to processes created by processes in the diagram. Information flow naturally generalizes communication constraints to *any* flow of information, not just IPC. The desired behavior is expressed in terms of process and object labels, which naturally track information through non-IPC channels like the file system. For example, a file created by DB should not be directly readable by processes other than DB and D (and possibly helper processes with the same labels); in fact, other processes should not even be able to discover the new file's existence.

Figure 1 can be translated into decentralized information flow labels. For example, labels can prevent L from sending information to other processes: L becomes "higher secrecy" than the other processes in the diagram. Some aspects of communication diagrams have no exact information flow equivalents. (For example, a full implementation of the communication pattern W $\leftrightarrow$ DBP $\leftrightarrow$ DB would prevent W from contacting DB directly independent of DBP's behavior, but Asbestos-like information flow cannot completely implement this constraint. If W can send to DB via a proxy DBP, then DBP must be sufficiently privileged that it could grant W the right to send to DB directly.) Nevertheless, communication patterns are a useful starting point for investigating simplified specifications of information flow policies.

Unfortunately, the label translation of Figure 1 is not trivial: initializing the corresponding processes requires 20 label operations in Asbestos. The issue is that a communication pattern like Figure 1 corresponds to several *interacting* information flow policies, requiring separate privilege domains and privilege manipulations. For instance, the relationship between W, DBP, and DB requires policies that (1) prevent W from sending data to any outside process, but (2) allow W to communicate with DBP, (3) prevent DB

Process	Tracking label $\mathbf{T}_X$	Clearance label $\mathbf{C}_X$
N	$\{w\star, \mathbf{1}\}$	$\{w\mathbf{3}, \mathbf{2}\}$
D	$\{l'\star, w\star, \mathbf{1}\}$	$\{w\mathbf{3}, \mathbf{2}\}$
L	$\{l\mathbf{3}, l'\star, \mathbf{1}\}$	$\{l\mathbf{3}, l'\mathbf{0}, \mathbf{2}\}$
W	$\{w\mathbf{3}, \mathbf{1}\}$	$\{w\mathbf{3}, \mathbf{2}\}$
DBP	$\{w\star, db\star, db'\star, \mathbf{1}\}$	$\{w\mathbf{3}, db\mathbf{3}, \mathbf{2}\}$
DB	$\{db\mathbf{3}, db'\star, \mathbf{1}\}$	$\{db\mathbf{3}, db'\mathbf{0}, \mathbf{2}\}$

Figure 2: Asbestos labels implementing the policy of Figure 1.

from communicating with any outside process, and (4) allow DBP to communicate with DB. Implementing this requires at least two different kinds of contamination and the corresponding privilege. First, DB is contaminated to prevent its communication with outside processes; however, DBP may remove this contamination, and must thus hold the corresponding privilege. Second, W is also contaminated to prevent it from sending data to outside processes, but since W and DB have different communication patterns, the contamination governing W must differ from that governing DB.

These interactions are not easy to get right without debugging, and to make matters worse, debugging an incorrect label configuration is difficult itself. Plausible errors such as DBP lacking receiving privilege for DB's messages have effects indistinguishable from exploit attempts, so the kernel must hide even these errors' existence from applications.

3.1 Asbestos

To explain these issues more concretely, we now present an overview of the Asbestos operating system. More detail has been published elsewhere [14]. Asbestos is a message passing operating system based on decentralized information flow control. The fundamental IPC primitive is a message sent from one process to another. The kernel tracks the flow of information among processes by manipulating processes' labels.

An Asbestos label is a function mapping opaque identifiers called *tags* to sensitivity *levels*. To the kernel tags are opaque; applications give them semantic meaning. Any process can allocate a tag. This operation returns a previously unused member of the set of 2^{61} possible tags and gives the allocating process privilege over that tag, meaning that the process can freely manipulate information flow for that tag. Privilege is represented by a special level, denoted $\star$. The other levels allow Asbestos to combine secrecy and integrity tracking into a single namespace (more usually, systems track secrecy and integrity using separate labels [8, 10]). These levels, written **0** through **3**, have the following meanings:

0 High integrity;	**2** Low integrity;
1 Default integrity and secrecy;	**3** High secrecy.

A label combines explicit levels for zero or more tags with a *default level* that applies to all tags not otherwise mentioned. For instance, the label $L = \{a\mathbf{0}, b\star, c\mathbf{3}, \mathbf{1}\}$ specifies $L(a) = \mathbf{0}$, $L(b) = \star$, and $L(c) = \mathbf{3}$; for any other tag t, $L(t) = \mathbf{1}$.

Each process P has two labels, a *tracking label* and a *clearance label*. The tracking label $\mathbf{T}_P$ represents the information P has seen. The default tracking label is $\{\mathbf{1}\}$, a label with level **1** for every tag.

When a process allocates a new tag it acquires privilege with respect to that tag. For instance, if P allocated tag t, its tracking label would become $\{t\star, \mathbf{1}\}$. Receiving a message raises a process's tracking label to indicate the flow of information. For instance, if P received a message from a process Q with tracking label $\mathbf{T}_Q = \{u\mathbf{2}, \mathbf{1}\}$, its tracking label would change to $\mathbf{T}_P = \{t\star, u\mathbf{2}, \mathbf{1}\}$. This least-upper-bound operation [5] is implemented as component-wise maximum, except that privilege is preserved (i.e., components at level $\star$ are preserved). The clearance label $\mathbf{C}_P$ limits what information a process can view. A process P cannot receive a message that would increase its label above its clearance for any tag. The default clearance label is $\{\mathbf{2}\}$, allowing a process to receive low-integrity information, but not high-secrecy information. For instance, if processes P and R have $\mathbf{T}_R = \{v\mathbf{3}, \mathbf{1}\}$ and $\mathbf{C}_P = \{\mathbf{2}\}$, then P cannot receive any message from R since $\mathbf{T}_R(v) > \mathbf{C}_P(v)$. A process may freely raise levels in its tracking label and lower levels in its clearance label (as long as the tracking label remains no greater than the clearance label), but lowering tracking levels and raising clearance levels requires the intervention of processes with privilege for the relevant tags. Processes can apply additional labels when sending messages, either to indicate additional contamination or to transfer privilege when appropriate. Asbestos's label changes contain covert channels, but these are not fundamental to all dynamic DIFC systems [8, 16].

The label primitive serves multiple higher-level functions, including *contamination* (where certain entities, such as processes and/or files, have higher secrecy than default with respect to one or more tags) and *privilege* (the ability to remove contamination, effectively *declassifying* information, by lowering its secrecy level). All processes and/or objects with similar information flow characteristics for some tag are informally described as being in a *compartment*; the policy language adopts this term. A process may create arbitrarily many tags, and therefore may belong to or hold privilege for arbitrarily many compartments.

Figure 2 shows an example set of Asbestos labels that implement the policy of Figure 1. For example, DB's $db\mathbf{3}$ tracking label component prevents it from sending information to any process with clearance less than $db\mathbf{3}$; only DBP has the required clearance. If the operations required to set up these labels are poorly designed or implemented, the resulting system would be non-functioning, insecure, or both. For instance, if DBP is not granted privilege to send information to DB ($db'\star$ tracking label component), the system will not function. Similarly, if DB's sending and receiving ability are not restricted appropriately, the system may be insecure.

Expressing policies in Asbestos label terms is a difficult task, especially when performed "from scratch." Both the difficulty of this process and the consequences of incorrectly defining and implementing application policies underline the need for mechanisms and tools that would make policy definition and implementation easier to manage, more human friendly, less error prone, and easier to understand.

Several other Asbestos features impact our designs for improving the DIFC manageability. First, Asbestos messages are sent to Mach-style *ports*, whose names share the tag namespace. Applications can create arbitrarily many ports. Additionally, each port has its own private clearance label, allowing applications to implement capability-style security policies. Second, Asbestos's file system [2] allows applications to store labeled data on disk. Special files called *pickles* are used to map tags, and privilege, to file names; the *unpickle()* operation can recover tag values and privilege from the pickle files. Finally, a process-like abstraction called the *event process* (EP) reduces the memory required to implement isolated server applications. Event processes involve a specialized communication pattern that must be reflected in our policy language. They are used to represent child processes with different labels from the parent, but whose memory state is expected to remain close to the parent's; various kernel resources are optimized for the case of small deltas between parent and child. Each EP starts up inheriting its parent's labels and consequently all its privilege and restrictions. An event process-based server application, such as the Asbestos web server [14], can greatly reduce the memory burden for maintaining memory caches of labeled data; the Asbestos web server uses

1. N can send to any process but L and DB. It can receive from any process but L and DB.
2. D can send to any process but DB. It can receive from any process but L and DB.
3. L can send to no process in the system, and can receive from no process but D.
4. W can send to no process but N, D and DBP, and can receive from any process but L and DB.
5. DBP can send to or receive from any process but L.
6. DB can send to or receive from no process but DBP.

Figure 3: Our example policy expressed as communication restrictions.

about 1.4 memory pages per differently labeled server instance. In the AWS example of Figure 1, W and all worker processes would be implemented using EPs.

4 POLICY MANAGEMENT

DIFC systems take on the responsibility of enforcing security policies, but shift the responsibility of policy definition and implementation to application developers. This allows developers to create more interesting policies, but with current tools the policies themselves are difficult to construct. Practical reasons for this include expressing policies directly in code, which spreads policy implementation across multiple code locations and complicates reading, sharing, and debugging the policy, and the label abstraction, which may be more difficult to engineer than process communication relationships. We therefore develop a prototype language to experiment with expressing labels in terms of pairwise process communication relationships. This requires ways to refer to components participating in pairwise communication and, most importantly, allowed and forbidden communication behavior. Our proposed *policy description language* is capable of expressing application policies in terms of communication constraints. We compile these constraints down into the appropriate labels that implement the constraints.

4.1 Policy Description Language

The hypothesis underlying our policy description language is that developers would prefer to express security policies in terms of communication relationships rather than in labels. We thus designed a language for expressing policies that resembles the relationships diagrammed in Figure 1 (and summarized in Figure 3), but compiles to labels like those in Figure 2. The language specifies these relationships in one compact and intentionally simple definition, rather than scattering necessary operations throughout application code.

The main task of the policy language is to represent the principals involved in the policy, which we call *compartments*, and to specify the *communication rules* that constrain communication between those compartments. Additional constructs facilitate policy instantiation and handling of dynamic policy-related requirements at runtime.

First, a *compartment* represents a set of objects that should be treated uniformly by the security policy. In Asbestos, these objects include application processes, process-like abstractions such as event processes, system services such as the network daemon, and files. Each compartment has a unique name and is defined by the *comp* construct. Figure 4 presents a simplified version of our Asbestos web server policy written in the policy description language; lines 1–17 define the system's 6 compartments.

Given a set of compartments, the system's communication behavior can be defined pairwise: for any two compartments X and Y, the policy defines what communication is allowed between X and

```
1       comp N {
2         default <>
3         env NET_S NET_R
4       }
5       comp DB {
6         default !
7         unpickle /path/db_s /path/db_r
8       }
9       comp D DBP {
10        default <>
11      }
12      comp L {
13        default !
14      }
15      comp W {
16        default <
17      }
18
19      L < D
20      W <> N
21      W <> D
22      W <> DBP
23      DB <> DBP
```

Figure 4: A simplified implementation of our example policy in our policy language. The parser will use this description to produce the labels of Figure 2.

Y. The four possibilities for each pair are no communication, bidirectional communication, and (less frequently) unidirectional communication in either direction. We write these possibilities as X ! Y (X and Y cannot communicate), $X > Y$ (X can send to Y but not receive from Y), $X < Y$ (X can receive from Y but not send to Y), and $X <> Y$ (X and Y can communicate freely). The last rule stated for a given pair of compartments takes precedence. Lines 19–23 define explicit *communication rules* for the AWS application.

To avoid the tedium of writing a full pairwise rule matrix, most relationships are defined implicitly through *default* rules. Each compartment is associated with a default communication pattern, either bidirectional (<>), send-only (>), receive-only (<), or isolated (!). In the absence of an explicit rule definition, the communication between X and Y is defined as the intersection of the corresponding defaults. For example, Figure 4 implies that W ! DB (the intersection of W's default $W < DB$ and DB's default DB ! W). The default communication pattern also constrains a compartment's communication with entities not explicitly mentioned in the policy.

The language is able to express all possible pairwise communication patterns, including patterns that have no sensible mandatory DIFC equivalents. For example, information flow generally obeys a transitive property: if A can send to B, and B can send to C, then A can send directly to C. Asbestos can prevent A from sending directly to C only if B is trusted not to transfer its right to send to C. This leaves the preservation of the communication pattern at B's discretion. Our current prototype does not generate warnings on such cases, but rather spreads privilege as required to implement a policy. (Alternately, B could lose the ability to communicate with one compartment after sending a message to the other; in our policy language, $\ll$ and $\gg$ communication operators support this pattern.)

4.1.1 *Implementation*

The basic task for the implementation of this simple language is to calculate labels for each compartment's processes and other entities that, together, constrain communication as required by the policy. However, since we intend the language to replace existing error-prone label manipulations, the implementation should *produce* these labels at run time, rather than simply providing them for the developer's information. Several language features support this *launcher* functionality, including definitions of application binaries associated with each compartment; the launcher program can

X default	$\mathbf{T}_X$	$\mathbf{C}_X$
$<>$	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$
!	$\{x\mathbf{3}, x'\star, \mathbf{1}\}$	$\{x\mathbf{3}, x'\mathbf{0}, \mathbf{2}\}$
$<$	$\{x\mathbf{3}, \mathbf{1}\}$	$\{x\mathbf{3}, \mathbf{2}\}$
$>$	$\{x'\star, \mathbf{1}\}$	$\{x'\mathbf{0}, \mathbf{2}\}$

Figure 5: The label implementations for X's four possible default communication behaviors. For instance, in the case of "!", $x\mathbf{3}$ in $\mathbf{T}_X$ prevents sending to any process that doesn't have special $x\mathbf{3}$ clearance, while $x'\mathbf{0}$ in $\mathbf{C}_X$ blocks all incoming messages unless the sender has $x'\star$ privilege. Note that $\mathbf{T}_X$ must always be less than or equal $\mathbf{C}_X$, for any tag. (The default tracking and clearance levels for these tags in other processes are $\mathbf{1}$ and $\mathbf{2}$, respectively.)

then start these binaries with the correct labels. Advanced application features require additional support. For example, server applications can create and destroy user compartments at run time, as users join and leave the system. The policy language supports this by allowing the user to dynamically parameterize compartment properties at run time. In this section, we describe the labels generated for each combination of communication rules, then describe more advanced implementation features.

The policy language implementation begins by defining two tags per compartment, one for controlling sending information and one for controlling receiving information. For compartment X, these are written x (the "send tag") and x' (the "receive tag"), respectively. Two tags per compartment are sufficient to implement any pairwise communication policy. Although two tags are not necessarily minimal—some policies would require fewer than two tags for some compartments—tags are not a limited resource and more tags cause little performance penalty in practice [14].

Processes in compartment X have their label components for x and x' defined by X's default communication pattern. Figure 5 presents the Asbestos label implementation of the four compartment defaults for compartment X. Note that the unrestricted default communication rule, $<>$, leaves a process's labels unchanged from the system-wide defaults.

A communication rule involving two compartments affects the labels of processes in either compartment. Figure 6 demonstrates partial labels for the sixteen possible combinations of rules and defaults. Our implementation chooses to implement rules of the form $X\ ?\ Y$ using X's tags. Y's tags are involved only if the requested communication differs from to Y's default; for example, implementing a rule like $X < Y$, which allows Y to send to X, would use the y and y' tags only if Y's default were ! or $<$, which prevent Y from sending by default.

We use the label fragments from Figure 6 to set up compartment labels. First, the compartment tags implement the defaults of all compartments, based on Figure 5's translations. Then, for each rule, we chose the table from Figure 6 that corresponds to the left-hand compartment's default and use it to look up the rule translation. For instance, if X's default is "$<$" ("receive-only") and the rule is $X > Y$, then we will use X's compartment tags as shown on the third line of the "default: $<$" table. If the rule operator violates the other compartment's default, we interpret the rule using its tags as well.

Having identified the translation rules that allow us to map defaults and rules to Asbestos labels, we built a parser capable of translating the policy language to Asbestos label configurations. Using Figure 6's rules, the policy description of Figure 4 is translated to labels identical to Figure 2.

4.1.2 *Launcher*

An application *launcher* incorporated into the language parser instantiates these labels at run time using additional language constructs.

Executables Using an *exec* block the developer may declare application executables to be started in a given compartment, including the path of the executable ("bin"), any arguments to be passed ("args"), and any other compartments to which the processes will belong ("belongs"). If an executable belongs in multiple compartments, its process's labels will be the combination of all compartments' calculated policy labels; if the compartments have conflicting defaults (e.g., one is unrestricted while another is isolated), then each of the defaults will be implemented in the process labels, which effectively enforces the most restrictive default. As it initializes a compartment, the launcher creates the compartment's send and receive tags. To start a process, the launcher effectively forks, sets up the forked process's labels, and executes the named executable, much like a shell.

The language may also describe *external* compartments, which contain processes that run independently of the launched application. An example is the network daemon. The tags used to implement communication rules for external compartments are defined by *env* or *unpickle* language statements, which tell the launcher to examine environment variables or file system files, respectively, to find the tag values. Our prototype launcher does not actually implement the defined policy on external compartments; instead, it assumes the external compartments are already acting correctly.

Event Processes EPs are modeled as forks of a base process, like a "dynamic process" that may have multiple instances. A *dynexec* declaration block nested within an *exec* block defines EP policy. Since an EP is an executable instance, its declaration may specify most of the properties of an executable, such as additional compartments. Each new EP is hosted in a separate, dynamically created compartment. The "source" property identifies the executable that is responsible for spawning new EPs by sending the relevant messages to the base process.

The declarations inside the *dynexec* block are initialized at EP creation time by the process responsible for creating the EP. That process executes a special block of code generated by the policy launcher; that generated code instantiates the policy. The code is parameterized through environment variables at run time, allowing the caller to customize each EP.

In our example, each of the EPs is "spawned" when D dispatches a new user to W, and needs to be in a receive-only, per-user compartment. The automatically generated code ran by D upon creation of a new EP expects to initialize the new EP compartment tags using the environment variables USER_S and USER_R. Before calling the code, D ensures that USER_S and USER_R have been initialized appropriately to reflect the current user.

Using the generated code, the process spawning the EPs is able to automatically implement application policy for each new EP, create and transfer ports to bootstrap communication, and set EP environment variables all in one step.

Bootstrapping The *port* directive allows developers to declare uniquely named ports for a process or EP. Also, since Asbestos ports are labeled and can participate in the policy, we support the declaration of compartments to which a port's label belongs, therefore ensuring that the port label permits the reception of messages that carry the contamination of those compartments. Moreover, developers can further specify the policy port labels implement by declaring whether the port is "restricted" or "open": a restricted port's label requires privilege with respect to the port for a message to go through, while an "open" port's label does not.

The *env* and *env** properties declare environment variables initialized using process or EP port names, exporting port information

X default: $<>$	$\mathbf{T}_X$	$\mathbf{C}_X$	$\mathbf{T}_Y$	$\mathbf{C}_Y$
$X <> Y$	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$
$X \,!\, Y$	$\{x\mathbf{2},\mathbf{1}\}$	$\{x'\mathbf{1},\mathbf{2}\}$	$\{x'\mathbf{2},\mathbf{1}\}$	$\{x\mathbf{1},\mathbf{2}\}$
$X < Y$	$\{x\mathbf{2},\mathbf{1}\}$	$\{\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{x\mathbf{1},\mathbf{2}\}$
$X > Y$	$\{\mathbf{1}\}$	$\{x'\mathbf{1},\mathbf{2}\}$	$\{x'\mathbf{2},\mathbf{1}\}$	$\{\mathbf{2}\}$

X default: $<$	$\mathbf{T}_X$	$\mathbf{C}_X$	$\mathbf{T}_Y$	$\mathbf{C}_Y$
$X <> Y$	$\{\underline{x\mathbf{3}},\mathbf{1}\}$	$\{\underline{x\mathbf{3}},\mathbf{2}\}$	$\{x\star,\mathbf{1}\}$	$\{x\mathbf{3},\mathbf{2}\}$
$X \,!\, Y$	$\{\underline{x\mathbf{3}},\mathbf{1}\}$	$\{\underline{x\mathbf{3}},x'\mathbf{1},\mathbf{2}\}$	$\{x'\mathbf{2},\mathbf{1}\}$	$\{\mathbf{2}\}$
$X < Y$	$\{\underline{x\mathbf{3}},\mathbf{1}\}$	$\{\underline{x\mathbf{3}},\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$
$X > Y$	$\{\underline{x\mathbf{3}},\mathbf{1}\}$	$\{\underline{x\mathbf{3}},x'\mathbf{1},\mathbf{2}\}$	$\{x\star,x'\mathbf{2},\mathbf{1}\}$	$\{x\mathbf{3},\mathbf{2}\}$

X default: $!$	$\mathbf{T}_X$	$\mathbf{C}_X$	$\mathbf{T}_Y$	$\mathbf{C}_Y$
$X <> Y$	$\{\underline{x\mathbf{3}},\underline{x'\star},\mathbf{1}\}$	$\{\underline{x\mathbf{3}},\underline{x'\mathbf{0}},\mathbf{2}\}$	$\{x\star,x'\star,\mathbf{1}\}$	$\{x\mathbf{3},\mathbf{2}\}$
$X \,!\, Y$	$\{\underline{x\mathbf{3}},\underline{x'\star},\mathbf{1}\}$	$\{\underline{x\mathbf{3}},\underline{x'\mathbf{0}},\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$
$X < Y$	$\{\underline{x\mathbf{3}},\underline{x'\star},\mathbf{1}\}$	$\{\underline{x\mathbf{3}},\underline{x'\mathbf{0}},\mathbf{2}\}$	$\{x'\star,\mathbf{1}\}$	$\{\mathbf{2}\}$
$X > Y$	$\{\underline{x\mathbf{3}},\underline{x'\star},\mathbf{1}\}$	$\{\underline{x\mathbf{3}},\underline{x'\mathbf{0}},\mathbf{2}\}$	$\{x\star,\mathbf{1}\}$	$\{x\mathbf{3},\mathbf{2}\}$

X default: $>$	$\mathbf{T}_X$	$\mathbf{C}_X$	$\mathbf{T}_Y$	$\mathbf{C}_Y$
$X <> Y$	$\{\underline{x'\star},\mathbf{1}\}$	$\{\underline{x'\mathbf{0}},\mathbf{2}\}$	$\{x'\star,\mathbf{1}\}$	$\{\mathbf{2}\}$
$X \,!\, Y$	$\{x\mathbf{2},\underline{x'\star},\mathbf{1}\}$	$\{\underline{x'\mathbf{0}},\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{x\mathbf{1},\mathbf{2}\}$
$X < Y$	$\{x\mathbf{2},\underline{x'\star},\mathbf{1}\}$	$\{\underline{x'\mathbf{0}},\mathbf{2}\}$	$\{x'\star,\mathbf{1}\}$	$\{x\mathbf{1},\mathbf{2}\}$
$X > Y$	$\{\underline{x'\star},\mathbf{1}\}$	$\{\underline{x'\mathbf{0}},\mathbf{2}\}$	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$

Figure 6: Mapping of $X\ ?\ Y$ communication rules to labels. The label components corresponding to X's default (Figure 5) are underlined; X and Y's labels are further modified with respect to x and x' according to the non-underlined elements. We assume Y's default is $<>$; other defaults are analogous.

out from the launcher. The *env** property also grants the launching process privilege with respect to the named ports.

Running the Application When instantiating the application using the configuration file, the launcher forks a process for each executable, then performs the following tasks:

- Implementation of the policy by setting up process labels as calculated for each process.
- Creation of all process ports and setup of port labels according to policy.
- Environment variable initialization for all processes using the newly created port values.
- Granting of port privilege to processes according to policy.
- Transfer of ports to their respective owners.
- Making the new processes runnable.

Once the developer has expressed the policy using the policy configuration language, little or no code modification is required for the application to run using the launcher.

A complete implementation of the AWS policy, including executable declarations, environment variables and EPs, can be found online [1].

4.2 Discussion

Our language attempts to simplify policy description, but does not seek to replace Asbestos labels. If we find that labels are subsumed by our policy language, then there might be no need for labels. Currently, though, our language cannot capture the full expressiveness of labels. For instance, a process may participate in multiple different policies, each of which is depicted on its labels. The combination of all policies defines the process's final behavior. Our policy description language is able to define each policy separately, but can not replace the globally enforced process labels in IFC tracking. Although we have achieved label operation performance capable of supporting realistic applications, such as the AWS, it will require significant effort before the policy language is optimized enough to reach similar performance levels. Nevertheless, our policy language can already use communication relationships to represent fairly complex policies.

5 DEBUGGING MECHANISMS

Although the policy description language helps implement correct policies, it is hard to eliminate all development errors causing policy-related bugs, such as improper declassification, lack of necessary privilege or clearance, or processes getting contaminated improperly. These bugs usually manifest themselves as label errors and—from the kernel's point of view—as attempts to violate information flow rules indistinguishable from genuine attempts to break system policy. Even in the absence of label errors, while developing for Asbestos we had to resolve other, more traditional types of bugs that caused unexpected process death, such as system call failures (e.g., due to bad arguments) or null pointer dereferences. Debugging mechanisms in a conventional development environment assume free access to system and application state. Gathering and exposing information such as stack traces, system call traces, file operations, and communication behavior can be done with few or no restrictions. In the presence of DIFC, debugging is rendered challenging because this ability to gather system information is restricted by policy rules.

Exposing debugging information to developers almost always causes debugging data to flow between compartments. As with any other data exchange between compartments, debugging messages generated by the system must not violate policies enforced by Asbestos DIFC. Our goal was therefore to develop useful debugging facilities whose information flow behavior cleanly maps onto Asbestos's existing DIFC model, and especially its privilege model. We previously performed most of our debugging by exercising our access to the Asbestos console, where we could inspect all related messages printed by the kernel. Of course, unprivileged developers cannot be allowed to utilize this global channel.

The local channels of decentralized privilege did guide our design, however. Without console access, an Asbestos developer might gain debugging visibility by spreading privilege more widely than usual. For example, the application's components might get privilege for the tags corresponding to a special "debug user." This widespread privilege would relax communication restrictions for the corresponding tags, giving debuggers better visibility into application behavior. Of course, privilege would also change application behavior; to be useful for debugging, the application itself would emulate the unprivileged behavior, reporting any errors it observed.

Although this hypothetical system would present implementation difficulties, such as correctly emulating unprivileged behavior at user level, it would clearly fit into Asbestos's DIFC model. Inspired by this analogy, we introduce the *debug domain* primitive and present a system that utilizes kernel extensions to provide similar behavior as the hypothetical system, but with much better ease of use. The result cleanly fits debugging into Asbestos DIFC.

5.1 Label Errors

The high frequency and importance of label errors for Asbestos development make them a good working example for DIFC debugging. Assume that process P attempts to send a message to process Q, but the attempt fails because of a label error caused by poor policy implementation. (For instance, a level in $\mathbf{T}_P$ might be above the corresponding level in $\mathbf{C}_Q$.) Such an error will be treated as any other label error—that is, as an attempt to violate information flow rules—and information about the error, including its very existence, is concealed by the Asbestos kernel. This makes it particularly difficult for an unprivileged application developer to diagnose and fix the bug. It would be very helpful if the system provided information such as:

- The source and destination of the message that caused the label error;
- Identifying message details (e.g., its type and ID);
- The tag/port that caused the label error (the "faulting tag");
- The levels of the faulting tag in the sender's tracking label and the receiver's clearance label; and
- The particular type of label error.

A message containing this information conveys to its recipient information from both the sender's and receiver's compartments, such as the type of label error and the faulting tag level on both sides' labels. Therefore, in order to preserve information flow, the message should also carry both processes' contaminations. This will ensure that the debug message may only be received by processes that have clearance to receive information from both P and Q.

In IFC terms, collecting debugging information belonging to various compartments requires privilege to declassify information with respect to those compartments. Since security policies rarely provide application developers with the required privilege, we need to provide a way for developers to exercise privilege in a limited, controlled way, only for debugging purposes. Essentially, we want to create a new type of *debugging* privilege, which will represent the ability to declassify debugging information with respect to a set of tags. The developer will exercise this privilege by granting the application debugging privilege over a set of application tags, such as the tags corresponding to a fake user intended only for debugging. When an error occurs, the system will search for error information subject to debugging privilege and report any such information to the relevant debugger process or processes. However, to maintain proper information flow control, the kernel appropriately labels the debugging information; in the case of the label error, the resulting label is $\mathbf{T}_P \sqcup \mathbf{T}_Q$, which combines both P and Q's tracking labels. A debugging process will only see the information if allowed, but since P and Q's labels will generally consist of tags subject to debugging privilege, issues with hidden errors will likely not arise.

5.2 Asbestos Debug Domains

The *debug domain* (DD) primitive represents this debugging privilege and attempts to address DIFC's debugging challenges. A DD models three types of privilege: the privilege to declassify debugging information with respect to certain compartments, the privilege to receive such information, and the privilege to manage this mechanism. In essence a DD specifies to the kernel what tags should generate debugging messages and which processes should receive those messages. DDs may address a number of different debugging problems sharing three basic characteristics:

- They are triggered by a specific type of event, such as a label error;
- They involve a specific set of triggering tags, such as a set of potential faulting tags; and
- all parties that hold the appropriate debugging privileges and have declared interest in such events should receive debugging messages when such events occur.

The primary element of a DD is a collection of tags called DD *member tags*, or simply *members*. The DD represents privilege to debug with respect to its members, essentially representing privilege to declassify debug information—with respect to members—to anyone able to receive debug messages from that particular DD.

Holding privilege over the DD gives a process the right to manipulate its member list and the right to *connect* listening ports to the DD. By connecting a listening port to a DD, a process instructs the DD to send debug messages to that port when bugs involving one (or more) of the DD members occur. Each DD can have an arbitrary number of listening ports, as well as an arbitrary number of member tags. Apart from privilege over the DD, adding member tags and/or connecting listening ports also requires privilege over the tags/ports that are being added/connected. By requiring privilege over all relevant tags in order to manipulate a DD we bring debugging privilege into the label system as an instance of an already-existing privilege.

Privilege is also required to modify the properties of a DD, such as the types of events that are being monitored and the parties that will be notified when such events are triggered (the owner of the DD, the sender of the message that triggered the event, the recipient of the message, or any combination of the three). However, processes that receive messages sent to listening ports need not have privilege for the DDs to which those ports are connected.

Figure 7 is a visual representation of a DD represented by tag *ddt*.

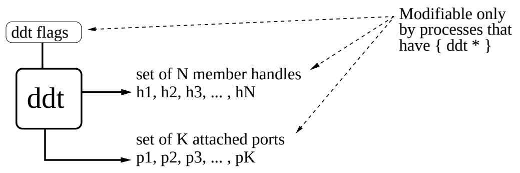

Figure 7: A visual representation of a debug domain represented by tag *ddt*, including its member tags, connected listening ports, and flags (characteristics).

Any process may create an arbitrary number of DDs, whose members may or may not be disjoint; any tag may be a member of multiple DDs. A process's ports may be connected to multiple DDs.

5.3 Implementation

A debug domain tag *ddt* represents a single DD and is used to manage its associated member tags and listening ports. *ddt* is created by calling the sys_new_tag() system call with the appropriate arguments, and DD properties can be managed through the use of sys_debug_ctl().

The DD primitive is used in conjunction with different types of triggering events to implement four debugging applications:

- Label error debugging, using label errors as triggering events.
- System call debugging: using issuing of system calls as triggering events, we generate debug messages containing information about the system call, its caller and its results.
- Label history debugging: using label changes as triggering events, we generate debug messages that contain the label delta.
- Process exiting: using a process's death as triggering event, we generate a debug message that informs processes that may be interested in this event (e.g., processes that have called wait() on that process).

Each triggering event is handled by a wrapper function responsible for that type of error. Eventually, these wrapper functions generate and send debug messages to all appropriate destination ports. Sending a debug message involves the following steps:

Message type	Message label
Label-error debugging	$(\mathbf{T}_P \sqcup \mathbf{T}_Q) \sqcap \{d_1\star, d_2\star, \ldots, d_n\star, p_l\star, \mathbf{3}\}$
Syscall tracing	$\mathbf{T}_P \sqcap \{d_1\star, d_2\star, \ldots, d_n\star, p_l\star, \mathbf{3}\}$
Label tracing	$\mathbf{T}_P \sqcap \{d_1\star, d_2\star, \ldots, d_n\star, p_l\star, \mathbf{3}\}$
Process exiting	$\mathbf{T}_P \sqcap \{d_1\star, d_2\star, \ldots, d_n\star, p_l\star, \mathbf{3}\}$

Figure 8: Labels for messages generated by the four debug domain applications. $\sqcup$ is the least upper bound operator and $\sqcap$ the greatest lower bound operator. $d_1, d_2, \ldots, d_n$ are the member tags of the DD, while p_l is the connected port to which the message will be sent. For label-error debugging process P has attempted to send a message to process Q; for the other applications, P is the relevant process.

1. Check that the relevant process has declared interest in this type of event. If not, return.
2. Iterate over the debug domains of which the faulting tag (i.e., the tag/port that triggered the event) is a a member. Discard the domains that are not related to the event type in question.
3. For each remaining debug domain, investigate its listening ports. If the port belongs to one of the processes that are supposed to be notified, generate and send a debug message to that port. For instance, if the DD flags specify that the sender of the relevant message must be notified, the debug message will be sent to any listening ports belonging to the sender. Other possible recipients include the destination of the message and the owner of the debug domain.

The most sensitive aspect of writing a debug wrapper function is identifying the proper label that needs to be attached to the message.

Our implementation of DDs and their applications involved numerous changes in the Asbestos kernel, including debug domain implementation, identifying cases requiring debug message generation, actually generating the messages, and managing debug domains via system calls. At the user level, we implemented interfaces that allow developers to create and manage DDs as well as higher-level library calls that use DDs to provide a service, such as system call tracing and debugging libraries.

5.4 Applications

Label Error Debugging Label error debugging uses DDs to provide useful information about label errors. Each such debug message contains information about the type of the label error, the faulting tag, the source and destination of the faulting message, and the level of the tag in the sender's tracking label and the receiver's clearance label. (In the presence of more than one faulting tag we would fault on each of them separately, generating multiple debug messages.) Since this message is revealing information about the destination (e.g., the destination clearance label with respect to the faulting tag) we need to make sure that it is properly contaminated: the recipients of such debug messages will carry the contamination of both the sender and receiver of the faulting message. For instance, if P tries to send a message to Q and fails because of a label error, label debugging needs to examine the faulting tag, determine which processes may receive a debug message with respect to that tag, and label the message with the least upper bound of P and Q's labels, explicitly lowered for debug domain members (since a privileged process has explicitly permitted debugging—i.e., declassification—with respect to those tags). The exact contamination carried by the debug message in this example is presented in Figure 8.

To make the use of DDs more concrete, we revisit our AWS example. The worker code example of Figure 9 uses a private file to store some of its state. Function *worker_init*() creates a new tag *mytag* (line 7), uses it to make the file private by contaminating it with $\{mytag\,\mathbf{3}\}$ (line 8), and grants itself clearance to receive such

```
 1  void
 2  worker_init(char ** argv, int argc) {
 3    tag_t mytag, ddt, dport, debugger;
 4
 5    /* create new tag and use it to contaminate
 6       private file */
 7    sys_new_tag(&mytag, "my secret port");
 8    writefile(priv_file, Contamination={mytag 3, 1});
 9
10    /* allow ourselves to access
11       contamination {mytag 3} (i.e. read file) */
12    self_give_clearance(mytag, 3);
13
14    /* create new DD (ddt) and connect dport to it at
15       creation time. Add mytag to its members */
16    sys_new_dd(&ddt, DEBUG_LABELS, &dport);
17    sys_add_member_to_dd(ddt, mytag);
18
19    /* spawn debugger and transfer dport to it, so it
20       can receive debug message from it */
21    spawn_process(&debugger);
22    sys_transfer_tag(dport, debugger);
23    ...
24
25    /* prematurely drop {mytag *} privilege */
26    sys_tag_drop_privilege(mytag);
27    r = http_output("Initialization: success!");
28    ...
29    return;
30  }
31
32  int
33  main(char ** argv, int argc) {
34    worker_init(argv, argc);
35    ...
36    /* by reading file, we get contaminated with
37       {mytag 3} since we dropped privilege */
38    read_from_my_file();
39
40    /* {mytag 3} contamination may not
41       escape to the network. LABEL ERROR! */
42    http_output(input);
43    ...
44  }
```

Figure 9: Code example where DDs (used by the debugger) would help diagnose a bug causing a label error (because of dropping privilege prematurely on line 26).

contamination later (line 12). Then the function creates a label-error DD, represented by tag *ddt*, connects port *dport* to it (at creation time), and adds *mytag* to its members (lines 16–17). This allows the owner of *dport* to receive debugging messages whenever a label error in relation to *mytag* occurs. A new debugger process is spawned and ownership of *dport* is transfered to it (lines 21–22). On line 26 the process mistakenly drops privilege with respect to *mytag* and therefore becomes susceptible to $\{mytag\,\mathbf{3}\}$ contamination, which it receives when the main function tries to access the private file on line 38. This leads to a label error when the process tries to output to the network on line 42, since $\{mytag\,\mathbf{3}\}$ is private to the worker and the network daemon does not have clearance for $\{mytag\,\mathbf{3}\}$.

In this example, the debugger has the $mytag\,\star$ privilege required to declassify messages with respect to *mytag*, and therefore can notify the developer about the label error debug message received because of the forbidden operation on line 42.

System Call Tracing We used debug domains to provide users with system call tracing messages. By adding the special control tag of process Q to a DD that is configured to use system calls as triggering events, we enable system call tracing for Q. Debug messages containing information about every system call Q issues (system call type, arguments passed to it and return value) will be sent to all ports connected to the DD.

Since these messages directly expose information related to the process that is being traced (Q), each debug message must carry Q's contamination, omitting any contamination related to debug domain members.

Label History and Exiting Processes Debugging is facilitated by a mechanism that informs developers of changes to a process's labels. If process *P*'s control tag is added to a DD using label modifications as triggering events, "label history" debug messages will be generated every time *P*'s labels change. Each message contains the differences in the label components as well as the type of action that led to the change (e.g. "reception of message"), and carries *P*'s contamination. In the context of the example presented in Figure 9 the developer can use label history to identify the calls that led to dropping *mytag* privilege (line 26) and getting contaminated with respect to *mytag* (line 38).

Additionally, by using the death of a process as the triggering event, a final type of DD identifies when processes exit, potentially due to bugs or failures.

6 EXPERIENCES AND EVALUATION

To evaluate our system management mechanisms, we show how they could aid the development of applications previously presented by DIFC systems such as Asbestos, HiStar, and Jif. More specifically, we use our policy management tools to express a variety of DIFC application policies, and then demonstrate that debug domains are able to deliver policy-safe debug messages with reasonable overhead.

6.1 Implementation of Sample Policies

Our policy language's parser and launcher are implemented in Python. The actual runtime cost of parsing and launching policy configurations is minimal, even in the case of long, complex policies, but is currently hampered by large Python startup costs on Asbestos.

We have used our policy language to describe several interesting policies. Although no static policy language could describe every dynamic information flow policy, our policy language's ability to express a range of previously presented policies, including challenging ones, indicates its fitness for practical and user-friendly information flow policy specification.

Our simplified version of the AWS policy presented in Figure 1 is based on our implementation of the full Asbestos web server policy [14] in our policy language. This was a challenging exercise since the policy uses all of Asbestos's features, including event processes, to provide both system-based information flow isolation and high performance.

In this example we make use of the *dynexec* directive to describe worker EPs. We have fully implemented the AWS policy, including all executables and ports required for the application. Therefore, the 511-line long specialized AWS launcher previously used for label initialization and process spawning is no longer necessary. Also, a preliminary port of the AWS, currently under development, will allow us to remove at least 28 additional policy related operations from various places in the code. Figure 10 presents the part of the AWS policy that declares the web server worker processes which utilize EPs to handle user connections. The full policy is also available [1].

Using its Unix compatibility layer, HiStar [16] can run the ClamAV anti-virus program, ensuring no leakage of private data even if the ClamAV processes become compromised. The main ClamAV process may receive information from the rest of the system (for instance, it may read a virus database), but it is prevented from exporting information so as to avoid leaks (lines 3 and 15–20). It also has clearance to receive information contaminated with

```
1   comp W { default < }
2   # worker processes for login and
3   # profile viewing AWS services
4   exec worker1 worker2 {
5       bin /okws-login login
6       bin /okws-view view
7       belongs W
8       # each worker needs to communicate
9       # with the rest of the system
10      port WP { type open }
11      # port that will be used as
12      # worker verify handle
13      port WV { type open }
14      # EP declared for each worker.
15      # Each new EP is created when the demux
16      # sends a new user to a worker
17      # (hence the "source" directive)
18      dynexec USER {
19          source demux
20          # EP implements different default
21          # that parent process
22          default !
23          # and belongs to (externally initialized)
24          # user compartment
25          belongs (env USER_S USER_R default <)
26          # file path where generated EP
27          # initialization code will go
28          file /ep-configs/user-tmpl
29          port WRPORT { type open }
30          # user's "grant" tag. {UG *} is required
31          # to modify user's data
32          port UG {
33              type restricted
34              owner parent
35              belongs (env USER_S USER_R default <)
36          }
37          # NETROOT belongs to the net daemon
38          env* NETPORT=port:NETROOT
39          env  WRPORT=port:WRPORT
40          env* DPORT=port:DEMUX_USERP # port to D
41          env* UG=port:UG
42      }
43      env  SELFPORT=port:WP
44      env* DBP=port:DBP # port to database proxy
45      env  MYVERIFY=port:WV
46      # port to the demux
47      env  DEMUXPORT=port:DEMUX_USERP
48  }
```

Figure 10: Part of the AWS policy description showing the declaration of the compartment and executable for AWS workers. The executable also includes worker EP declaration.

respect to the calling user (line 40), but doesn't hold the user privilege required to modify user data. The same applies to the helper processes it spawns. It also utilizes a private */tmp* directory that contains sensitive user data related to ClamAV, and therefore carries both user and ClamAV contaminations (lines 25–30). Finally, a privileged process can declassify information out of the ClamAV compartment and send it to a terminal (lines 4, 7–12 and 38–39). HiStar uses a specialized launcher to run the ClamAV anti-virus program, the 110-line long *wrap* process. This process sets up the application's policy. The ClamAV anti-virus policy example expressed in our policy configuration language is described in 40 lines (Figure 11). Just as *wrap* can protect processes other than ClamAV, so simple changes to Figure 11 can protect different executables; in a sense, Figure 11 is a generalization of *wrap*'s policy.

HiStar also presents a VPN isolation example. This example assumes that a user is simultaneously connected to both the Internet and a virtual private network (VPN), using two separate network stacks and two browser instances (one for each network). A VPN client is placed between the two networks, holding privilege to forward information from the one to the other only when the user explicitly allows it—for instance, after a file from the Internet has been checked for viruses or after a file from the private net-

```
# We first declare the three compartments
comp USER { env USER_S USER_R default ! }
comp AV { default < }
comp PRINTER { default <> }

# Executable declassifying output to the tty
exec tty_printer {
        belongs PRINTER
        port PRINTER_PORT { type restricted }
        env MYPORT=port:PRINTER_PORT
        env AV_PORT=port:CLAMAV_PORT
}
# ClamAV process. Also spawns helper process
# belonging in same compartments
exec avscanner {
        belongs AV
        port MAIN_AV_PORT { type restricted }
        env MYPORT=port:MAIN_AV_PORT
        env PRINTER_PORT=port:PRINTER_PORT
}
# Process modeling private /tmp folder.
# Could be replaced by labeled FS
# This process also belongs to the
# externally initialized user compartment
exec private_tmp_file_server {
        belongs AV USER
        port TMP_PORT { type restricted }
        env MYPORT=port:TMP_PORT
        env AV_PORT=port:MAIN_AV_PORT
}
# Process modeling private user data.
# Could be replaced by labeled FS
exec user_data_server {
        belongs USER
        env AV_PORT=port:MAIN_AV_PORT
}

AV <> PRINTER
USER <> PRINTER
USER > AV
```

Figure 11: HiStar's ClamAV security policy implemented using our policy language.

```
# We declare the five compartments
comp VPN INTERNET INTERNET_IPSTACK { default < }
comp VPN_CLIENT { default <> }
comp NETD { default ! }

# Both the vpn browser and the vpn IP stack
# belong to VPN
exec browser_vpn ipstack_vpn { belongs VPN }
exec browser_internet { belongs INTERNET }
exec ipstack_internet { belongs INTERNET_IPSTACK }
exec vpn_client { belongs VPN_CLIENT }
exec netd {  belongs NETD }

VPN_CLIENT <> VPN
VPN_CLIENT <> INTERNET_IPSTACK
INTERNET_IPSTACK <> INTERNET
INTERNET_IPSTACK <> NETD
```

Figure 12: A policy similar to HiStar's VPN isolation, implemented using our policy language.

work is verified to be non-confidential. Figure 12 shows a similar, but more restrictive policy configuration (we completely disallow browser and IP stack instances from interacting with the rest of the system). Information can escape from one network to the other only if the VPN client declassifies it.

Another interesting policy we were able to express was Jif's [12] medical study example. In this scenario a hospital (modeled on line 3 of Figure 13 by an external compartment) possesses patients' personal data. This data should be anonymized using a data extractor process E and then forwarded to a group of researchers R. To process patient information the researchers use a statistical package SP that accesses a database DB with statistical methods. The researchers should also be able to export the results to an output OUT, modeled as an outside compartment. In this policy we primarily want to ensure that no patient data leak to the outside world (even if application modules have bugs), and secondarily ensure that only the statistical package has access to the confidential statistical database. Unlike the Jif solution, Figure 13 does not provide isolation at the granularity of application variables, but it still achieves the goal of protecting against leakage of patient data.

```
 1   # The external compartment the
 2   # "hospital" process belongs to
 3   comp H { env HOSPITAL_S HOSPITAL_R default <> }
 4   # Compartments for data extractor, researchers,
 5   # statistics package, and DB
 6   comp E R SP DB { default ! }
 7   # Compartment for the process that
 8   # outputs results of study
 9   comp OUT { env OUT_S OUT_R default <> }
10
11   E <> H
12   E > R
13   R <> SP
14   SP <> DB
15   R > OUT
```

Figure 13: A policy configuration implementing the security model of the Jif medical example. For brevity we omit executable declarations.

	Number of AWS users				
	1000	5000	20000	50000	100000
Unmodified AWS	1652	1574	1533	1498	1479
Modified AWS	1651	1567	1493	1490	1454

Table 1: Throughput comparison comparison between the unmodified version of the AWS and the version using debug domains. Each column corresponds to the number of users in the system, ranging from 1000 to 100000. Measurements correspond to connections per second.

6.2 Debug Domains in Use

We have successfully used debug domains to implement debugging tools and proof-of-concept debugging tests. Label error debugging has been verified using instrumented test cases—including situations where the user has no console access—and debug messages were successfully collected. To evaluate the performance hit of debug domains we modified the Asbestos web server so that every tag it generates is added to a label-error DD. Asbestos is running on a 2.8GHz Pentium 4 with 1GB of RAM, connected on a 1Gbps switch. We ran throughput measurements and compared our results to the unmodified AWS. As shown in Table 1, the performance hit for the throughput of AWS is insignificant (overall less than 3% and in most cases less that 1%), even for large numbers of users in the system. (The general throughput improvement relative to previously reported data is due to an improved label implementation [14].)

We ran micro-benchmarks to measure the average cost of some major kernel operations for debugging. The results, presented in Table 2, and show that the cost is reasonable for frequent operations, such as adding a member or connecting a port to a DD, as well as for debug message generation. For label errors, debug message generation requires the kernel to repeat the label checks that lead to the error to capture the necessary details; for all debugging messages, including label errors, the kernel must form the debug message and perform operations to calculate its label.

System call tracing was used to implement an asynchronous $strace()$ library call. Similarly, label history debugging was used to implement a label tracing library call ($lt()$) that reports all of a process's label changes. Additionally, exiting process debugging has been used to implement the Asbestos $wait()$ library call. Finally,

	Add member/ connect port	Debug message generation label (to sender/rcvr)	syscall	exit
cycles	2291 / 1275	31625 / 26316	12548	5606

Table 2: Cost in cycles of debug domain operations: adding of a member tag, connecting a new port, and generating debug messages for label errors (addressed to the sender and receiver of the offending message), system call tracing, and exiting processes.

we have implemented a simple debugger library that is using the debug domain mechanisms to gather debugging information on behalf of one or more processes. Each process may fork a new debugger by calling *debugger_spawn()*. All privilege the process possesses at that time is inherited by the debugger, so that debug information can be declassified even if the process loses privilege at a later time. Library calls have been implemented to grant additional privilege to an already existing debugger if needed.

As a proof of concept application, we have also built a simple tool around AWS that allows users to upload untrusted web server extensions and runs them within a DD. The tool then captured all debug messages the developer had clearance to receive. Through a web-based interface, the tool was able to provide two basic functions: debugging console-like output (e.g. label error reports) and system call tracing.

7 FUTURE WORK

Our policy description language is a first step towards better policy management in DIFC systems. We intend to improve the language interface and give developers better control over policy description. More specifically, developers are expected to produce sensible policy descriptions and our parser is currently unable to identify the configurations that are impossible to implement using IFC. We would like to formalize the characteristics of policy descriptions that cannot be mapped to valid (and secure) label implementations so as to identify such cases and handle them accordingly (e.g. produce helpful, diagnostic error messages).

Although we have already used our policy description language to express a number of policies, we want to test it further by building large applications with challenging policies. More specifically, it would be interesting to build such an application from scratch, without having to write any policy related code "by hand".

We would also like to investigate the reverse problem of translating Asbestos label setups (e.g. snapshots of application labels at runtime) to equivalent high-level policy descriptions. That could facilitate debugging of policy problems that appear only at runtime (e.g. due to interaction with the rest of the system). This problem is very challenging, since it is not always easy to infer the policy from a given label setup.

Finally, we want to improve the usability of our debugging mechanisms, primarily by improving the existing debugger and by implementing more tools and libraries that use debug domains.

8 CONCLUSION

At the heart of any DIFC application lies the application policy, which specifies the rules that govern information flow. The restrictions imposed by the policy affect system management tasks and introduce new challenges for developers.

In this paper we have investigated and proposed solutions for two such system management challenges in Asbestos: policy management and debugging. Our policy description language is able to express a wide variety of policies in a human-friendly way. Tools translate high-level policy descriptions to equivalent Asbestos label configurations and optionally instantiate the policies using application binaries. Furthermore, we identified the requirements for information flow aware debugging mechanisms that can assist developers without violating application policy. We introduced the *debug domain* primitive and used it to implement debugging mechanisms and tools such as label error debugging, system call tracing and label history tracking.

We tested our system management mechanisms using synthetic tests as well as examples of interesting policies from Asbestos and HiStar and observed significant improvement in the ease of policy description, development and elimination of bugs. Although this work is only a first step, hopefully these and other programmability improvements will bring the security benefits of DIFC to a wider community of developers.

ACKNOWLEDGMENTS

The authors would like to thank the members of the Asbestos group, especially Steve VanDeBogart, Max Krohn, Alex Yip, and Micah Brodsky for comments on and usage of prior versions of the system, and the anonymous reviewers and our shepherd Petros Maniatis for their comments and suggestions.

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. Eddie Kohler is supported by a Sloan fellowship and a Microsoft Research New Faculty Fellowship.

REFERENCES

[1] Sample policy implementations. `http://read.cs.ucla.edu/~pefstath/policies/`.

[2] M. Brodsky, P. Efstathopoulos, F. Kaashoek, E. Kohler, M. Krohn, D. Mazieres, R. Morris, S. VanDeBogart, and A. Yip. Toward secure services from untrusted developers. Technical Report TR-2007-041, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2007. `http://hdl.handle.net/1721.1/38453`.

[3] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP '07)*, Oct. 2007.

[4] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the USENIX Security Symposium 2007*, 2007.

[5] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), May 1976.

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

[7] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of 1984 IEEE Symposium on Security and Privacy*, Apr. 1984.

[8] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP '07)*, Oct. 2007.

[9] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of 2001 USENIX Annual Technical Conference—FREENIX Track*, June 2001.

[10] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8), Aug. 1992.

[11] A. C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999.

[12] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4), Oct. 2000.

[13] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the 8th USENIX Security Symposium*, Aug. 1999.

[14] S. VanDeBogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Transactions on Computer Systems*, 25(4):11:1–11:43, Nov. 2007.

[15] R. Watson, W. Morrison, C. Vance, and B. Feldman. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proceedings of 2003 USENIX Annual Technical Conference*, June 2003.

[16] N. B. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Nov. 2006.