University of California

Los Angeles

Policy Management and Decentralized Debugging in the Asbestos Operating System

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Petros Efstathopoulos

2008

Ὁ βίος βραχύς, ἡ δὲ τέχνη μακρή,
ὁ δὲ καιρὸς ὀξὺς, ἡ δὲ πεῖρα σφαλερὴ,
ἡ δὲ κρίσις χαλεπή.

–Ιπποκράτης, *Αφορισμοί*, 400 π.Χ.

Στους γονείς μου,
Βαγγέλη και Σοφία.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost I would like to thank my advisor Eddie Kohler for a number of very important reasons. I consider myself very lucky for having the privilege to work with him and be exposed to his insight and technical expertise. Being who he is, he made the journey towards my graduation a lot more meaningful and fun. But most importantly I want to thank him for being so supporting and understanding in times of trouble.

Furthermore, I would like to thank all the people that contributed to the Asbestos operating system. My thanks go to Steve VanDeBogart, Maxwell Krohn, Cliff Frey, David Ziegler, David Mazières, Frans Kaashoek and Robert Morris.

Additionally, I would like to thank the members of my doctoral committee, professors Vasilios Manousiouthakis, Todd Milstein and Jens Palsberg.

I have no words to thank Verra Morgan, the Student Affairs Officer at the UCLA Computer Science Department. Not only for her practical help with academic issues—especially during my first years at UCLA—but, most importantly, for her invaluable assistance, words of advice and moral support when life away from home became difficult to bear.

I would also like to thank the members of the TERTL lab for all the fun conversations, the technical discussions, the group meetings and every other moment that we shared together.

Last but not least, I would like to express my gratitude to the people that are very close to me. I owe everything I am to my wonderful parents, Vangelis and Sophia whose love brought me this far. I also want to thank my sister Eleni for making me feel safe knowing that there will always be someone I can count on. I consider my dear friends Hector, Aris, Thanos, Katerina and Dionissis familly and

I can not thank them enough for their support. I also want to thank my friends Eleni and Voula for their help and support during the very difficult last couple of months as a graduate student. Finally, I want to express my deepest gratitude and love to Marialena Athanasopoulou, because she was there to lovingly share with me the good and bad moments of most of my journey, with honesty and patience.

Πάνω από όλα όμως θέλω να ευχαριστήσω τον Θεό που επέτρεψε να φτάσω ως εδώ.

Vita

1978	Born, Athens, Greece
1996	Graduated Varvakio Protypo Lykio, Athens, Greece
2001	Diploma in Electrical and Computer Engineering, National Technical University of Athens, Greece
2004	M.S. in Computer Science, University of California, Los Angeles, CA, USA.
2008	Ph.D. in Computer Science, University of California, Los Angeles, CA, USA.

Publications

Petros Efstathopoulos and Eddie Kohler. Manageable Fine–Grained Information Flow. In *Proceedings of the ACM SIGOPS European Conference in Computer Systems (Eurosys)*, Glasgow, UK, April 2008.

Steve VanDeBogart, Petros Efstathopoulos, Maxwell Krohn, Cliff Frey, David Ziegler, Eddie Kohler, David Mazires, Frans Kaashoek and Robert Morris. Labels and Event Processes in the Asbestos Operating System. *ACM Transactions on Computer Systems 25(4)*, November 2007, pages 11:1-11:43.

Micah Brodsky, Petros Efstathopoulos, Frans Kaashoek, Eddie Kohler, Maxwell Krohn, David Mazieres, Robert Morris, Steve VanDeBogart and Alexander Yip. Toward Secure Services from Untrusted Developers. In *CSAIL Technical Reports*, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory.

Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazires, Frans Kaashoek and Robert Morris. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazires, Robert Morris, Michelle Osborne, Steve VanDeBogart and David Ziegler. Make Least Privilege a Right (Not a Privilege) In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, June 2005.

Abstract of the Dissertation

Policy Management and Decentralized Debugging in the Asbestos Operating System

by

Petros Efstathopoulos

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2008

Professor Eddie Kohler, Chair

The continuing frequency and seriousness of security incidents underlines the importance of application security. We have developed *Asbestos*, a novel operating system focusing on security that uses *Asbestos labels* to implement *decentralized information flow control* (DIFC). Using DIFC Asbestos is able to track information flow and contain the effects of malicious or poorly implemented applications. This way, Asbestos applications can be made significantly more secure than applications built with conventional operating systems abstractions, since application security is preserved even in cases where large parts of the applications are compromised. However, our development experience in Asbestos applications showed that achieving Asbestos's benefits was simply too difficult. We believe that an important reason for this problem is Asbestos's challenging programming model.

Based on our development experience for Asbestos, we attempt to improve its programming model. We identify and investigate two important security policy management problems that are critical for Asbestos development: *security policy specification* and *debugging*.

First we present a *policy description language* that can be used to facilitate

application policy management. Using our policy language developers are able to describe application policy in terms of pair-wise communication rules between application components—an interface that is far more compact, intentionally simple and human-friendly than Asbestos labels. Our *policy language parser* is able to translate these high-level policy descriptions to equivalent Asbestos label configurations. Furthermore, developers can use the policy language to describe important run-time application properties that are required to automatically instantiate the application policy using our policy *launcher*.

Secondly, we propose a new mechanism to facilitate security policy debugging in Asbestos, namely *debug domains*. Performing system state inspection—e.g. during debugging—would, if unchecked, leak information from a compartment and violate information flow. Debug domains implement a decentralized debugging primitive that adheres to the information flow policies enforced by Asbestos.

We evaluate our policy language by using it to describe policies from major DIFC systems. We also use synthetic tests to evaluate the effectiveness and performance overhead of debug domains. Our results suggest that our proposed mechanisms are able to assist developers with reasonable overhead, can be beneficial to DIFC systems other than Asbestos, and improve the DIFC programming model.

CHAPTER 1

Introduction

1.1 Security Challenges

Breaches of Web servers and other networked systems routinely divulge private information on a massive scale [Lem05, New05, Tro06, Lem06]. Eliminating all software flaws is extremely difficult, if not impossible, in practice, but systems can be engineered to contain the effects of exploits. Ideally, we would like applications to enforce the *principle of least privilege* [SS75], ensuring that each component holds the minimum privilege required to accomplish its tasks. A system enforcing least privilege policies would prevent a server acting for one principal from accessing data belonging to another principal. A full implementation of the principle of least privilege is a daunting task: fully understanding and specifying the necessary privilege requirements for each application is not much less complex than writing the application itself and would require costly fine grained privilege management mechanisms. Finding a useful way to define privilege is an important goal when designing secure systems. We can achieve similar security improvements for a large class of server applications by pursuing a weaker goal in the same direction, *data isolation*.

Using application-level data isolation one can implement an instance of the principle of least privilege: a policy preventing a server acting on a user's behalf from accessing data belonging to another user. Such a policy, implemented using

a small, trusted part of the application and enforced by the operating system, would prevent whole classes of exploits, making servers much safer in practice. For instance, a typical SQL injection attack would target a Web service with badly formated and poorly checked query forms. Although each user is supposed to have access to her own records, an attacker would use a carefully crafted input that would change a legitimate database query into one that returns information about every user in the system. A system providing data isolation would prevent such an attack since user data would be strictly isolated according to application policy enforced by the operating system. The server process handling the user request would hold privilege to access only that particular user's data and the attempted attack would fail to steal information from the database.

Unfortunately, current operating systems provide little or no means to enforce data isolation. Even the weaker goal of isolating Web services from one another requires complex and error-prone abuse of primitives designed for other purposes [Kro04]. Most servers thus retain the monolithic code design with many privileges, allowing all server components to access all application data. As a result of this highly insecure design, high-impact breaches continue to occur.

1.2 Example Applications

Bugs in the implementation of networked server applications can lead to serious security problems. For instance, Web server bugs have lead to multiple security incidents and it has been shown [Kro04] that isolating Web services is very hard.

The typical architecture of a Web server running on a conventional operating system places the same amount of trust on all Web server components. Figure 1.1 is a simplified illustration of such a Web server with "monolithic" privilege man-

agement architecture: Web server components inside the dashed box run with the same amount of privilege and without any data isolation restrictions between them. Essentially, each of the components inside the dashed box is *trusted*.

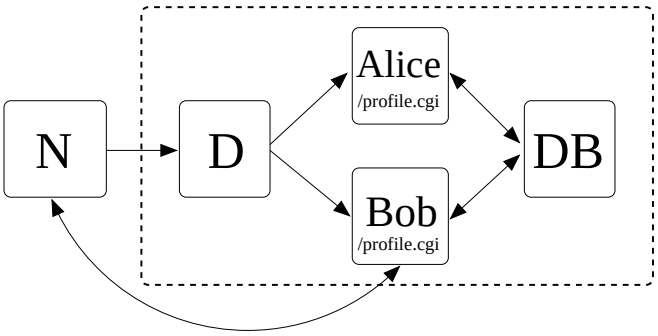

Figure 1.1: A simplified Web server architecture. All Web server components inside the dashed box are equally trusted (i.e. run with the same amount of privilege). All worker processes have equal access to the database—even if they are serving different users.

When user Alice connects to the server through the network, the system network daemon N forwards the connection to the Web server, where it is first handled by a demux process D. The demux despatches the Alice's request to one of the Web server worker processes (i.e. server processes responsible for executing user requests) which acts on behalf of the Alice. Worker processes have equal privilege (and therefore equal access to the database) and service requests by sending results to the remote user, directly through the network daemon. Although Alice's worker acts on her behalf, there is no mechanism restricting it from accessing any information from the database DB—including other users' data. Therefore, the security of the system relies heavily on the assumption that user workers are correctly implemented and not misbehaving.

When Alice's worker, e.g. a CGI script such as "profile.cgi", attempts to access user data from the database DB, the server relies on the CGI script implementation to enforce data isolation/confidentiality and not reveal to Alice information

belonging to other users, such as Bob's social security number.

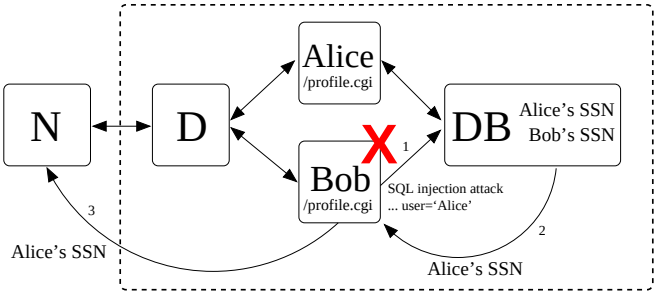

Figure 1.2: Bob exploits a vulnerability in the CGI script and issues a SQL injection attack. The lack of data isolation allows the compromised CGI script to leak Alice's SSN to Bob. First Bob launches a SQL injection attack posing as Alice (1), then the database responds to Bob's worker process with Alice's SSN (2) which leaks it to Bob through the network (3).

By relying on correct implementation of the Web server components for data isolation/confidentiality enforcement, the system's security becomes vulnerable to software bugs. For instance, if profile.cgi contains an exploitable bug and becomes compromised by a malicious user, it can be used to exercise full Web server privilege in any way the attacker wants. Figure 1.2 presents a typical attack scenario: user Bob knows that profile.cgi is poorly implemented (e.g. does insufficient argument checking) and launches an SQL attack allowing him to steal Alice's social security number from the database—or even the whole database for that matter.

Taking into account the recent innovations in the Web services development model, such as Facebook [Faca] and OpenSocial [Ope], server-side security becomes even more fragile if user-uploadable application modules are supported—in order to allow third-party developers to extend the functionality of the service. In this model, server-side security is not only threatened by software bugs introduced by otherwise trusted developers, but it also faces the threat of malicious user-uploaded code. For example, most interesting services provided by third-

party modules would require database access. In our example of Figure 1.1, if Bob is able to run his own code on the server, it is no longer possible to rely on the Web service implementation to enforce confidentiality, since Bob is considered untrusted. Nevertheless, it is necessary to ensure that Bob can not export the whole database by uploading a carefully crafted third-party application. The data isolation/confidentiality risks in this development model are particularly concerning, since server-side security depends on the legitimacy of third-party developers.

The security weaknesses presented in our Web server examples demonstrate how conventional networked server applications can be vulnerable to attacks: coarse-grained privilege management leads to overly privileged application modules and makes it very hard—if not impossible—to provide data isolation in the presence of bugs. In the example of Figure 1.2, the demux may need to hold certain types of privilege in order to perform its tasks. For example, if the demux is responsible for authenticating incoming users, then certain amount of trust is necessary to be placed on D. Similarly, the database is responsible for saving each user's private information and therefore is considered a trusted application component with increased privilege requirements. Bob's worker process though only needs to access Bob's data, as opposed to having privilege to access the whole database. A mechanism ensuring that Bob's worker (and for that matter all untrusted application components) holds limited database access privilege, allowing it to serve Bob's request without being able to access other user's information, would prevent Bob from stealing Alice's social security number despite the existence of an exploitable software bug. Such a mechanism would also address the security concerns in the example of third-party applications: a worker uploaded by Bob would run on the server holding only a very limited set of privilege, therefore preventing it from leaking data not belonging to Bob.

In order to provide the desired level of data isolation and fine-grained privilege management for server applications we must introduce new operating system primitives [KEF05] enabling application developers to express and implement the necessary security policies. To this end, we have designed and implemented the *Asbestos* [EKV05] operating system, whose novel labeling mechanism provides data isolation by implementing *decentralized information flow control*.

1.3 Decentralized Information Flow Control

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 [Dep85, LS01, WMV03, KH84]. IFC's *label* formalism [Dep85, Den76, MR92] can implement data isolation security policies such as secrecy protection (preventing protected information from escaping a system) and integrity protection (preventing untrusted 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. For example, operating systems using label variants to implement Mandatory Access Control (MAC) [LS01, WMV03] provide strong, end-to-end security guarantees on application effects, but centralize policy management privilege. This has security benefits, but important application policies often require a form of privilege. 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 [ML00], or DIFC, addresses these problems by decentralizing the notion of privilege. 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.

Asbestos labels give application developers fine-grained control over different types of privilege, allowing them to significantly reduce the amount of privilege held by each application component. Developers are able to express diverse security policies using the fundamental *compartment* isolation primitive: applications can create an arbitrary number of compartments and define the rules that govern information flow between compartments as well as the rest of the system. These policy rules are strictly enforced by the Asbestos kernel across all processes in the system. This way, Asbestos is successful in providing data isolation, therefore containing the effects of security breaches caused by a number of possible reasons such as exploitable software bugs, misconfiguration, poor software design, and so forth.

Figure 1.3 presents a Web server design that uses fine-grained privilege control to address the data isolation/confidentiality problems of Figure 1.1. The worker processes serving Alice and Bob's requests are not considered trusted application components and do not need to run with full Web server privileges. Instead, they

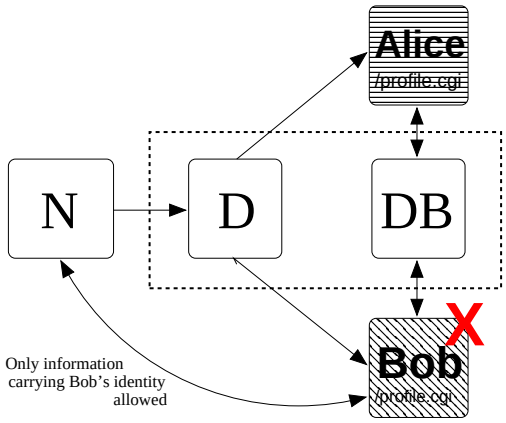

Figure 1.3: Using DIFC a developer could remove the worker processes from the set of trusted applications components. The demux and the database each hold some amount/kind of trust still, but Bob's worker belongs to Bob's compartment (marked with the diagonal line pattern), making it impossible to receive information belonging to a user other than Bob. Furthermore, allow information leaving Bob's compartment (i.e. Bob's worker) are marked (contaminated) with his identity. Even if the CGI script is buggy, DIFC rules prevent Bob stealing and leaking Alice's information.

are executed inside each user's private, isolated compartment (represented by the horizontal and diagonal lines for Alice and Bob respectively), allowing them to receive data belonging only to that particular user. Additionally, any data leaving Bob's compartment carry his identity (contamination) and only processes with relevant privilege may receive it. For instance, using Asbestos DIFC, the communication channel between Bob's worker and the network daemon can be setup so that it holds privilege to accept only Bob's contamination. If Bob is trying to leak data belonging to Alice over the network, his messages will get rejected since the communication channel is not privileged to receive data carrying Alice's contamination. If the Web service allows third-party applications, the data isolation guarantees provided for each user worker still stand: each worker can only receive (and export) information it has been cleared to receive.

Asbestos labels allow developers to use information flow control to implement

a variety of data isolation policies, like the one in our Web server example, in a decentralized fashion that requires no special system privilege.

1.4 System Management Challenges

The decentralized nature of Asbestos gives developers great flexibility by allowing them to manage application security policies without requiring any type of special privilege. Unfortunately, this benefit comes at a price: the responsibility of application security policy management is shifted to developers, involving policy management tasks of critical importance for application security.

First, developers must define the correct application policy and write privileged application code that will implement it by manipulating labels. Although the threat to the security of the system posed by application bugs is significantly lessened by data isolation, the correct behavior of individual application modules relies on the correctness of the label manipulations performed by the application developer.

Secondly, developers are responsible for debugging poorly defined and/or implemented application security policies, given the constraints imposed by the system's strong data isolation guarantees.

In our experience with Asbestos's system-based DIFC [VEK07], as well as other DIFC systems [ML00, Mye99, ZBK06, KYB07], these security policy management problems are difficult enough to hamper adoption. Although operating systems typically provide mechanisms that can be used to build system management tools, DIFC systems have failed to provide adequate support for such DIFC-safe system management. In this thesis we focus on the two major categories of system management mechanisms that are particularly important for

DIFC systems: *security policy management* and *security policy debugging* mechanisms.

Policy management mechanisms aim to provide users with means to express and reason about security policies. DIFC policy management is critical for security and has been proven very difficult in real life. 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.

Over-simplified security policy management mechanisms may prove inadequate, with limited ability to express diverse application policies. Highly flexible, fine grained security policy mechanisms may, on the other hand, prove to be error prone and hard to use. Ideally, the system should provide effective policy specification and enforcement mechanisms that facilitate administrative tasks while keeping the likelihood of misconfiguration minimal.

Security policy debugging mechanisms make it possible for users to collect information regarding security policy-related matters. Such information may be used to identify poor policy definition, implementation errors, malfunctioning policies or just to keep track of system activity.

Let us consider a simple example demonstrating how information flow control complicates debugging. Most operating systems provide a way to enumerate processes in the system, along with information about them. In UNIX-like operating systems this mechanism is used by the *ps* command, allowing users to get information such as the name of the process, the user it belongs to, how many

resources the process is using, etc. The *ps* command—or any equivalent—exposes information about each process's state to the calling user. Such information exposure in Asbestos should not violate the information flow rules. This effectively means that an Asbestos equivalent of the *ps* functionality needs to be implemented in an information flow aware manner, ensuring that there is no explicit information leakage and that implicit flows are minimized.

Message exchange rules enforced by trusted components (such as the operating system kernel) are used to prevent explicit information leakage. Implicit information flow, or *covert channels*, leak information by modulating some observable property of the system (e.g. the number of processes belonging to a user may leak an integer while the existence of a file conveys one bit). In the *ps* example, any observable property of a process—including its existence—can potentially leak information, if modulated appropriately. Just the simple task of notifying a user that one of her processes exited, requires care in order to preserve information flow rules.[1]

One can say that all system management tasks in Asbestos require special care to ensure that information flow rules are not violated. For instance, a task that has been proven daunting in practice is policy debugging. 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. The mere existence of the bug is subject to information flow rules and is concealed by the kernel. DIFC systems' strong in-

[1]Actually, we strongly believe that a leak-free equivalent of the *ps* command may be impossible in Asbestos, due to both explicit and implicit information flows.

formation flow guarantees limit the ability to collect system information required to perform the system management tasks required to solve such bugs. Increasing system visibility for users and developers would violate the information flow constraints the system is trying to enforce. Likewise, Asbestos's strong isolation guarantees affect system management significantly. In Asbestos we aim to expose information required by system management tools while preserving the system's security guarantees: ***information flow policies should not be violated***.

Policy debugging requires system state inspection and exposure. and the relevant mechanisms should allow users to access only information they are cleared to receive. Additionally, exposure to such information (e.g. sending debugging information to the calling user) should be tracked by the information flow control system based on the policies in effect, just like any other information flowing through the system. Exposure of such information involves privilege: users should have access only to system state that are cleared to inspect and debugging information should escape a DIFC compartment if and only if an adequately privileged process removes all relevant contamination—i.e., *declassifies* the information. These privilege requirements necessary for DIFC-safe policy debugging can be better addressed by new mechanisms able to systematically model such types of privilege—e.g., debugging privilege. To preserve the decentralized nature of DIFC, this new type of privilege should also be managed in a decentralized fashion, allowing each user to create and use it without requiring any kind special system privileges.

Problems such as security policy management and debugging are two important instances of the greater issue we have identified with DIFC systems: *current DIFC systems provide very limited—if any—mechanisms to support system management tasks*. Yet system management mechanisms are essential for achieving

the reasonable, user-friendly programming model necessary for wider DIFC adoption.

1.5 Goals and Contributions

When designing Asbestos, our higher level goal was to make computing more secure by providing mechanisms that contain the effects of application bugs. Although Asbestos labels are a primitive capable of serving this goal, developing for Asbestos we observed that it exposes a programming model that makes it particularly unappealing. Despite all the benefits of DIFC, a DIFC application is effective only to the extent that application developers are able to use and take advantage of the mechanisms provided.

Our goal is to improve the DIFC programming model and make it easier to use, reason about and adopt. We believe that by addressing some of the pressing system management issues we can significantly improve the programming model. We investigate the challenges related to two important system management problems: security policy and system state management. Our contributions include two mechanisms incorporated to Asbestos to make privilege-separated applications easier to design, build, and debug. Although our work has focused primarily on Asbestos, we believe that the solutions we propose have applications to other DIFC systems as well.

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 starting 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 associated processes are not allowed to see—with usability.

1.6 Outline

The remainder of this thesis is organized as follows.

Chapter 2 presents related work while Chapter 3 presents the Asbestos operating system, including basic goals and design principles , the Asbestos labeling mechanism and its use in implementing DIFC (Section 3.1), as well as examples, applications and lessons learned from developing for Asbestos (Sections 3.2 and 3.5). Chapter 4 introduces the requirements for DIFC-safe policy management and debugging mechanisms. Chapter 5 presents the policy description language and the relevant tools we have proposed and implemented in order to facilitate DIFC policy management. Chapter 6 presents Asbestos debugging challenges as well as the design and implementation of the debug domain primitive used to facilitate debugging and other system state management tasks. Fi-

nally, Chapter 7 summarizes this thesis, presents some concluding remarks and potential directions for future work.

Appendix A presents full policy descriptions of examples from Asbestos, HiStar and Jif—including the full AWS policy—using our policy language.

CHAPTER 2

Related Work

Most MAC systems are geared towards military specifications, which require labels to specify at least 16 hierarchical sensitivity classifications and 64 non-hierarchical categories [Dep85]. This label format severely limits what kinds of policies can be expressed. The fixed number of classifications and categories must be centrally allocated and assigned by a security administrator, preventing applications from crafting their own policies with labels alone. Thus, MAC systems typically combine labels with a separate discretionary access control mechanism. Ordinary Unix-style users and groups might enforce access control within the secret, nuclear level. This structure allows users to have limited control over policy management using the discretionary access control mechanism available to them. However, these discretionary mechanisms are not adequate to implement more complex server policies, and application developers may not perform policy management and debugging tasks without holding special system privilege.

Variants of mainstream operating systems have used label variants to implement MAC. SELinux [LS01] and TrustedBSD [WMV03] are based on implementations of the Flask MAC architecture [SSL99], which employs a special kernel component called the Security Server (SS), and require administrator privilege to perform policy management tasks. The SELinux Policy Server [MBM06] has tried to correct this shortcoming by adding a meta-policy; the policy specifies which subjects can modify the policy in what way. However, the security administrator

must still anticipate and approve of the policy structure of every individual application. Furthermore, policies are defined in multiple files, therefore increasing the dispersion of trusted code. These restrictions prevent applications from using MAC primitives as security tools without the cooperation and approval of the security administrator.

Legacy systems suffer from similar policy management problems: the idea of dynamically adjusting labels to track potential information flow dates back to the High-Water-Mark security model [Lan81] of the ADEPT-50 in the late 1960s. Numerous systems have incorporated such mechanisms, including IX [MR92] and LOMAC [Fra00]. The ORAC model [MMN90] supported the idea of individual originators placing accumulating restrictions on data, somewhat like creating tags, except that data could still only be declassified by users with the privileged Downgrader role. This effectively means that managing policies and declassifying debugging information to users requires some kind of special system privilege.

Asbestos and similar research operating systems—such as HiStar [ZBK06] and Flume [KYB07]—that implement DIFC using Asbestos labels, allow any process to dynamically create non-hierarchical compartments. An application can manage arbitrary security policies involving compartments it creates, without requiring any type of system privilege. This makes Asbestos's decentralized MAC an effective tool for application *and* administrative use. Nevertheless, the strict isolation enforced by DIFC policies defined by developers makes policy correctness critical and introduces new system management challenges such as those addressed in this thesis.

Debugging in SELinux and TrustedBSD 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. The decentralized privilege management model of Asbestos contrasts the use of such centralized, highly privileged system state management mechanisms. By design, DIFC systems have no "super-user" privilege that can override security policies and be exercised to perform debugging in violation. Application defined policies are *always* enforced.

Mandatory access control can also be achieved with unmodified traditional operating systems through virtual machines [Gol73, KZB90]. For example, the NetTop project [VMw01] uses VMware for multi-level security. Virtual machines have two principal limitations, however: performance [KC03, WSG02] and coarse granularity. One of the goals of Asbestos is to allow fine-grained information flow control, so that a single process can handle differently labeled data. To implement a similar structure with virtual machines would require a separate instance of the operating system for each label type. This type of strict, coarse-grained data isolation precludes the safe exchange of private data between mutually untrusted programs, possibly including debugging information. Additionally, it is very difficult to implement complex policies using VMs (e.g., policies that support any type of IFC-safe data sharing), and—unlike DIFC—policy management requirements are relatively simple.

Additionally, due to the large number of separate virtual machine instances, system management tasks in such a setup would be extremely difficult and costly.

The system management challenges we faced while developing Asbestos applications such as the Asbestos Web server [VEK07] were the main motivation for our work. HiStar [ZBK06] 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 [KYB07] 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 [ML00] and JFlow [Mye99] 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 [CVM07] and the Swift framework [CLM07], 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.

Policy management and debugging challenges become particularly concerning when unprivileged third-party users attempt to contribute code to server applications. To avoid the consequences of any breach in a server application, existing attempts at server extensibility expose only a fraction of the server's resources and private data. Livejournal [liv], for example, allows journal authors to upload sandboxed PHP renderers for their journals, but each renderer can read only a limited set of user data accessible through a strict API and has very limited write access. Similarly, Facebook [Faca] supports user-uploadable, third-party applications, written using a very restrictive API. In order to protect system security, third-party developers have very limited freedom to define interesting custom subpolicies but they are able to expose any kind of system information they have access to, with no additional security restrictions, since they API is supposed to restrict their access to private information. Recent security incidents [Facb, Facc] demonstrate that limiting third-party developers by means of narrow APIs is difficult, and the slightest mistake may compromise the security of the system. Seeing that the requirement for strong server security contrasts the increasing demand for more powerful and diverse third-party applications, we expect that more sophisticated security mechanisms, such as DIFC, will become more popular with Web service providers. If Web application developers (both trusted and untrusted) need to use DIFC mechanisms, the need for more elaborate, DIFC-aware policy management and debugging tools will become pressing.

Furthermore, several earlier systems have used confinement to safely execute untrusted programs. Web browsers and active network systems like ANT [Wet99] execute untrusted Java [GJS05] programs by running them in a restricted virtual machine. Browsers confine the untrusted programs by restricting the disk and network access of the Java virtual machine. The ANT execution environment further restricts untrusted code by limiting the Java language as well. An untrusted third-party server extension must be able to read, write and process data as long as it does not export it off the server in a way that violates the information-flow rules. VM-based solutions for running untrusted Web applications pose policy management and debugging problems similar to those of regular VMs: coarse granularity leaves little policy management flexibility, while at the same time a virtual machine relinquishes all control over data that it exports to other VMs for processing by another user's untrusted module, making it hard to impose any type of flow control on debugging information.

Almost all Web srevices use a database back-end to store application data. Existing database systems [LM02, RB04] also support per-row security labels based on users. Such database labels though have semantic meaning only within the context of the database application and do not have a close integration with operating system labels. This greatly simplifies the design of the database row labeling mechanism and makes policy management a lot easier. For instance, Oracle security labels [Ora07] associate one clearance label with each user and one secrecy label per row. This simplified security label mechanism is centrally managed by a database/security administrator either through graphical tools or by using a basic scripting language. Asbestos labels though are a decentralized and far more versatile security mechanism that is closely integrated with the operating system, requiring specialized policy management tools, such as our policy language.

CHAPTER 3

The Asbestos Operating System

We investigate the DIFC policy management and debugging challenges in the context of the Asbestos operating system. We present an overview of the Asbestos operating system, its goals and design. This will make the challenges we have addressed and our proposed solutions clearer and more understandable.

The Asbestos operating system can provide better security by containing the effects of exploitable software bugs. In a nutshell, Asbestos aims to achieve a goal that is very hard—if not impossible—to achieve on a conventional operating system: *Asbestos should support efficient, unprivileged, and large-scale server applications whose application-defined users are isolated from one another by the operating system, according to application policy.* A *large scale server application* responds to user requests—usually coming from the network. These server applications, such as Web servers, support large number of dynamically changing *application-defined users* that request service at random times. In order to contain the effects of software bugs and remedy security issues such as the ones that have lead to huge data leaks in the past [Lem05, New05, Tro06, Lem06], Asbestos server applications make use of Asbestos's mechanisms to provide strict *user data isolation.* A server process acting on behalf of an application-defined user (e.g. user of a particular Web service that does not necessarily correspond to a system user) cannot have access to another user's private data.

Given the diverse requirements of each server application, Asbestos allows

developers to define and implement such security policies without requiring any special type of system privilege. These *application-defined policies* are strictly enforced by the operating system. This way, applications are responsible to define the necessary policies, but they are not trusted to enforce them, since only a minimal set of application components are considered trusted by the kernel. The enforcement of the policies is assigned to the trusted Asbestos kernel, that ensures that the policy is in effect even if the untrusted parts of the application have been compromised. Of course, the system mechanism providing this added security must incur reasonable overhead, that does not affect application *efficiency*.

The contributions of Asbestos are twofold. First, we use *Asbestos labels* to track information flowing through the system. Asbestos labels are used to track what kinds of information an entity has already received and determine the kinds of information it may receive, therefore specifying the ways information may flow in the system. Furthermore, Asbestos labels can track *privilege* with respect to categories of information. Labels are able to enumerate positive rights like traditional discretionary capability systems, but unlike traditional capability system, they are also able to track and control the flow of information. Privilege management is performed at the label level and each process holds privilege for any new category of information it creates. This *decentralized information flow control* (DIFC) mechanism makes Asbestos labels a powerful mechanism able to express diverse policies including capability-like policies, traditional multi-level security (MLS) as well as application-specific isolation policies.

Second, Asbestos's *event process* (EP) abstraction is used to replace monolithic server processes and efficiently implement isolated, per-user server process instances. A monolithic server process that handles private user data, would either have to hold excessive privilege with respect to all users, or quickly become overly

contaminated (and unusable) by accessing multiple user's information. Event processes solve this problem securely, efficiently and with minimal resource overhead. Each server process that uses EPs keeps private state for each user, but isolates that state so that a compromised process may only expose one user's data.

Asbestos [VEK07] is a message-passing operating system; its IPC mechanism resembles that of micro-kernels such as Mach. The fundamental IPC primitive is a message sent from one process to an Asbestos communication end-point, called a *port*. Applications are able to construct and tear down an arbitrary number of isolation domains, which we refer to as *compartments*. Most resources in Asbestos, including ports and compartments, are represented by a unique[1] 61-bit identifier—called *tags*—generated by the kernel. To the kernel tags are opaque; applications give them semantic meaning. A process may create an arbitrary number of compartments and ports by requesting from the kernel a new, unique tag for each of them. The allocating process gains privilege over that tag, meaning that the process can freely manipulate information flow for that tag.

3.1 Asbestos Labels

Asbestos labels are used to control and track information flow, so as to provide isolation of user data according to application policies and prevent data leaks during communication. An Asbestos label is a function mapping between tags and sensitivity *levels*, each representing a process's amount of privilege (or contamination level) with respect to the compartment represented by the tag. Each process has a *tracking label* and a *clearance label*. The tracking label $\mathbf{T}_P$ of process P keeps track of the compartments whose information has been exposed to

[1]Each identifier is generated upon request and uniqueness is ensured through the use of a Blowfish cryptographic hash.

P, either directly or indirectly. We often refer to this exposure as the *contamination* that the process has received. The clearance label $\mathbf{C}_Q$ of process Q governs the ability of the process to receive messages with respect to each compartment. The clearance label specifies the kind of contamination the process is allowed to receive.

Asbestos ports also carry labels. Port labels specify the amount of contamination (or privilege) a message destined to the port may carry, therefore resembling clearance labels. Processes supply an initial port label when creating a port; most often this is $\top = \{\mathbf{3}\}$, which adds no restrictions relative to the process's clearance label, but it can be $\{\mathbf{2}\}$ or anything else. Port labels augment the expressiveness of Asbestos labels, making it possible to implement more diverse policies.[2]

There are five levels a tag may be at inside a label. Privilege is represented by the special level $\star$. For instance, when a process P allocates a new tag b it acquires privilege with respect to that tag, and its tracking label would reflect this by including the tag-level pair $\{b\star\}$. This allows P to declassify information with respect to the compartment represented by b, as well as pass this privilege on to any other process it can communicate with. The other levels allow Asbestos to combine secrecy and integrity tracking into a single name-space (more usually, systems track secrecy and integrity using separate labels [MR92, KYB07]). These levels, written (in increasing order) **0**, **1**, **2** and **3**, have the meanings shown in Table 3.1. Notice that $\star < \mathbf{0}$ ($< \mathbf{1} < \mathbf{2} < \mathbf{3}$).

A label combines explicit levels for zero or more tags with a *default level* that applies to all tags not otherwise mentioned. The default level for a process' tracking label is **1**, while the default for a clearance label is **2**. Level **3**, on the other hand, represents contamination with respect to a tag in a label. In a clearance

[2]For instance, port labels are necessary to implement capabilities using Asbestos labels.

Level	Meaning
$\star$	Privilege for the tag
0	High integrity for the tag
1	Default tracking level, default integrity and secrecy—no restriction for the tag
2	Default clearance level, low integrity (used in taint tracking)
3	High secrecy (contamination) for the tag

Table 3.1: Asbestos label levels and common uses

label, **3** represents the fact the process is cleared to receive the corresponding contamination. Note that $\star < \mathbf{1} < \mathbf{3}$ and that the default level for a clearance label does not allow contamination (i.e. $\mathbf{2} < \mathbf{3}$).

We represent a label as a set of tag-level pairs followed by the default level for that label. For instance, the label $L = \{a\,\mathbf{0}, b\,\star, c\,\mathbf{3}, \mathbf{1}\}$ specifies $L(a) = \mathbf{0}$, $L(b) = \star$, and $L(c) = \mathbf{3}$, while for any other tag t, $L(t) = \mathbf{1}$, or:

$$L(t) = \begin{cases} \mathbf{0} & \text{if } t = a, \\ \star & \text{if } t = b, \\ \mathbf{3} & \text{if } t = c, \\ \mathbf{1} & \text{otherwise.} \end{cases}$$

Assuming that label L is the tracking label of process P (also denoted by $\mathbf{T}_P$), $L(b) = \star$ signifies that P privileged with respect to tag b (and the compartment, port or other resource b represents). Similarly, $L(a) = \mathbf{0}$ means that process P is high integrity with respect to a (it has been modified and/or affected only by privileged and/or high integrity entities with respect to a), while $L(c) = \mathbf{3}$ signifies that P is marked high secrecy with respect to c (i.e. P has been exposed to information carrying contamination with respect to the c compartment). Notice that if L was process P's clearance label (also denoted by $\mathbf{C}_P$) the meaning of

the levels would be analogous, only referring to P's security clearance—i.e. P's ability to receive information at the relevant level. For instance, $L(c) = \mathbf{3}$ at P's clearance label would mean that P is cleared to receive messages marked as high secrecy with respect to c (i.e. contaminated with c), while $L(a) = \mathbf{0}$ would mean that P may only receive messages whose sender is high integrity with respect to a.

3.1.1 Communication Rules

In Asbestos label terms, in order for P to send a message to Q, the kernel requires that P's tracking label is "less than or equal" to Q's clearance label, essentially requiring that *for each tag t in the system $\mathbf{T}_P(t)$ is less than or equal to $\mathbf{C}_Q(t)$*:

$$\mathbf{T}_P \sqsubseteq \mathbf{C}_Q \tag{3.1}$$

where

$$\mathbf{T}_P \sqsubseteq \mathbf{C}_Q \quad \text{iff} \quad \forall t : \mathbf{T}_P(t) \leq \mathbf{C}_Q(t). \tag{3.2}$$

This fundamental rule, which is enforced by the Asbestos kernel during message exchange, ensures that the receiver has clearance to receive the contamination carried by the message. If this label check fails, the message is silently dropped by the kernel. When the message is delivered to Q, the kernel automatically updates Q's tracking label to reflect information flowing from P to Q. This is done by setting Q's tracking label to the least-upper-bound of the two processes' tracking labels:

$$\mathbf{T}_Q \leftarrow \mathbf{T}_Q \sqcup \mathbf{T}_P \tag{3.3}$$

The least upper bound operator works by taking the highest level in either of the

System labels (maintained by the kernel)	
$\mathbf{T}_P$	Process P's tracking label.
$\mathbf{C}_P$	Process P's clearance label. The system maintains the invariant that $\mathbf{T}_P \sqsubseteq \mathbf{C}_P$.
$\mathbf{PC}_p$	Port p's port clearance label. May be set arbitrarily by p's owning process.
Discretionary labels (optionally set by the process sending a message)	
$\mathbf{T}^+$	Increases the message's effective tracking label. Used when a message contains data at a higher secrecy level than the process itself. Commonly used by privileged processes; defaults to $\{\star\}$, which implies no increase.
$\mathbf{T}^-$	Decreases the receiving process's tracking label. This represents declassification and/or privilege transfer. Defaults to $\{\mathbf{3}\}$, which implies no decrease; setting it to another value requires privilege for the affected tags.
$\mathbf{C}^+$	Increases the receiving process's clearance label, granting it clearance to further raise its tracking label. Defaults to $\{\star\}$, which implies no increase; setting it to another value requires privilege for the affected tags.
$\mathbf{V}$	Declares an upper bound on the sender's tracking label. This might represent the privilege the sender intends to use for this operation. The kernel verifies $\mathbf{V} \sqsubseteq \mathbf{T}_P$, then passes it to the receiver along with the message. Defaults to $\{\mathbf{3}\}$, which confers no information.

Figure 3.1: Notation and description for Asbestos system and discretionary labels.

labels for each tag:

$$\forall t, \mathbf{T}_Q(t) = max(\mathbf{T}_P(t), \mathbf{T}_Q(t)) \tag{3.4}$$

For instance, let us assume that P's send and clearance labels are $\mathbf{T}_P = \{a\,\mathbf{3}, b\star, \mathbf{1}\}$ and $\mathbf{C}_P = \{a\,\mathbf{3}, b\,\mathbf{3}, \mathbf{2}\}$ respectively, while Q's are $\mathbf{T}_Q = \{b\,\mathbf{3}, \mathbf{1}\}$ and $\mathbf{C}_Q = \{a\,\mathbf{3}, b\,\mathbf{3}, \mathbf{2}\}$. In this example P is contaminated with $\{a\,\mathbf{3}\}$ but is able to send a message to Q, since $\mathbf{C}_Q(a) = \mathbf{3}$ denoting that Q holds the necessary clearance with respect to a. The Asbestos kernel enforces *transitive contamination* to track information flow. Any message sent from P to Q carries P's contamination and, once received, contaminates Q accordingly. In our example, P is contaminated with $\{a\,\mathbf{3}\}$ and Q has clearance to receive that contamination. After Q has received a message from P, its tracking label will become $\mathbf{T}_Q = \{b\,\mathbf{3}, a\,\mathbf{3}, \mathbf{1}\}$.

Apart from the sender's contamination, Asbestos messages may carry any of the four optional types of discretionary labels, $\mathbf{T}^+$, $\mathbf{T}^-$, $\mathbf{C}^+$ and $\mathbf{V}$, described in Figure 3.1. The four optional discretionary message labels are key in achieving Asbestos's flexibility in decentralized policy implementation and privilege man-

agement. Through the use of discretionary labels processes are able to transfer contamination, exercise and transfer privilege according to application policy.

The $\mathbf{T}^+$ discretionary label is used to increase the contamination carried by the message with respect to a set of tags. Notice that increasing contamination makes the system more restrictive and thus the use of $\mathbf{T}^+$ requires no privilege with respect with the tags involved. In the presence of a $\mathbf{T}^+$ message label, the *effective tracking label* ($\mathbf{T}_E$) defining the contamination carried by a message sent from P to Q is the least-upper-bound of the contaminate label and the P's tracking label:

$$\mathbf{T}_E \leftarrow \mathbf{T}^+ \sqcup \mathbf{T}_P \tag{3.5}$$

and similarly to rule 3.3, Q's tracking label would be set to:

$$\mathbf{T}_Q \leftarrow \mathbf{T}_Q \sqcup \mathbf{T}_E \tag{3.6}$$

Notice that rule 3.7 does not take into account the privilege already held by the receiver: if $\mathbf{T}_Q(t) = \star$, while $\mathbf{T}_E(t) > \star$ then the strict enforcement of the least-upper-bound operator will clobber Q's privilege with respect to t. Given that in Asbestos processes that have privilege with respect to a tag are immune to contamination from that tag (i.e. privilege is maintained), we need to adjust rule 3.6 to "preserve stars":

$$\mathbf{T}_Q \leftarrow \mathbf{T}_Q \sqcup (\mathbf{T}_E \sqcap \mathbf{T}_Q^\star) \tag{3.7}$$

where[3]:

$$(\mathbf{T}_E \sqcap \mathbf{T}_Q^\star)(t) = \begin{cases} \star & \text{if } \mathbf{T}_Q(t) = \star, \\ \mathbf{T}_E(t) & \text{otherwise.} \end{cases} \tag{3.8}$$

[3]This implies that $\mathbf{T}_Q^\star = \{t_1 \star, t_2 \star, \ldots, t_N \star, \mathbf{3}\}$ and for all $i = 1, 2, \ldots, N$ $\mathbf{T}_Q(t_i) = \star$.

$\star, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}$	Label levels, in increasing order
$\mathbf{T}_P$, $\mathbf{C}_P$	Process P's label and clearance
$L_1 \sqsubseteq L_2$	Label comparison: true iff $\forall h$, $L_1(t) \leq L_2(t)$
$L_1 \sqcup L_2$	Least-upper-bound label: $(L_1 \sqcup L_2)(t) = \max(L_1(t), L_2(t))$
$L_1 \sqcap L_2$	Greatest-lower-bound label: $(L_1 \sqcap L_2)(t) = \min(L_1(t), L_2(t))$

send(*port*, data, $\mathbf{T}^+, \mathbf{T}^-$, $\mathbf{V}, \mathbf{C}^+$)
Let P be the sending process
Let Q be the process with receive rights for *port*
Let $\mathbf{T}_E = \mathbf{T}_P \sqcup \mathbf{T}^+$
Requirements:
(1) $\mathbf{T}_E \sqsubseteq (\mathbf{C}_Q \sqcup \mathbf{C}^+) \sqcap \mathbf{V} \sqcap \mathbf{PC}_{port}$
(2) If $\mathbf{T}^-(t) < \mathbf{3}$, then $\mathbf{T}_P(t) = \star$
(3) If $\mathbf{C}^+(t) > \star$, then $\mathbf{T}_P(t) = \star$
(4) $\mathbf{C}^+ \sqsubseteq \mathbf{PC}_{port}$
Effects:
Grant $\mathbf{T}^-$ and contaminate with $\mathbf{T}_E$, but preserve $\mathbf{T}_Q$'s $\star$ tags
$\mathbf{T}_Q \leftarrow (\mathbf{T}_Q \sqcap \mathbf{T}^-) \sqcup (\mathbf{T}_E \sqcap \mathbf{T}_Q^\star)$
$\mathbf{C}_Q \leftarrow \mathbf{C}_Q \sqcup \mathbf{C}^+$

Figure 3.2: Summary of basic Asbestos label operations

The $\mathbf{T}^-$ discretionary label is used to grant privilege to the recipient of the message by lowering the level of the receiver's label with respect to certain tags (greatest-lower-bound operator), while the $\mathbf{C}^+$ label is used to grant clearance to the receiver, by raising its clearance label. Exercising the $\mathbf{T}^-$ and $\mathbf{C}^+$ labels requires privilege with respect to the relevant tags. Given the $\mathbf{T}^-$ label, the receiver contamination rule 3.7 becomes:

$$\mathbf{T}_Q \leftarrow (\mathbf{T}_Q \sqcap \mathbf{T}^-) \sqcup (\mathbf{T}_E \sqcap \mathbf{T}_Q^\star) \tag{3.9}$$

while the $\mathbf{C}^+$ discretionary label alters the the global communication rule 3.1 as follows:

$$\mathbf{T}_E \sqsubseteq (\mathbf{C}_Q \sqcup \mathbf{C}^+) \tag{3.10}$$

The $\mathbf{V}$ label is used by the sender in order to explicitly state the privilege she wants to exercise during the message exchange, so as to prevent inadvertent use of ambient authority (the "confused deputy" problem [Har88]). During message exchange the $\mathbf{V}$ label is checked so that the sender does not attempt exercise more privilege that she currently holds. This is ensured by applying the greatest-

lower-bound operator between the right-hand side of rule 3.10 and the $\mathbf{V}$ label:

$$\mathbf{T}_E \sqsubseteq (\mathbf{C}_Q \sqcup \mathbf{C}^+) \sqcap \mathbf{V} \tag{3.11}$$

In order for the communication rules to be complete, we also need to take into account the destination port label. The port label doesn't not affect receiver contamination, but it does takes part in the rule determining whether the message can go through or not, by requiring that the message does not carry more contamination than the port label is willing to permit. Consequently, rule 3.11 becomes:

$$\mathbf{T}_E \sqsubseteq (\mathbf{C}_Q \sqcup \mathbf{C}^+) \sqcap V \sqcap \mathbf{PC}_{port} \tag{3.12}$$

Figure 3.2 summarizes the basic Asbestos label operations, including the information flow checks for sending a messages and the updates to the sender's labels.

3.2 Example

With most of the mechanisms for Asbestos labels in place, we present an example with four processes to make labels more concrete. The example, shown in Figure 3.3, involves a trusted multi-user file server, shells for users A and B, and a terminal to which user A is logged in. The application policy is that no process but user A's terminal can export user A's data. For the purposes of this example, we assume that process labels are assigned out of band.

Each user needs a security compartment, so we assign each user a tag. In this instance, a tag is used to represent a compartment and each user's data are isolated in a separate compartment; we represent user A's compartment through

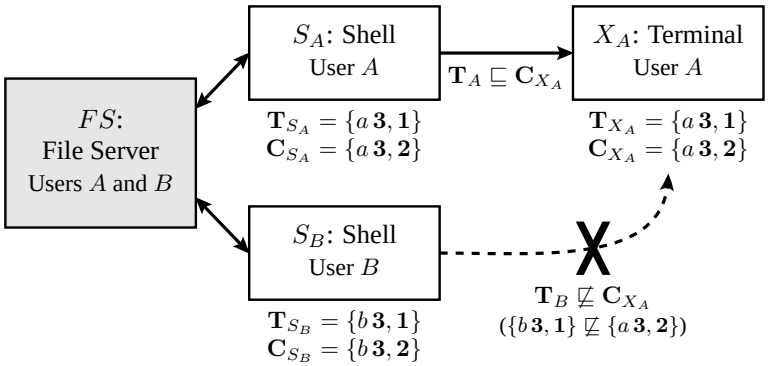

Figure 3.3: Simplified process communication with labels. The file server is trusted.

tag a. The next step is to differentiate processes that have seen A's private data from those that have not. Since we want to use the information flow capabilities of Asbestos labels, we need to use a level higher than level **1**, the default level for send labels. (With level **1** or lower, the contamination rule would not propagate the information flow.)

The default behavior for this policy is to deny communication, so the labels used to implement the policy should prevent communication if the destination process has a default clearance label. Therefore, the only viable contamination level for a process that has seen A's data is $\{a\,\mathbf{3}\}$; a contamination of $\{a\,\mathbf{2}\}$ would allow communication with default receive labels. As a result of using **3** for contamination, the clearance labels of processes that should be able to receive A's data have to be changed. Raising clearance labels, or granting *clearance* for a particular kind of contamination makes the system more permissive, so it requires special privilege: processes are not free to raise their clearance labelsarbitrarily.

Figure 3.3 shows the resulting system. The shell processes S_A and S_B are contaminated with a and b (that is, $\mathbf{T}_{S_A}(a) = \mathbf{3}$ and $\mathbf{T}_{S_B}(b) = \mathbf{3}$), and their clearance labels allow them to receive the data of their respective users. Any processes they

create or are able to communicate with will have the same characteristics. User A's terminal, X_A, has the same labels as S_A. S_A can send messages to X_A, since $\mathbf{T}_{S_A} \sqsubseteq \mathbf{C}_{X_A}$, but S_B cannot, since $\mathbf{T}_{S_B}(b) > \mathbf{C}_{X_A}(b)$, and neither can any other process that has seen B's data.

3.3 Event Processes

Asbestos services often expect to respond to many differently-contaminated requests over time. Server processes that handle multiple users' data—each with a different type of contamination—present a challenging information flow problem: any such server implemented as a single process would soon become contaminated with multiple users' tags. The server could then spread this over-contamination and further complicate the problem. Eventually, the server process will become unable to perform its tasks and essentially unusable.

We could address this problem by granting the server process privilege with respect to *all* user private compartments, therefore making the server immune to all user contamination. This would render the server process a trusted application component exposed to all kinds of potential attacks. Any compromise of the server process and every exploitable bug would immediately allow the attacker access to *all* private user data. Of course this solution is not acceptable: Asbestos aims to remedy exactly this kind of problem caused by overly privileged server components.

Another potential solution could be to fork a separate, dedicated process for each user being serviced by the system. Although this solution would meet our isolation requirements, it is obvious that it poses a resource challenge: a service may need to serve tens or even hundreds of thousands of users simultaneously.

```
1    while (1) {
2    event = get_next_event();
3    user = lookup_user(event);
4    if (not seen(user))
5          user.state = create_state();
6    process_event(event, user);
7    }
```

Figure 3.4: Event loop for a typical event-driven server application

The resource overhead (memory consumption, kernel data structures, context switching, etc.) of maintaining a separate server process for such large numbers of users renders this solution inadequate.

In order to avoid the problem of server process over-contamination in an efficient manner Asbestos provides the *event process* (EP) abstraction. Event processes are limited, fast forks of a process that have their own labels and address space.

Event processes were inspired by a simple observation of how many event-driven servers [PDZ99, BCZ03, Kro04] operate by using a main event dispatch loop, shown in Figure 3.4.

In this event-driven server model, user state is stored in data structures and gets initialized/recalled whenever a new/returning user requests service. Based on this observation, event processes implement a similar event loop that forks memory and label state for each new user and recalls forked state for returning users. Forks are stored by the kernel in efficient data structures (memory "diffs" and "copy-on-write" labels). When a returning user requests service, the kernel applies the user's memory "diff" to a base process's memory state and switches the server process labels to those that correspond to the user in question.

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

Figure 3.5: Typical event loop of an Asbestos server application using event processes.

Figure 3.5 presents the same server software architecture implemented with Asbestos event processes. Using the **ep_checkpoint** system call on line 1 the process enables event process and enters an event loop, waiting for the next message that will fork a new EP. If the new event is for an existing user, Asbestos restores the event process corresponding to that user (applies memory "diff" to the base process page tables and restores the appropriate labels). If a new user requests service, then the state variable will not have been restored once the loops is entered (line 2) and the subsequent code will initialize the state data structure and the communication port for the new user (lines 3 and 4). Each EP runs until it relinquishes control with the **ep_yield** (line 7) that completes the EP-loop iteration.

Using event processes to develop large server applications, we observed good performance, scalability up to 150,000 of simultaneous users, and efficient resource utilization on the order of a few pages of memory per user EP.

3.4 Asbestos Persistence

Most useful Web services use persistent data that must persist even if the server reboots and all volatile storage is cleared.

A persistent store must uphold the system's information-flow invariants, even across reboots. For example, if a contaminated process writes contaminated data to the hard disk and then later, another process reads the file, the reading process must also become contaminated. Furthermore, the data store must also preserve *privilege*, lest it be impossible for applications to extract labeled data after a reboot. Systems such as EROS [SSF99] and HiStar [ZBK06] avoid this problem by introducing a single-level store: reboot returns to a checkpointed system state, and a process's handles and capabilities are stored along with its virtual memory. However, conventional file systems with non-persistent memory are more familiar to most programmers and may simplify the process of recovering from application crashes without losing associated tag state.

3.4.1 File System Semantics

Each file in the Asbestos file system has a *contamination* label f_{C} and a *clearance* label $\mathbf{V}_f$. These are analogous to the Asbestos send and clearance process labels. Like a process's label, a file's contamination label represents the contamination of the file's data. The file system contaminates any process that reads from the file f with its label f_{C}. Similarly, a file's clearance label is like a process's clearance label; a process P with send label $\mathbf{T}_P$ may only write to a file f if $\mathbf{T}_P \sqsubseteq \mathbf{V}_f$.

The file system label rules are intuitively similar to the process label rules. If a file contains contaminated data, a process that reads it should collect the contamination. Furthermore, a writer should only be allowed to write to a file

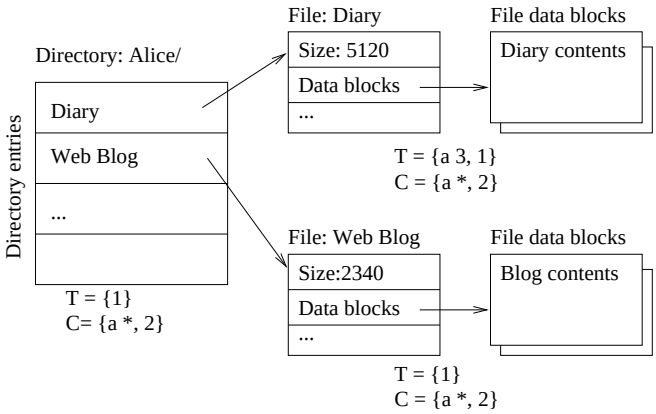

Figure 3.6: File and directory labels. User Alice owns a publicly readable directory *Alice*, a publicly readable file *Web Blog*, and a private file *Diary*. Only processes with privilege $\{a, *\}$ may modify her files.

if the file's clearance label allows it. Clearance labels let users control which processes may modify a file. Directories have labels and clearances exactly like regular files. For example, in Figure 3.4.1, user Alice owns the tag a and creates a file *diary* with clearance label $\mathbf{V}_{diary} = \{a\star, \mathbf{1}\}$, then the only processes that may modify *diary* are uncontaminated processes to which Alice grants the privilege $\{a\star\}$. Figure 3.7 summarizes the label rules for file system operations.

Unlike process labels, file labels are immutable. Files may not be dynamically contaminated or granted privilege, and a file meant to hold a secret must be tagged appropriately when it is created. The immutable label and clearance are supplied at creation time; the file system ensures that the new file is at least as contaminated as the creating process ($\mathbf{T}_P \sqsubseteq f_{\mathrm{C}}$), maintaining the information-flow rules, and that the clearance is no more tagged than the label ($\mathbf{V}_f \sqsubseteq f_{\mathrm{C}}$).

3.4.2 File System Pickles

Asbestos uses a flexible technique for preserving privileges called *pickling*. Using pickles Asbestos provides two persistent storage services, a file system and a

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

Figure 3.7: File operations on file f in directory d by process P.

shared database. The labeled file system enables the system to store user data without the risk of leaking them to unauthorized recipients. This effectively means that mutually untrusted users and programs are able to safely read and write the file system without risking privacy leaks. The Asbestos file system uses *pickle files* to serialize tags.

A *pickle file*, or *pickle*, is a serialized tag represented as a file in the file system. A process with privilege for a tag may preserve that privilege by creating a pickle. Later on, another process may *unpickle* the pickle, thus acquiring the privilege that was preserved in the pickle.

To create a pickle of tag t, process P sends a request to the file system containing t and the maximum privilege (i.e., smallest level) that the file system should grant as the result of unpickling the pickle. Since a pickle is also a file, P also specifies its pathname, label and clearance.

To acquire the stored privilege in the pickle, process Q (where Q may be the process equivalent to P after one or more reboots) issues an unpickle request to the file system.

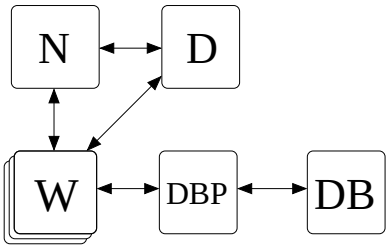

Figure 3.8: The main components of the Asbestos Web server and their interactions.

3.5 Developing for Asbestos

Asbestos labels have been used to develop a privilege separated Web server [EKV05] inspired by the OK Web server [Kro04]. The Asbestos Web server (AWS) labels user data as well as user network connections in order to ensure data isolation and avoid information leakage—e.g. leaks by bug exploits in the CGI scripts or any other untrusted application component. The main AWS components—as illustrated in Figure 3.8—include:

- A trusted *network daemon* (N), a system process with permission to access the network card. N labels network connections so that contaminated data can not leak to connections without adequate clearance.

- A trusted *demux* process (D) that accepts connections and redirects incoming requests to the appropriate worker process. When a new user connects to the service through N, the connection is forwarded by N to D and D looks up the tag identifying the incoming user—possibly performing a username/password authentication step. 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 contaminates incoming data as Alice-confidential and allows Alice-confidential data to escape on that connection. As a result, the demux, like the network

daemon, is trusted.

- A set of *worker EPs* (W), each handling requests for a particular user and contaminated accordingly. Alice's worker has clearance to access only Alice's data from the database and is considered untrusted (e.g. buggy CGI script).

- A *database proxy* process (DBP), responsible for marshalling all data coming in and out of the database: it sanitizes requests, checks privilege for writes, and contaminates all outgoing data appropriately—using the owner's tag. Since DBP handles all user data it is considered a trusted component.

- A simple *database* (DB) back-end that stores user data.

The AWS uses DIFC to protect the secrecy of user data even when (the untrusted) parts of the application have exploitable bugs. For instance, the network connection used by Alice is marked with Alice's contamination $\{a\,\mathbf{3}\}$ and is given clearance to receive Alice's contamination (i.e. $\mathbf{C}_{netd}(a) = \mathbf{3}$). With this label configuration, Alice's data may be exported over the network but Alice's network connection does not have clearance to export Bob's data. Consequently even if Alice compromises some untrusted component of the application (e.g. a CGI script) and gains access to Bob's secrets, she can not export that information due to information flow restrictions imposed by Asbestos labels.

The AWS was also used to implement Muenster [BEK07], a job searching Web service resembling some of the popular services such as Monster.com or HotJobs.com. Using Asbestos labels, Muenster is able to provide a novel application development paradigm that is inspired by the recent success of Wikipedia [Wik] and other similar collaborative efforts: Untrusted developers (essentially users of the Web service) are able to upload their own code (e.g. in the form of binaries)

to customize and augment the functionality of the Web service. In the context of a job-search Web site, both job seekers and employers are able to upload binaries for executing custom searching on the database as well as custom declassifiers that make decisions about their private information based on each user's criteria.

The Muenster application demonstrates that using the information flow control tools provided by Asbestos labels we were able to successfully implement a service that allows mutually untrusted users to upload untrusted custom code to the Web server without breaking system security guarantees and application policy. More specifically, Muenster's features prevent a user's uploaded (untrusted) module to export data about another user without her permission. Furthermore, write access to the database is also protected by Asbestos labels to ensure that appropriate privilege is required to write to each row. Finally, uploaded code is able to create private compartments implementing custom private policies. Using Asbestos labels, Muenster is successful in providing a discreet job searching and posting service, keeping user data private despite untrusted extensions running on the server.

3.6 Summary

The Asbestos operating system makes nondiscretionary access control mechanisms available to unprivileged users, giving them fine-grained, end-to-end control over the dissemination of information. Privilege management is performed in a decentralized fashion: creating new isolation domains—or compartments—does not require and special type of system privilege. By decentralizing privilege management, Asbestos allows developers to use the Asbestos labeling mechanism to design, implement a wide range of security policies, including data isolation and confidentiality policies necessary to improve the security of networked server

applications.

Fine-grained privilege management allows Asbestos developers to significantly reduce the amount of privilege each application module holds. However, most applications need to be able to restore their privilege after an application and/or system restart. Asbestos's labeled file system supports this functionality by providing means to serialize and store labels persistently, therefore preserving privilege.

Using Asbestos we developed a privilege-sperated Web server, able to provide data isolation for the Web application users, even in the presence of bugs. Although the security policy is defined by the application developer, it is strictly enforced by the operating system kernel, therefore providing strong security guarantees even when the CGI scripts ran on the server have been contributed by untrusted third-party developers (e.g. in the Muenster application).

The data isolation features provided by the Asbestos Web server and the Muenster application demonstrate that developers can significantly improve the security of networked server applications by using decentralized information flow control. However, it is necessary though for developers to design and express the desired application security policy in Asbestos label terms.

CHAPTER 4

DIFC Policy Management Challenges

In this section, we use a concrete example derived from the Asbestos Web server (AWS) [VEK07] architecture to further motivate the policy management and debugging challenges addressed in this work.

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. An application written for a DIFC system is as secure as the application policy that it implements (and, of course, the security kernel that enforces the policy). 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 4.1 presents a more elaborate version of the policy of the Asbestos Web server of Figure 3.8. 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

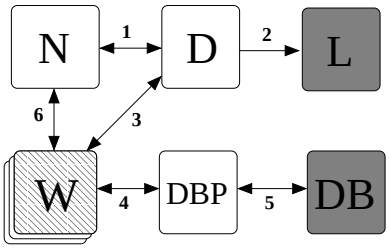

Figure 4.1: A representation of the explicit communication requirements in an AWS-like policy. Arrows represent expected communication patterns; single arrows denote one-way communication. In the absence of arrows, a per-component default communication rule applies: white components (N, D, DBP) are able 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.

not escape to some other user, even in the presence of bugs in worker processes. We examine how each of the application processes should (not) communicate with the rest of the system, and define the application policy accordingly.

The network daemon N and the demux D are considered trusted components. [1] After D has identified and authenticated the incoming user (e.g. Alice), and has notified N, it informs the system logger L of the new connection. For safety 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, only for performance reasons it is using the event process mechanism. 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

[1] We assume that the demux as well as all other trusted components such as N are part of the application's trusted computing base (TCB).

DBP. For our confidentiality policy to be enforced, we do not need to restrict W's default ability to receive information freely: information that can reach W—without having been blocked by the security policy on the sender side—does not pose a confidentiality threat, as long as W can not leak it.[2] Therefore, 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.

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, interposing on all queries and data to and from 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 (i.e. data carrying Bob's contamination), but not Alice-confidential data.

The arrows in Figure 4.1 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

[2]Notice that "receive only" behavior does not mean that W can freely receive any type of contamination: it means that it has the default Asbestos receiving behavior, with no additional receiving restrictions.

communication behavior does not need to be limited by default in this high-level policy. [3] (These processes may, and do, limit their own communication behavior to enforce finer-grained policies—for instance, N may use Asbestos port labels to restrict access to certain ports, requiring privilege to send information over to them.) This is represented in Figure 4.1 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 4.1 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 4.1 can be translated into decentralized information flow labels. For example, labels can prevent L from sending information to other processes: L

[3] Additionally, services such as the network daemon may often require that its communication with the rest of the system is unrestricted (so that any process can use its services).

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 4.1 is not trivial: one simple label implementation requires initializing the corresponding processes requires 20 Asbestos label operations in order to initialize the relevant process labels for a single user, as shown on Table 4.1. This complexity of policy management tasks is one important aspect that significantly complicates the DIFC programming model for developers. The issue is that a communication pattern like Figure 4.1 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 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.

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	$\{db\,\mathbf{3}, db'\star, \mathbf{1}\}$	$\{db\,\mathbf{3}, db'\,\mathbf{0}, \mathbf{2}\}$

Table 4.1: Asbestos labels implementing the policy of Figure 4.1.

These interactions between components and their label implementation are not easy to get right without debugging, and to make matters worse, debugging an incorrect label configuration is difficult itself. For instance, let us assume that DBP, in an attempt to remove unnecessary privilege from its labels, mistakenly drops privilege to send information to the database. The next time it attempts to forward data to W a label error will be generated because of the missing privilege. Any debugging data in this case would explicitly or implicitly convey information about DBP and W (e.g. DBP's lack of database privilege, or W's requirement for privilege) and their release creates an information flow not subjected to the system's IFC rules. Although such a label error in this scenario is the result of a programming bug, its details—including its very existence—will be concealed in order to avoid information leaks, and the developer will have extremely limited knowledge about why the application is not functioning.

4.1 Summary

Although system-based DIFC can simultaneously achieve high performance and effective isolation [VEK07], it offers a challenging programming model, as demonstrated by the example derived from our experience developing the Asbestos Web server. There are fundamental problems with the model even aside from general

developer unfamiliarity. Using Asbestos labels to design and implement security policies is a difficult and error prone process because the label interface is from human-friendly. Furthermore, fine-grained policy specifications are spread over several application pieces and require multiple label operations, making it hard to inspect and reason about policies. Consequently, policy implementation bugs are very common, even if the developer is familiar with the labeling mechanism of the DIFC system.

From the DIFC system's (i.e. the kernel's) point of view, policy bugs and common programming errors are usually indistinguishable from policy exploit attempts; the system cannot expose developers to information about these errors, complicating debugging. Exposing debugging information in an uncontrolled fashion may—explicitly or implicitly—violate the security policy.

In order to improve the DIFC programming model, it is necessary to address these policy management and debugging issues. First, developers need a high-level, human-friendly, easy-to-use way to specify application policies. Additionally, it is important for developers to be able to debug policy errors using debugging mechanisms that do not violate the security guarantees of the system.

CHAPTER 5

Policy Description Language

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—even in the case of relatively simple policies, like the one presented in Figure 4.1. Practical reasons for this include:

- Policies are expressed by developers, directly in code
- Policies are expressed in terms of Asbestos label, which may be more difficult to work with than process communication relationships
- Policy implementation is spread across multiple code locations, which complicates reading, sharing and reasoning about the policy
- Policy definition is error-prone and debugging policies implemented at the label level is challenging

Despite the fact that Asbestos labels are a very expressive and effective primitive for policy implementation, we believe that developers understand and reason about policies by describing the communication behavior of each component with respect to the rest of the system. Consequently, a developer would find it more natural to express the desired policy in terms of communication patterns that are

(not) allowed within the application. 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* [EK08] is capable of expressing application policies in terms of communication constraints.

Although communication rules are a convenient and appropriate way to describe a security policy, DIFC labels are a much better implementation technique for security policies. This is because a process's labels form a concise and complete description of its communication constraints that can apply even across shared resources like the file system or a shared database. For example, a policy may require that the contents of a secret file may not escape to the network. In a DIFC system a process that may send information to the network daemon is able to do so as long as it has not accessed the secret file. Once the file is accessed, the process "automatically" loses the ability to send data over the network. Using information flow rules—rather than strict, explicit communication restrictions—is much more effective, since it provides fine-grained control over the policy and more flexibility (e.g. the ability to access system services such as the network daemon) than a strict communication rule that might need to forbid network access to the process altogether.

To benefit from the advantages of DIFC, we must translate the policy described by developers in terms of communication restrictions, into information flow rules. To achieve this, we compile the communication constraints down into the appropriate labels that implement the DIFC policy.

5.1 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 resemble the relationships diagrammed in Figure 4.1, but compiles to labels like those in Table 4.1. Figure 5.1 summarizes the policy of Figure 4.1 in terms of communication relationships between components—a high-level description we believe is closer to the way developers can understand policies (as opposed to Asbestos label expressions). Our policy language specifies these relationships in one compact and intentionally simple definition, rather than scattering necessary operations throughout application code. Notice that in order for the high-level

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 5.1: Our example policy expressed as communication restrictions.

policy description (corresponding to the communication restrictions of Figure 5.1) to reflect the AWS application policy of Figure 4.1 realistically, we need to ensure that it can refer to and be instantiated with real, pre-existing tags utilized by the system. For instance, the worker W may correspond to some user, Alice, who has

state stored on the file system (or records in the database) that has been marked with one or more tags. The goal of the policy is to allow the export of Alice's information to her worker, but not any other secret information.

After communication patterns between application components have been compiled down to equivalent Asbestos label setups they can be used to instantiate process labels and launch the application. Our policy language provides additional constructs able to describe important runtime properties of the application, used to instantiate policies and launch applications.

5.2 Implementation

In order to provide a policy language capable of simplifying policy management, we had to address the following goals and challenges:

1. Design a language capable of describing compartments and communication patterns between them. Ensure that the language is powerful enough to describe interesting policies, yet simple and readable.

2. Identify the fundamental label translation rules that will be used to map higher level policy expressions to equivalent DIFC label arrangements.

3. Define language constructs able to capture important runtime properties of applications (e.g. event processes) and integration with the system (e.g. interface with pre-existing compartments).

4. Provide developers with a tool capable of translating the high level description to equivalent Asbestos label configurations.

5. Provide a component capable of instantiating policy labels and launching the application.

Compartments The first task of the policy language is to represent the main principals involved in the policy, each corresponding to an Asbestos *compartment*, and specify the *communication rules* that constrain communication between those compartments.

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. In the language, each compartment has a unique name and is defined by the *comp* construct. Figure 5.2 presents a simplified version of our Asbestos Web server policy from Figure 4.1 written in the policy description language; lines 1–17 define the system's 6 compartments.

```
 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 5.2: A simplified implementation of our example policy in our policy language. The parser will use this description to produce the labels of Figure 4.1.

Communication Rules 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 Y. The four possibilities for each pair are no communication, bidirectional communication, and (less frequently) unidirectional communication in either direction. As presented in Table 5.1, we write these possibilities using four basic communication operators: "!", "<>", "<" and ">". The last rule stated for a given pair of com-

Operator	Example	Meaning
!	X ! Y	**Isolation**: X can neither send nor receive from Y
<>	X <> Y	**Unrestricted (bidirectional communication)**: X can both send and receive to/from Y
<	$X < Y$	**Receive-only**: X can only receive messages from Y (cannot send to Y)
>	$X > Y$	**Send-only**: X can only send messages to Y (cannot receive from Y)

Table 5.1: The four basic communication operators used between compartments (explicit rules) or as compartment defaults.

partments 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 one of the communication operators of Table 5.1 which signifies the compartment's default communication behavior (or simply the compartment's *default*). The compartment's default governs the compartment's communication with all compartments and/or processes for which there is no explicit communication rule. The meaning of the four possible compartment default values is analogous to that presented on Table 5.1: bidirectional (<>), send-only (>), receive-only (<), or isolated (!) rules may be applied to non-explicit communication relations. 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 5.2

implies that $W \mathbin{!} DB$ (the intersection of W's default $W < DB$ and DB's default $DB \mathbin{!} W$). Note that the compartment default constrains a compartment's communication with entities not explicitly mentioned in the policy as well as entities that are mentioned explicitly in the policy definition.

Label calculation The second 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. A closer look at Figure 4.1 and Table 4.1 reveals the basics of mapping rules and compartment default behaviors to labels. In order to control a compartments communication abilities, we need at most two tags: one for controlling sending and one for controlling reception of information. For a compartment X we write these tags as x (referred to as the "send tag", used to restrict the compartment's sending ability) and x' (referred to as the "receive tag", used to implement receiving restrictions). The send and/or receive tag may not be used (i.e. appear in any label) if the compartment has no sending and/or receiving restrictions. Two tags per compartment are sufficient to implement any pairwise communication policy representable with Asbestos labels at all. 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 [VEK07].

Processes in compartment X have their label components for x and x' defined by X's default communication pattern. Table 5.2 presents how we can use X's compartment tags to implement the compartments default communication behavior and—in essence—represent the compartment in Asbestos label terms. The unrestricted default communication rule, <>, leaves a process's labels unchanged from the system-wide defaults, thus posing no restrictions (with respect

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

Table 5.2: The label implementations for X's four possible default communication behaviors.

to x and x') to X's sending and receiving ability. We implement the ! default rule by using both x and x': $\{x\,\mathbf{3}\}$ 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. Notice that the $\{x\,\mathbf{3}\}$ contamination implies the corresponding $\{x\,\mathbf{3}\}$ clearance, since Asbestos requires that for any process X, $\mathbf{T}_X \sqsubseteq \mathbf{C}_X$ at all times (this practically means that process should always be able to send a message to itself).

The "receive-only" (<) and "send-only" (>) defaults are implemented using only x and x' respectively: in the receive-only case (third line of Table 5.2), we do not make use the x'—since we do not have any receiving restrictions—and use x to restrict X's sending ability through the presence of $\{x\,\mathbf{3}\}$ in $\mathbf{T}_X$ (and the necessary corresponding $\{x\,\mathbf{3}\}$ clearance). Given that the clearance label default level is $\mathbf{2}$, this contamination prevents X from sending messages freely to processes that do not have explicit $\{x\,\mathbf{3}\}$ clearance.

Similarly, for the send-only default (fourth line of Table 5.2) we do not make use of x (no sending restrictions) and use x' to restrict X's receiving ability lowering X clearance with respect to x' below the default contamination of $\mathbf{1}$ ($\mathbf{C}_X(x') = \mathbf{0}$). Given that the tracking label default level is $\mathbf{1}$, lowering X's clearance to $\mathbf{C}_X(x') = \mathbf{0}$ prevents X from receiving messages by default (since a sender's default contamination with respect to x' is greater than $\mathbf{0}$) .

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

Table 5.3: Mapping of rules to labels when X's default is "<>": as the first line of Table 5.2 indicates, X's default does not imply any changes to X's labels. Tags x and x' are used to implement only the explicit communication rules between X and Y.

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

Table 5.4: Mapping of rules to labels when X's default is "!": as indicated by the second line of Table 5.2, x and x' are first used to implement X default (underlined changes to X labels only). Given X's fully restrictive default, we use x and x' to implement the explicit communication rules between X and Y. In practice this amounts to giving Y the necessary privilege with respect to x and/or x' in order to override X's communication restrictions.

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

Table 5.5: Mapping of rules to labels when X's default is "<": as indicated by the third line of Table 5.2, x and x' are first used to implement X default (underlined changes to X labels). We use x and x' to also implement the explicit rules between X and Y—given the changes due to X's "receive-only" default.

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

Table 5.6: Mapping of rules to labels when X's default is ">": as indicated by the fourth line of Table 5.2, x and x' are first used to implement X default (underlined changes to X labels). x and x' implement the explicit rules between X and Y—given X's "send-only" default.

A communication rule involving two compartments affects the labels of processes in either compartment. Tables 5.3, 5.4, 5.5, and 5.6 demonstrate partial labels for the sixteen possible combinations of rules and defaults. In essence, the Tables demonstrate the label components that need to appear to each process's labels in order for each combination of default and explicit rules to be implemented, when X is the left-hand-side operand. 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 Tables 5.3, 5.4, 5.5 and 5.6 to set up compartment labels. First, the compartment tags implement the defaults of all compartments, based on Table 5.2's translations. Then, for each rule, we choose the table 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 Table 5.5. If the rule operator violates the other compartment's default, we interpret the rule using its tags as well. In our previous example, if rule $X > Y$ violated Y's default, we would also consider the equivalent "reverse rule" ($Y < X$) and use y and y' in the produced label setup. Otherwise, only x and x' are used to implement the (original) rule and we do not need to consider the reverse rule.

Translation example Let us reconsider the AWS example of Figure 4.1, as expressed using our policy language on Figure 4.1.

On line 21, the rule $W <> D$ ensures that workers can have a two way com-

munication with the demux, despite the worker's "receive-only" default (declared on line 16). The translation of W's default—which was performed at an earlier step—will have already setup W's labels[1] with respect to w and w' according to the underlined components of Table 5.5: $\mathbf{T}_W(w) = \mathbf{3}$, $\mathbf{T}_W(w') = \mathbf{1}$ (by default), $\mathbf{C}_W(w) = \mathbf{3}$ and $\mathbf{C}_W(w') = \mathbf{2}$ (by default), therefore implementing W's "receive-only" default behavior. As with any rule, we first examine the "forward direction" using the left-hand-side operand's tags—i.e. check how the rule needs to implemented using w and w', given W's default. Since W's default is $<$ and the rule operator is $<>$, we will use the first row of Table 5.5 to translate this rule, substituting x and x' with w and w'. The changes necessary to implement the rule—as indicated by the first row of Table 5.5—also involve the right-hand-side operand's labels. D's labels need to be set to:

- $\mathbf{T}_D(w) = \star$, giving D privilege with respect to W's compartment, so that D can not get contaminated with $\{w\,\mathbf{3}\}$ when receiving data from W
- $\mathbf{T}_D(w') = \mathbf{1}$, by default
- $\mathbf{C}_D(w) = \mathbf{3}$, giving D clearance with respect to W's compartment so that D able to receive data contaminated with $\{w\,\mathbf{3}\}$
- $\mathbf{C}_D(w') = \mathbf{2}$, by default

Translating the "forward direction" is not always adequate, since it only ensures that we override the left-hand-side operand's default (if necessary). It is important to ensure that the communication rule in question does not oppose the right-hand-side operand's default. In this example, since the rule operator ($<>$)

[1]Notice that W and D are compartments that may contain multiple processes and files. Although we refer to "W's labels" for brevity, technically we refer to changes necessary to the labels of any process belonging to the W compartment.

does not contradict the right-hand-side operand's default (D's default, $<>$), we do not need to translate the rule in the "reverse" direction (i.e. $D <> W$, using d and d' for the translation). These necessary changes to the labels of W and D are also visible on Table 4.1.

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 automatically. Using the rules of Tables 5.3, 5.4, 5.5 and 5.6 the parser is able to translate the policy description of Figure 5.2 to the label setups of Table 4.1.

5.2.1 Launcher

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. Furthermore, typical Asbestos applications include a "wrapper" or "launcher" component, responsible for spawning application processes and setting up their labels according to application policy. Having calculated an application's policy labels using our policy parser, we are able to take a step further and remove the need for such specialized wrappers. Several language features support this *launcher* functionality, including definitions of application binaries associated with each compartment; the launcher program can then start these binaries with the correct labels. Most application features require additional support. For example, server applications can create and destroy user event processes at run time, as users join and leave the system. The policy language supports this by allowing the user to describe EP properties and dynamically parameterize user compartments at run time.

An application launcher incorporated into the language parser instantiates

these labels at run time using additional language constructs. We have extended the language so as to make it able to describe important application run-time properties, necessary for application launching.

Launching an 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.

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, and any other compartments to which the processes will belong ("belongs"). For instance, the following fragment

from the AWS policy presented in Appendix A describes the demux (D), stating the executable the launcher must spawn ("/okws-demux"), the arguments to be passed to it ("sh sql login edit view"), and that the process belongs to the "DEMUX" compartment and therefore its labels should be set up accordingly.

```
exec demux {
    bin /okws-demux sh sql login edit view
    belongs DEMUX
    ...
}
```

If an executable belongs in multiple compartments, its process 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.

External compartments Many useful application policies require the application components to interact with the rest of the system—such as compartments that are external to the application compartment structure. For example, the database may contaminate the data it stores with a tag that is stored persistently on disk (using the pickle() mechanism) so that it can be retrieved after system restart. Similarly, the AWS application of Figure 4.1 may not run its own network daemon, but instead use an external system service to access the network. Moreover, the network daemon process is a component external to the

application: the developer is not responsible for launching the process and does not have control over it, yet it participates in the application policy. In both cases, the AWS application policy needs to know of and interact with external compartments and process that have not been created and are not managed by the application itself.

Our policy language provides developers with the ability to describe such interactions by supporting the notion of *external compartments* and *external executables*. The developer may implicitly declare a compartment E as external by specifying the (external) location of its send and receive tags (e and e') through the use of one of two mechanisms: environment variables or file system pickles. For instance, in Figure 5.2 the database compartment DB is initialized through a pair of pickle files storing the values used to initialize the compartment tags db and db'. Compartment N is also declared external (e.g. because the network daemon is an already running system service) and the launcher will initialize the compartment tags n and n' using the values of the environment variables NET_S and NET_R.

Similarly, executables can also be declared external by stating that the executable belongs to an external compartment—using the appropriate *belongs* statement in the relevant *exec* block. Notice that external compartments executables may belong to can be declared "in place"—inside the *exec* block. For instance, one could make an executable external by adding one of the following lines to an *exec* block:

```
belongs    (env EXT_SEND EXT_RECEIVE)
```

or

```
belongs    (unpickle /path/to/pickle1 /path/to/pickle2)
```

If an external compartment participates in a policy rule that requires its labels to be modified, then the launcher will generate a relevant warning when processing the labels. In the case of external executables, the launcher will attempt to send to one of the ports of the external executable a message carrying labels that would have the desired effect on the recipient's labels. Note that this is possible because almost all potential modifications to the external executable's labels can be performed by any process holding privilege with respect to the relevant compartment handles (e.g. the launcher) by making appropriate use of discretionary labels attached to messages sent to a port the external executable is receiving from. The only case that can not be handled using message discretionary labels is when the external process's clearance label needs to be lowered to $\mathbf{1}$ with respect to a compartment's send tag (e.g. Table 5.6, rows 2 and 3). For example, a policy may involve a local compartment X and an external executable ex:

```
comp X {
     default >
}
exec ex {
     belongs    (unpickle /path/to/pickle1 /path/to/pickle2)
}

X < ex
```

According to Table 5.6, rows 3, it is necessary to lower ex's clearance labelin order to implement the $<$ ("receive only") operator between X and ex: having $\{x\,\mathbf{1}\}$ in its clearance label will prevent ex from receiving messages from X. In these particular cases the label manipulation would have to be performed by a privileged third-party process. Alternatively, a userspace protocol could be

implemented between the launcher and the external process—which is assumed to be cooperating.

The detailed implementation of the AWS in Appendix A makes use of external executables.

Event Processes EPs differ significantly from all other application components because they cannot be launched during application startup. Instead EPs are created dynamically, at unpredictable points in time. The way our policy language models EPs is by using *dynexec* blocks, which are based on the observation that each server process EP essentially implements the same policy as the other EPs of that process, only for a different user. In that sense, EPs resemble some sort of policy "template", parameterized and applied to a new EP instance for each user.

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 (the EP base process) 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 the EP belongs to. Each new EP is hosted in a separate, dynamically created compartment.

A "source" property identifies the executable that is responsible for spawning new EPs by sending the relevant messages to the base process. For example, in the AWS the demux D is responsible for instantiating new event processes as users log in. To do so, it sends the necessary messages to the base worker process W, and therefore, in our language, the demux is the "source" of the worker EPs.

The declarations inside the *dynexec* block are initialized at EP creation time by the process responsible for creating the EP, i.e. the EP source. The EP source

```
dynexec USER {
    source demux
    belongs (env USER_S USER_R default <)
    file user-tmpl
    port WORKER_PORT {
        type restricted
    }
    env WRPORT=port:WORKER_PORT
    ...
}
```

Figure 5.3: Fragment of the AWS policy description presenting the worker event process definition, using a *dynexec* block.

process executes a special block of code generated by the policy launcher; that generated code includes all the operations required to set up the new EP: instantiate the "policy template" for the new user by manipulating EP labels accordingly, create and transfer ports to bootstrap communication, and set EP environment variables. By setting EP environment variables the EP-creation code can parameterize each EP. For instance, when a new EP worker is created for an AWS user that just logged in, the EP-creation code will set the new EP's environment variables (e.g. port values that will bootstrap communication) to values corresponding that particular user. Our implementation of the launcher is able to generate both C and Python EP-creation code.

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. Therefore, before calling the code, D ensures that USER_S and USER_R have been initialized appropriately to reflect the current user.

Figure 5.2.1 presents a fragment of the AWS policy (presented in Appendix A) which is contained in the "worker" *exec* declaration block and defines the "USER" event process. The block starts by stating that the "demux" executable is the

source of each newly created EP. A new worker EP 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 is ran by D upon creation of a new EP and uses the $USER\_S$ and $USER\_R$ environment variables to initialize the new EP compartment tags. Therefore, before calling the code, D ensures that $USER\_S$ and $USER\_R$ have been initialized appropriately to reflect the current user's identity. The values of these environment variables are used to implement the EP compartment's "receive-only" default labels—according to Table 5.2. variables $USER\_S$ and $USER\_R$ that will be used to implement a "receive-only" default. The code generated by the launcher (a function with a predefined name and arguments) will be stored in the file "user-tmpl.c" or "user-tmpl.py"—depending on whether we instructed the launcher to generate C or Python code—and will be included in the EP-source process. In this example, the demux (EP source) will include the function stored in the file "user-tmpl.c" as part of its source code and call it whenever a new user EP is instantiated. For each EP, a new "restricted" port "$WORKER\_PORT$" will be created and an environment variable $WRPORT$ will be set to serve bootstrapping needs.

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 launcher uses port declarations to create communication endpoints between

application components and bootstrap communication.

The application bootstrapping process almost always requires that each application component needs to know of at least one port value in order to bootstrap its communication with the rest of the application and/or system. For instance, in the AWS policy fragment of Figure 5.2.1, each new EP needs to know the value of the port created for it ("*WRPORT*"). Additionally, it will need to know of additional port values, such as a port to communicate with the demux (Appendix A).

The *env* and *env** properties declare environment variables initialized using port and tag names, therefore exporting the port/tag values out of the launcher to the relevant processes and/or EPs. The *env** property also grants the receiving process privilege with respect to the named ports. Figure 5.2.1 shows that, when a new EP is initialized and the WORKER_PORT is initialized for it, the environment variable "WRPORT" is set for the new EP with the value of the newly created port. By using the *env** in this case, the EP would also be granted privilege with respect to *WORKER_PORT*. Similar to ports, environment variables can also be used in order to pass the values of regular Asbestos tags to processes and EPs if the application protocol requires it.

After all ports have been created and their ownership is passed to their respective owners, each application component is informed of significant port values through the relevant environment variables—whose values were properly set by the launcher—and therefore knows all ports it needs to listen or send to.

Process	Tracking label	Clearance label
A	$\{a\,\mathbf{3}, \mathbf{1}\}$	$\{a\,\mathbf{3}, \mathbf{2}\}$
B	$\{a'\star, \mathbf{1}\}$	$\{a\,\mathbf{3}, \mathbf{2}\}$
C	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$
A	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$
B	$\{\mathbf{1}\}$	$\{c'\star, \mathbf{2}\}$
C	$\{c'\star, \mathbf{1}\}$	$\{c'\,\mathbf{0}, \mathbf{2}\}$
A	$\{c'\,\mathbf{2}, \mathbf{1}\}$	$\{\mathbf{2}\}$
B	$\{c'\star, \mathbf{1}\}$	$\{\mathbf{2}\}$
C	$\{c'\star, \mathbf{1}\}$	$\{c'\,\mathbf{1}, \mathbf{2}\}$

Table 5.7: Three label setups implementing a circular communication pattern by granting extra privilege where necessary: A can communicate with B, B can communicate with C, but A is not allowed to communicate (directly) with C. In all three cases (each of them essentially corresponding to a different compartment default for A or C) B needs to hold extra privilege to avoid contamination and maintain the ability to communicate with C even after it has received information from A.

5.3 Discussion

Our 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. But in the case where there is a restriction in the cycle, DIFC systems require the exercise of privilege to avoid the potential effects of transitive contamination: 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, granting privilege as required to implement a policy. This is necessary in DIFC systems in order to make the B "immune" to the contamination preventing A from communicating with C, and our parser implements this behavior. In this case the intermediate process B is acting as a declassifier between A and C—and therefore B is a privileged entity. Table 5.7 presents possible label setups

Process	Tracking label	Clearance label
A	$\{a\,\mathbf{3}, \mathbf{1}\}$	$\{a\,\mathbf{3}, \mathbf{2}\}$
B	$\{\mathbf{1}\}$	$\{a\,\mathbf{3}, \mathbf{2}\}$
C	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$
A	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$
B	$\{\mathbf{1}\}$	$\{c'\,\mathbf{0}, \mathbf{2}\}$
C	$\{c'\,\star, \mathbf{1}\}$	$\{c'\,\mathbf{0}, \mathbf{2}\}$
A	$\{c'\,\mathbf{2}, \mathbf{1}\}$	$\{\mathbf{2}\}$
B	$\{\mathbf{1}\}$	$\{\mathbf{2}\}$
C	$\{c'\,\star, \mathbf{1}\}$	$\{c'\,\mathbf{1}, \mathbf{2}\}$

Table 5.8: The three labels setups of Table 5.7 implemented without granting extra privilege. In all three cases B is able to communicate with C as long as it hasn't received any information from A. Once B receives A's messages, transitive contamination will prevent communication with C.

that implement our example. In all three cases, B is granted privilege necessary to ensure that it will not lose its ability to communicate with C, while always enforcing the requirement that A should not be able to send data to C directly.

If no additional privilege is granted in such communication patterns, processes are necessarily left susceptible to contamination with respect to the compartments in question. In the previous example this would mean that B would be able to communicate with C as long as it hasn't received any message from A. Once A sends a message to B, then B would become contaminated with the same type of privilege that prevents A from communicating with C. From an IFC point of view, this behavior not only is acceptable, but could also be considered desired.[2] In our policy language, $\ll$ and $\gg$ communication operators support this pattern. Table 5.8 presents possible label setups that implement our example without granting B special privilege.

Furthermore, our language attempts to simplify policy description, but does

[2]This prevents processes from leaking information through a "proxy"—like B in this example.

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 including parameterized event process and interactions with existing compartments and processes that are external to the application.

5.4 Experiences and Evaluation

In order to evaluate our policy language, we want to demonstrate how it could aid the development of DIFC applications, by simplifying policy definition and implementation. We use examples previously presented by DIFC systems such as Asbestos, HiStar, and Jif.

Our policy language 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. This is primarily because during startup Python attempts to load a large number of libraries that have not been ported to Asbestos. Additionally, our untuned file-system implementation further increases Python initialization. Since the running time of the parser is minimal once the Python

interpreter has finished initializing, we could solve this issue by addressing the reasons that delay Python initialization.

We have used our policy language to describe several interesting DIFC policies, and were able to produce equivalent Asbestos label setups. 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 4.1 is based on our implementation of the full Asbestos Web server policy in our policy language, presented in Appendix A. This was a challenging exercise since the policy uses all

```
1   comp W { default < }
2   exec worker1 worker2 {
3       bin /okws-login login
4       bin /okws-view view
5       belongs W
6       port WP { type open }
7       port WV { type open }
8       dynexec USER {
9           source demux
11          default !
12          belongs (env USER_S USER_R default <)
13          file /ep-configs/user-tmpl
14          port WORKER_PORT { type open }
15          port UG {
16              type restricted
17              owner parent
18              belongs (env USER_S USER_R default <)
19          }
20          env* NETPORT=port:NETROOT
21          env  WRPORT=port:WORKER_PORT
22          env* DPORT=port:DEMUX_USERP # port to D
23          env* UG=port:UG
24      }
25      env  SELFPORT=port:WP
26      env* DBP=port:DBP # port to database proxy
27      env  MYVERIFY=port:WV
28      env  DEMUXPORT=port:DEMUX_USERP
29  }
```

Figure 5.4: Part of the AWS policy description showing the declaration of the compartment and executables for two AWS workers. Both worker executables make use of EPs.

of Asbestos's features, including event processes, to provide both system-based information flow isolation and high performance. As described in Section 3.5, each of the AWS worker processes implements a different Web service by using a separate EP per user, in order to achieve isolation while avoiding over-contamination. In this example we make use of the *dynexec* directive to describe worker EPs. Our full implementation of the AWS policy also includes all executables and ports required for the application. Using our policy description in conjunction to our policy launcher, we were able to remove the need for the 511-line long specialized AWS launcher previously used for label initialization and process spawning. Also, we have identified at least 28 additional policy related operations in various places inside the AWS code that are not required if we use our language for policy management. Figure 5.4 presents the part of the AWS policy that declares the Web server worker processes which utilize EPs to handle user connections. The full policy description is presented in Appendix A.

Using its Unix compatibility layer, HiStar [ZBK06] 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 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 5.5). Just as *wrap* can protect processes other than ClamAV, so simple changes to Figure 5.5 can protect different executables; in a sense, Figure 5.5 is a generalization of *wrap*'s policy.

```
1   # We first declare the three compartments
2   comp USER { env USER_S USER_R default ! }
3   comp AV { default < }
4   comp PRINTER { default <> }
5
6   # Executable declassifying output to the tty
7   exec tty_printer {
8           belongs PRINTER
9           port PRINTER_PORT { type restricted }
10          env MYPORT=port:PRINTER_PORT
11          env AV_PORT=port:CLAMAV_PORT
12  }
13  # ClamAV process. Also spawns helper process
14  # belonging in same compartments
15  exec avscanner {
16          belongs AV
17          port MAIN_AV_PORT { type restricted }
18          env MYPORT=port:MAIN_AV_PORT
19          env PRINTER_PORT=port:PRINTER_PORT
20  }
21  # Process modeling private /tmp folder.
22  # Could be replaced by labeled FS
23  # This process also belongs to the
24  # externally initialized user compartment
25  exec private_tmp_file_server {
26          belongs AV USER
27          port TMP_PORT { type restricted }
28          env MYPORT=port:TMP_PORT
29          env AV_PORT=port:MAIN_AV_PORT
30  }
31  # Process modeling private user data.
32  # Could be replaced by labeled FS
33  exec user_data_server {
34          belongs USER
35          env AV_PORT=port:MAIN_AV_PORT
36  }
37
38  AV <> PRINTER
39  USER <> PRINTER
40  USER > AV
```

Figure 5.5: HiStar's ClamAV security policy implemented using our policy language.

HiStar also presents a VPN isolation example. This example assumes that a user is simultaneously connected to both the Internet and a virtual private net-

work (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 network is verified to be non-confidential. Figure 5.6 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.

```
 1   # We declare the five compartments
 2   comp VPN INTERNET INTERNET_IPSTACK { default < }
 3   comp VPN_CLIENT { default <> }
 4   comp NETD { default ! }
 5
 6   # Both the vpn browser and the vpn IP stack
 7   # belong to VPN
 8   exec browser_vpn ipstack_vpn { belongs VPN }
 9   exec browser_internet { belongs INTERNET }
10   exec ipstack_internet { belongs INTERNET_IPSTACK }
11   exec vpn_client { belongs VPN_CLIENT }
12   exec netd {  belongs NETD }
13
24   VPN_CLIENT <> VPN
25   VPN_CLIENT <> INTERNET_IPSTACK
26   INTERNET_IPSTACK <> INTERNET
27   INTERNET_IPSTACK <> NETD
```

Figure 5.6: A policy similar to HiStar's VPN isolation, implemented using our policy language.

Another interesting policy we were able to express was Jif's [ML00] medical study example. In this scenario a hospital (modeled on line 3 of Figure 5.7 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 5.7 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 5.7: A policy configuration implementing the security model of the Jif medical example. For brevity we omit executable declarations.

5.4.1 Discussion

Although our work and evaluation focussed primarily on Asbestos, our goal is to propose system management mechanisms able to improve the DIFC programming model. As presented in more detail in Chapter 2, we believe that our policy language can have applications to other DIFC systems, apart from Asbestos. Systems that use Asbestos labels to implement DIFC—such as HiStar [ZBK06] and Flume [KYB07]—can make direct use of the policy language and the Asbestos label setups it produces. For instance, the Asbestos labels produced by our parser for the HiStar example applications could be used directly to implement these policies in HiStar. Porting our application launcher though would be

more challenging, since its implementation is system specific.

5.5 Policy Language Translation Correctness

We aim to show that the label translations performed by our parser implement the desired communication patterns. When using the term "external compartment", we refer to a compartment that has been declared and initialized outside the policy description and may or may not be used within the policy description (using the policy language support for "external compartments"). The term "internal compartment" refers to compartments declared and initialized within the policy description code.

Lemma 1: Given two processes A and B, where A is contaminated with tag a at level **2** or level **3**. Then after B receives a message from A, one of the following holds:

- B is contaminated with a at the same level.
- B has privilege for a.

Proof: This statement follows from the basic Asbestos label check and the fact that contamination in Asbestos is transitive. The communication rules enforced by the Asbestos kernel, presented in Figure 3.2, dictate that when process B receives a message from process A, then B's tracking label will be updated based on the following rule: $\mathbf{T}_B \leftarrow (\mathbf{T}_B \sqcap \mathbf{T}^-) \sqcup (\mathbf{T}_E \sqcap \mathbf{T}^\star_B)$, where $\mathbf{T}_E = \mathbf{T}_A \sqcup \mathbf{T}^+$.

First let us assume that $\mathbf{T}_B(t) = \mathbf{1}$, i.e. B has a at the default level and does not hold privilege with respect to it. The message sent from A to B can not carry a $\mathbf{T}^-$ label giving B privilege with respect to a, since A does hold such privilege

and, consequently, can not grant it B. Furthermore, for simplicity, we assume that A is not making use of the message discretionary label $\mathbf{T}^+$ to contaminate B at a level higher than $\mathbf{T}_A(t)$.

In any of these cases if B does not hold privilege with respect to a (i.e., $\mathbf{T}_B(t) \geq \mathbf{0}$), then $\mathbf{T}_B(t)$ will be set to: $\mathbf{T}_B(t) = max(min(\mathbf{1}, \mathbf{3}), \mathbf{T}_E(t)) = max(min(\mathbf{1}, \mathbf{3}), max(\mathbf{T}_A(t), \mathbf{T}^+(t))) = max(\mathbf{1}, max(\mathbf{T}_A(t), \star)) = max(\mathbf{1}, \mathbf{T}_A(t)) = \mathbf{T}_A(t)$

If B holds privilege with respect to a (i.e., $\mathbf{T}_B(t) = \star$), then the rules applies as follows: $\mathbf{T}_B(t) = max(min(\star, \mathbf{3}), min(\mathbf{T}_E(t), \star)) = max(\star, \star) = \star$.

Therefore, unless B holds privilege with respect to a, it will become contaminated at the same level as A with respect to a, once it receives A's message.

Lemma 2: For any two internal compartments A and B where A can not communicate information to B (either $A < B$ or A ! B), there exists some tag a where A is contaminated with a at either level **2** or level **3**.

Proof: Our policy parser translates < and ! rules to Asbestos labels based on the third and second line of translation Tables 5.3, 5.4, 5.5 and 5.6, one for each of the four possible values of A's default.

We first examine the labels produced by the parser for A when the rule $A < B$ is translated, for each of the four possible values of A's default.

A's default	<>	!	<	>
A's tracking label with respect to a and a'	$\{a\,\mathbf{2}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{3}, a'\,\star\}$	$\{a\,\mathbf{3}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{2}, a'\,\star\}$

In all cases either $\mathbf{T}_A(a) = \mathbf{2}$ or $\mathbf{T}_A(a) = \mathbf{3}$.

Let us now examine A's labels, as they are produced by the parser when translating the rule A ! B, for each of the four possible values of A's default.

A's default	<>	!	<	>
A's tracking label with respect to a and a'	$\{a\,\mathbf{2}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{3}, a'\,\star\}$	$\{a\,\mathbf{3}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{2}, a'\,\star\}$

Again, in all cases either $\mathbf{T}_A(a) = \mathbf{2}$ or $\mathbf{T}_A(a) = \mathbf{3}$.

Lemma 3: For any two internal compartments A and B where $A < B$ or A ! B, then a process belonging to compartment B cannot receive a message sent by any process that belongs to A and is contaminated with A's tag a at the relevant level.

Proof: Based on Lemma 2, we know that if process P belongs to A then either $\mathbf{T}_P(a) = \mathbf{2}$ or $\mathbf{T}_P(a) = \mathbf{3}$.

Let us now examine the labels generated by our parser for compartment B, given either of the two rules: $A < B$ or A ! B.

Our policy parser translates < and ! rules to Asbestos labels based on the third and second line of translation Tables 5.3, 5.4, 5.5 and 5.6, one for each of the four possible values of A's default.

We first examine A's tracking label and B's clearance label, as generated by the translation of rule $A < B$, for all four possible values of A's default.

A's default	<>	!	<	>
A's tracking label with respect to a and a'	$\{a\,\mathbf{2}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{3}, a'\,\star\}$	$\{a\,\mathbf{3}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{2}, a'\,\star\}$
B's clearance label with respect to a and a'	$\{a\,\mathbf{1}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{2}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{2}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{1}, a'\,\mathbf{2}\}$

In all cases $\mathbf{T}_A(a) > \mathbf{C}_B(a)$ and therefore a process in compartment B can not receive messages from any processes belonging to compartment A—i.e., contaminated with respect to a at the relevant level.

Let us now examine A's tracking label and B's clearance label, as generated by the translation of rule A ! B, for all four possible values of A's default.

A's default	<>	!	<	>
A's tracking label with respect to a and a'	$\{a\,\mathbf{2}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{3}, a'\,\star\}$	$\{a\,\mathbf{3}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{2}, a'\,\star\}$
B's clearance label with respect to a and a'	$\{a\,\mathbf{1}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{2}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{2}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{1}, a'\,\mathbf{2}\}$

Again, in all cases $\mathbf{T}_A(a) > \mathbf{C}_B(a)$ and therefore a process in compartment B can not receive messages from any processes belonging to compartment A—i.e., contaminated with respect to a at the relevant level.

Lemma 4: For any two internal compartments A and B where $A < B$ or A ! B, assume B receives information from A via some chain of external compartments $A \to C_0 \to C_1 \to \cdots \to C_m \to B$. Then at least one of the C_i has privilege for A's tag a.

Proof: Lemma 4 follows from Lemma 1, Lemma 2 and Lemma 3: According to Lemma 2, the existence of either rule $A < B$ or rule A ! B means that A is contaminated with a at either **2** or **3** Let as assume that none of the external compartments C_i holds privilege with respect to A's contamination tag a. Then, when A sends a message to external compartment C_0, according to Lemma 1, C_0 will become contaminated with a at the same level as A (since lack of privilege means that C_0 can not avoid contamination). Similarly, when C_0 forwards the

message to C_1, C_1 will become contaminated with A's contamination. Eventually, C_m will receive the message from C_{m-1} and become contaminated with A a. If C_m attempts to forward the message to B, sending will fail because, according to Lemma 3, B can not receive a message carrying A's contamination.

Therefore, one of the compartments C_i, with $0 \leq i \geq m$, must hold privilege with respect to A's contamination.

Lemma 5: Given a chain of processes $A = C_0 \rightarrow D_0^* \rightarrow C_1 \rightarrow D_1^* \rightarrow C_2 \rightarrow \cdots \rightarrow D_{m-1}^* \rightarrow C_{m-1} \rightarrow D_m^* \rightarrow C_m = B$, where the C_i's are processes in internal compartments, and the D^* are independent, possibly empty, sequences of processes in external compartments. Assume that information travels from A to B. Then for every i, $0 \leq i < m$, we must have that either $C_i > C_{i+1}$, or $C_i <> C_{i+1}$, or there exists a process in D_i^* that has privilege for A's contamination tag.

Proof: This follows by using Lemma 4 for each sub-chain $C_i \rightarrow D_i^* \rightarrow C_{i+1}$. More specifically:

Let as assume that for sub-chain $C_i \rightarrow D_i^* \rightarrow C_{i+1}$ there is no process in D_i^* holding privilege with respect to A's contamination tag, and the rule between C_i and C_{i+1} is neither $C_i > C_{i+1}$ nor $C_i <> C_{i+1}$. Then the rule between C_i and C_{i+1} is either $C_i < C_{i+1}$ or C_i ! C_{i+1}, and any communication of information from C_i to C_{i+1} is impossible (goes against Lemma 4). Therefore, either there is a process in D_i^* holding privilege with respect to A's contamination tag, or the rule between C_i and C_{i+1} allows C_i to communicate information to C_{i+1}.

Lemma 6: For any two internal compartments A and B where $A <> B$ or $A > B$, then, immediately after labels are assigned by the launcher, B can receive a message sent by any process contaminated with A's tag a at the relevant level.

Proof: Let us examine the labels generated by our parser for compartments A and B, given either of the two rules: $A <> B$ or $A > B$.

Our policy parser translates <> and > rules to Asbestos labels based on the first and fourth line of translation Tables 5.3, 5.4, 5.5 and 5.6, one for each of the four possible values of A's default.

We first examine A's tracking label and B's clearance label, as generated by the translation of rule $A <> B$, for all four possible values of A's default.

A's default	<>	!	<	>
A's tracking label with respect to a and a'	$\{a\,\mathbf{1}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{3}, a'\,\star\}$	$\{a\,\mathbf{3}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{1}, a'\,\star\}$
B's clearance label with respect to a and a'	$\{a\,\mathbf{2}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{3}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{3}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{2}, a'\,\mathbf{2}\}$

In all cases $\mathbf{T}_A(a) \leq \mathbf{C}_B(a)$ (and of course $\mathbf{T}_A(a') \leq \mathbf{C}_B(a')$—since a' is only used to affect A's receiving ability) and therefore a process in compartment B can receive messages from any processes belonging to compartment A.

Let us now examine A's tracking label and B's clearance label, as generated by the translation of rule $A > B$, for all four possible values of A's default.

A's default	<>	!	<	>
A's tracking label with respect to a and a'	$\{a\,\mathbf{1}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{3}, a'\,\star\}$	$\{a\,\mathbf{3}, a'\,\mathbf{1}\}$	$\{a\,\mathbf{1}, a'\,\star\}$
B's clearance label with respect to a and a'	$\{a\,\mathbf{2}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{3}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{3}, a'\,\mathbf{2}\}$	$\{a\,\mathbf{2}, a'\,\mathbf{2}\}$

Again, in all cases $\mathbf{T}_A(a) \leq \mathbf{C}_B(a)$ and therefore a process in compartment B can receive messages from any processes belonging to compartment A.

Theorem: Assume internal compartment A can communicate information to internal compartment B. Then either there exists a chain of compartments $A = C_0 \rightarrow C_1 \rightarrow \cdots \rightarrow C_m = B$ in the configuration language that allows this communication, or there exists an external compartment with privilege for A's tag.

Proof: Using Lemma 5, we know that if information is communicated from A to B, then there must through a chain of compartments $A = C_0 \rightarrow D_0^* \rightarrow C_1 \rightarrow D_1^* \rightarrow C_2 \rightarrow \cdots \rightarrow D_{m-1}^* \rightarrow C_{m-1} \rightarrow D_m^* \rightarrow C_m = B$ such that for every i, $0 \leq i < m$, we must have that either $C_i > C_{i+1}$, or $C_i <> C_{i+1}$, or there exists a process in D_i^* that has privilege for A's contamination tag. If every sequence of external compartments D_i^* is empty (i.e., no external compartments participate in the chain), then—according to Lemma 5— for every pair of internal compartments C_i, C_{i+1} there must exist a rule allowing information to be communicated from C_i to C_{i+1}. Otherwise, if C_i and C_{i+1} are connected by rule $C_i < C_{i+1}$ or rule C_i ! C_{i+1}, Lemma 5 implies that a process belonging to D_i^* must hold privilege with respect to A's contamination tag.

5.6 Summary

Although correct policies are critical for the security of DIFC applications, the current DIFC programming model requires developers to perform these tasks at the Asbestos label level, making policy management challenging and error-prone—even in the case of relatively simple policies, like the one presented in Figure 4.1. We have identified certain practical reasons that make it hard for developers to perform policy management, including expressing policies directly at the code level, having to spread policy implementation in multiple code locations

and difficulty in reading, sharing and reasoning about the policy implementation. These issues, though important, are overshadowed in our experience by another problem: developers understand and model policies in terms of communication patterns, not labels.

We have proposed a policy description language that allows developers to describe application policies in terms of pairwise communication rules between application and system components. In particular, our policy language achieves multiple important goals that simplify policy management and therefore improve the DIFC programming:

- it allows developers to model policies as allowed and forbidden communication patterns expressed as pairwise communication rules
- it concentrates policy specification into one place, making policies easier to write and reason about
- it supports policy parameterization through the use of environment variables and file system pickles
- it models Asbestos event processes
- it models important runtime application properties such as communication ports
- it is translated to equivalent Asbestos label configurations by our language parser at reasonable runtime cost
- it supports application instantiation using our policy launcher

Our experience using the language shows that it is able to significantly improve the development experience in Asbestos by simplifying policy management

and the likelihood of bugs. The implementation of policy examples from other DIFC systems, such as HiStar, demonstrates that our policy language can find applications to DIFC systems other than Asbestos, and contribute in improving their programming model.

CHAPTER 6

Debugging Mechanisms for Asbestos

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. But Asbestos's strong isolation guarantees complicate the programming model and make application development harder.

For instance, the Asbestos Muenster application [BEK07] shows that Web services can be built from untrusted components. But how can those components be built? In a conventional development scenario, a service programmer builds their service using private infrastructure, such as a development server, over which they have full control. When things go wrong with the service, the programmer can examine the entire machine, including error logs, console, and process memory. Even when physical access to the machine is not possible, developers may collect and expose debugging information (e.g. through the Web browser) without any

information flow restrictions. Gathering and exposing information such as stack traces, system call traces, file operations, and communication behavior can be done with few or no restrictions.

The Muenster development model changes this significantly. An untrusted developer's Muenster service runs in an environment owned and built by other developers, who may not make their code public. 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 service developers cannot be allowed to utilize this gigantic channel, or any type of similar global privilege (e.g. conventional OSes' *root* user privilege). Instead, service code is constrained to follow stringent security policies, which often prevent that code from exporting information, including debugging information such as backtraces and error logs. This problem is unique to information flow controlled systems. In the presence of DIFC, debugging is rendered challenging because the 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.

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* [EK08] 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.

6.1 Label Errors

The high frequency and importance of label errors for Asbestos development make them a good working example for DIFC debugging. Let us consider a programmer trying to develop a new network service for AWS whose code appears in Figure 6.1. The programmer has no console access and the network terminal (i.e. telnet or Web browser) is her standard output. The user first creates the new tag *mytag* (i.e. a new compartment), used to protect her private data (line 6). Then the user creates a new file to store the private application data and sets the file labels so that readers get contaminated to $\{mytag\,\mathbf{3}\}$ and writers need to have $\{mytag\,\star\}$ (line 7). Then the user drops the tag from her send label (line 8). This is done for two (hypothetical) reasons: first, she doesn't need to hold "unnecessary" privilege, and second she wants to keep the process' send label as short as possible for performance. In this block of code, the user would run into a number of problems. The first bug is on line 14: the user drops tag *mytag* prematurely. The user is able to read the secret file (since she granted herself clearance to do so

```
 1  void
 2  worker_init(char ** argv, int argc) {
 3    tag_t mytag;
 4
 5    /* create new tag and use it to contaminate private file */
 6    sys_new_tag(&mytag, "my secret port");
 7    writefile(priv_file, Contamination={mytag 3, 1}, Clearance={mytag *, 3});
 8
 9    /* allow ourselves to access contamination {mytag 3} (i.e. read file) */
10    self_give_clearance(mytag, 3);
11    ...
12
13    /* prematurely drop {mytag *} privilege */
14    sys_tag_drop_privilege(mytag);
15    r = http_output("Initialization: success!");
16    ...
17    return;
18  }
19
20  int
21  main(int argc, char ** argv) {
22    worker_init(argv, argc);
23    ...
24    /* by reading file, we get contaminated with {mytag 3} since we dropped privilege */
25    read_from_my_file();
26
27    /* {mytag 3} contamination may not escape to the network. LABEL ERROR! */
28    http_output(input);
29    ...
30  }
```

Figure 6.1: Block of C-like code demonstrating possible bugs when developing for Asbestos.

on line 10), but she has dropped privilege to declassify information with respect to *mytag*. Reading the file gets her process contaminated to $\{mytag\,\mathbf{3}\}$. This prevents the process from being able to communicate with the outside world—in this case sending the HTTP response to the user's browser, on line 28. Due to this bug, the attempt to read the secret file on line 25 renders the process unusable, producing a label error the moment the user tries to use the service, since the process is lacking *mytag* privilege in order to remain unaffected from reading the secret the file and retain its ability to communicate with the network. When this label error occurs the remote user/developer will have no indication as to what went wrong. We would like the developer to be able to receive some kind of notice for the label error caused by the lack of declassification privilege with respect to

mytag.

An error like the one presented in Figure 6.1 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 (also referred to as 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. For instance, if the sending process P was informed that the message was not delivered to the destination process Q because P's contamination level with respect to the faulting handle was higher than Q's clearance, then information about Q's clearance is revealed to P. Similarly, if Q is informed that it was not able to receive a message sent to it because its clearance with respect to the faulting handle was insufficient, then information about P's contamination is revealed to Q. Therefore, in order to preserve

information flow, each debug message should also carry both processes' contaminations. This will ensure that the message may only be received by processes that have clearance to receive information from both the sender and the destination of the offending message.

In IFC terms, collecting debugging information belonging to various compartments requires privilege to declassify information with respect to those compartments. The data isolation properties application security policies are provide aim to reduce the amount of privilege each application developer holds. Consequently, the declassification of useful debugging information out of the relevant compartments is almost always impossible for developers, because the privilege they hold is inadequate.

However, being able to debug policy errors is essential for making the improving the DIFC programming model and therefore 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, or a user-private tag representing a subcompartment—such as the one represented by *mytag*. 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. If P and Q are in the process of being debugged it's likely that their labels consist of tags for which the

debugger has debugging privilege. 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.

Each application should be able to manage its own set of debugging privilege instances. Managing each of these instances modulates the amount of information that would be released through the debugging mechanism and is therefore considered a security-sensitive operation that should require explicit relevant privilege.

Furthermore, label-error debug messages contain information about the faulting tag. Since label rules in Asbestos are kernel-enforced, only the kernel holds information such as the details and nature of a label error required to form the relevant debug messages when necessary. Reporting debugging information for a label error caused by a tag *mytag* is equivalent to declassifying information out of *mytag*'s compartment. Therefore, instructing the kernel to report such label errors should require privilege with respect to *mytag*, to ensure that information flow control rules are not violated.

The privilege requirements of the debugging mechanism suggest that, following Asbestos's decentralized privilege management principles, debugging should allow each application to create and manage its own instances of the debugging primitives in a decentralized fashion.

To achieve these goals we implemented a new primitive called *debug domains* as the primary mechanism to facilitate decentralized debugging, while preserving information flow semantics.

6.2 Asbestos Debug Domains

The *debug domain* (DD) primitive represents 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.

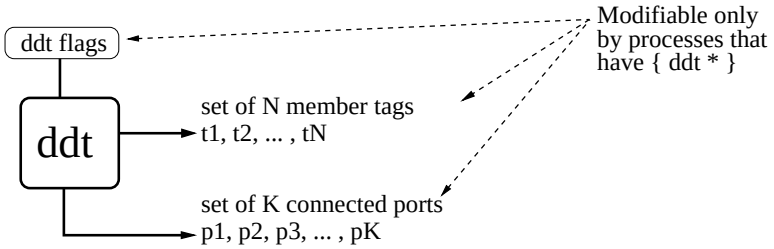

Figure 6.2: A debug domain represented by tag *ddt*, including its member tags, connected listening ports, and flags (debug domain properties). Modifying any debug domain property requires privilege with respect to *ddt*. Adding a member tag or a new connected port also requires privilege with the tag/port being added/connected.

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 6.2 illustrates a debug domain represented by tag *ddt*.

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.

6.3 Implementation

Like any other entity and resource in Asbestos, a debug domain is represented by a special tag type. A DD 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 instructing the kernel to create a new DD tag and initialize it to represent a new, empty DD. The caller of sys_new_tag() gets privilege with respect to the newly created tag and therefore the creator of the DD holds privilege over it and may exercise it to manage the DD's properties through calls to sys_debug_ctl(). Adding new member tags and connecting new ports to a DD not only requires privilege with respect to the DD ($\{ddt\star\}$), but also requires privilege with respect to the tags/ports being added/connected. The latter requirement is necessary because adding a new member tag *mytag* to a DD instructs the kernel to declassify debugging information with respect to *mytag*'s compartment, and $\{t\star\}$ should be required to do so. Similarly, when connecting a new port *myport* to a DD, it is important to hold privilege over *myport* before its receiving behavior is changed significantly by allowing debug messages to be sent to it.

The DD primitive is used in conjunction with different types of triggering events to implement different types of debugging applications. Our example applications include:

- 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).

A DD's *flags* are passed at creation time and specify DD properties, such as the types of triggering events DD members will be monitored for. For instance, a DD whose flags indicate that only label errors due to member tags are to be used as triggers, will not be considered for system call tracing debug messages—even if the system call trigger involves one of the DD members. A DD's flags may indicate more than one triggering event types the DD is interested in, instructing the kernel to consider it for multiple types of debug messages.

Each triggering event is handled by a wrapper function responsible for that type of error. Each event type asks for different kinds of handling. The application of debug domains to a class of triggering events mainly involves writing a new wrapper function to the debug device operations. The wrapper function performs the pre-processing of the event, formats the debug message payload, calculates the appropriate contamination the message should carry for this triggering event, decides what processes are supposed to get the message, and forwards the message to the lower level DD functions for delivery. The implementation is completed by inserting appropriate calls to the new wrapper function in the relevant positions of the kernel code.

Eventually, wrapper functions generate and send debug messages to all appropriate destination ports. Sending a debug message involves the following steps:

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 member. Discard the domains whose flags indicate that they are not monitoring 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.

Figure 6.3 presents a pseudo-code outline of the way the debug domain kernel mechanism functions when a triggering event occurs.

The most sensitive aspect of writing a debug wrapper function is identifying the proper label that needs to be attached to the message. Based on the circumstances under which the triggering event occurs and the payload of the generated message, the wrapper function needs to ensure that the debug message carries the contamination of all processes (explicitly or implicitly) involved. Our applications of debug domains in Section 6.4 demonstrate examples of debug message contamination.

Our implementation of DDs and their applications involved numerous changes in the Asbestos kernel. First, we implemented debug domains as a special tag

```
1: wait for trigger;
2:     if DDs are enabled:
3:         identify faulting tag;
4:         if faulting tag member of any DD:
5:             generate debug message;
6:             for each DD d the faulting is member of:
7:                 if d monitors the trigger type in question:
8:                     for each port p connected to d:
9:                         if p can get debug messages for trigger type:
10:                            send debug message to p
11:                        else:
12:                            continue;
13:                else:
14:                    continue;
15:        else:
16:            goto 1; // (i.e., ignore trigger)
17:    else:
18:        goto 1; // (i.e., ignore trigger)
```

Figure 6.3: The reaction of the debug domain kernel mechanism when a new triggering event occurs. Note that these operations are performed only if the Asbestos kernel has been compiled with debugging enabled.

type, able to keep track of member tags, connected ports and all necessary DD properties. Additionally, we implemented wrapper functions using DD functionality, identified kernel execution states that require debug message generation for each of the triggering event types and inserted calls to the wrapper functions—passing appropriate kernel state as arguments—at the relevant positions in the kernel code. Furthermore, we implemented new system calls necessary to create and manage debug domains.

At user level, we implemented the necessary interfaces (mostly system call wrappers) that allow developers to easily create and manage DDs as well as higher-level library calls that implement DD applications and system services, such as DIFC-safe system call tracing (*strace*()) and debugging libraries described in Section 6.4. Table 6.1 summarizes the major debug domain operations and indicates the privilege requirements for each of them.

Operation	Function	Privilege Required	Privilege Granted
Create a DD	**sys_new_dd(flags)** (calls *sys_new_tag()*	None	Privilege over new DD (caller gets $\{ddt\star\}$)
Add members or connect ports	**sys_debug_control(DD,** $t_1, t_2, \ldots, t_N$**)** where $t1, t2, \ldots, t_N$ is the list of tags/ports to be added/connected	Privilege with respect to the DD and new members or/and ports	None
Remove members or disconnect ports	**sys_debug_control(DD,** $t_1, t_2, \ldots, t_N$**)** where $t_1, t_2, \ldots, t_N$ is the list of tags/ports to be removed/disconnected	Privilege with respect to the DD and removed members and/or ports	None
Change DD properties (flags)	**sys_debug_control(DD, flags)**	DD privilege	None
Destroy a DD	**sys_destroy_dd(DD)** (calls *sys_tag_dissociate(DD))*	DD Privilege	None

Table 6.1: Debug domain creation and management operations. Notice that creating and destroying a DD is performed by creating and dissociating the relevant (special DD) tags. Consequently, destroying a DD does not necessarily destroy it, since all Asbestos tags (including those representing DDs) are reference counted.

6.4 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

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

Figure 6.4: Code example where DDs (used by the debugger) would help diagnose a bug causing a label error (because of dropping privilege prematurely on line 22).

Message type	Message label
Label-error debugging	$(\mathbf{T}_P \sqcup \mathbf{T}_Q) \sqcap \{t_1 \star, t_2 \star, \ldots, t_n \star, p_l \star, \mathbf{3}\}$
Syscall tracing	$\mathbf{T}_P \sqcap \{t_1 \star, t_2 \star, \ldots, t_n \star, p_l \star, \mathbf{3}\}$
Label tracing	$\mathbf{T}_P \sqcap \{t_1 \star, t_2 \star, \ldots, t_n \star, p_l \star, \mathbf{3}\}$
Process exiting	$\mathbf{T}_P \sqcap \{t_1 \star, t_2 \star, \ldots, t_n \star, p_l \star, \mathbf{3}\}$

Table 6.2: Labels for messages generated by the four debug domain applications. $\sqcup$ is the least upper bound operator and $\sqcap$ the greatest lower bound operator. $t_1, t_2, \ldots, t_n$ 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.

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 Table 6.2.

To make the use of DDs more concrete, we revisit our AWS example presented in Figure 6.1. Figure 6.4 demonstrates the use of debug domains in order to facilitate debugging for the problems we diagnosed earlier. Function *worker_init*() is modified as follows: it creates a debug domain represented by tag *ddt* and configured to receive label errors from the kernel (line 13). The port *myport* is connected to the debug domain (at creation time), and the "suspect" tag *mytag* is added to its members (lines 13–14). This allows the owner of *myport* to receive debugging messages whenever a label error in relation to *mytag* occurs. A new debugger process is spawned and ownership of *myport* is transfered to it (lines 17–18). Similar to Figure 6.1, the new version contains the same bug:

on line 22 the process mistakenly drops privilege with respect to *mytag* and consequently becomes contaminated with $\{mytag\,\mathbf{3}\}$ when the main function tries to access the private file on line 33. When the label error occurs during the attempt to communicate with the network daemon on line 37, the kernel generates a new label-error debug message and sends it to all subscriber ports connected to the label-error debug domains that *mytag* is a member of. In this example, the message will be sent to *myport*, which belongs to the debugger. The debugger has $mytag\star$ 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 37. Figure 6.5 illustrates the kernel components of the mechanism.

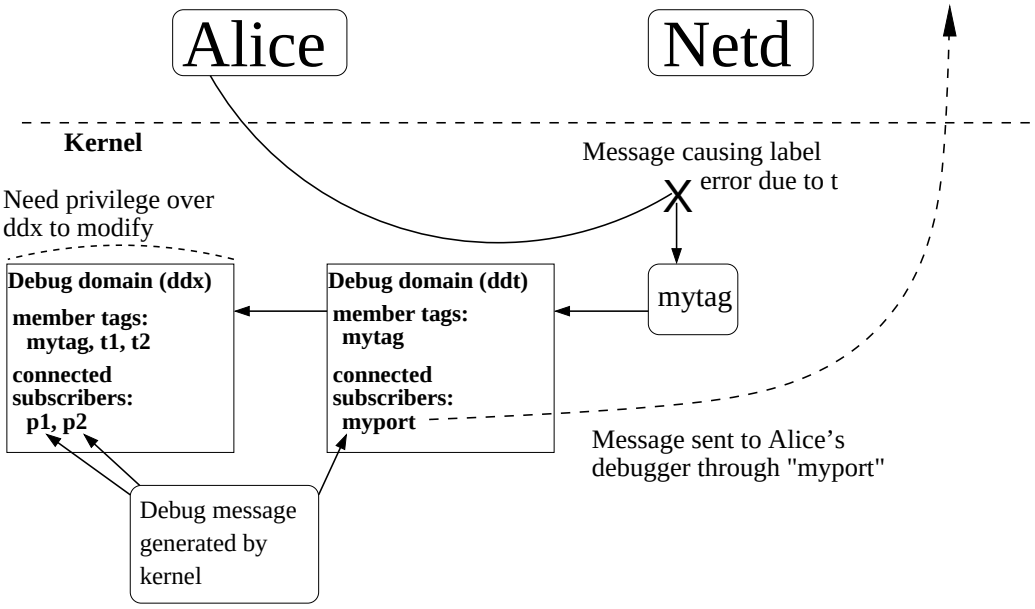

Figure 6.5: When the Asbestos kernel identifies the label error, it checks whether the "faulting tag" (*mytag*) belongs to any debug domains whose flags are configured for label error debugging, such as the ones represented by *ddt* and *ddx*. The kernel will sent the generated debug message to all connected subscriber ports: $p1$ and $p2$ from *ddx* and *myport* from *ddt*.

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 Policy errors often often manifest themselves as changes to application processes' labels that lead to unexpected, buggy behavior. For instance, dropping privilege with respect to *mytag* prematurely on line 22 of Figure 6.4 leads to a label change ($\{mytag\star\}$ disappears from the process's tracking label). In this case as well as in various other cases, being able to track changes to process labels can provide valuable information for solving policy bugs. We have used debug domains to facilitate policy debugging by implementing 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 6.4 the developer can use label history to identify the calls that led to dropping *mytag* privilege (line 22) and getting contaminated with respect to *mytag* (line 33).

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

6.5 Resource Annotation

Asbestos represents resources primarily as tags. When debugging an application, a developer often needs to identify and reason about opaque, 61-bit tag values. To facilitate debugging we introduced an optional, human understandable name/description for each tag, that is stored maintained by the kernel and allows us to explicitly annotate resources. To prevent information leakage through tag names, we only allow a tag name to be set on tag creation time (Figure 6.4, line 6). Furthermore, access to a tag's name requires either ownership of the tag or privilege with respect to it. (note that multiple processes may have a tag at $\star$, thus holding privilege with respect to that tag, but at most one may hold receive rights — i.e. ownership).

6.6 Experiences and Evaluation

We want to demonstrate that debug domains are able to deliver policy-safe debug messages with reasonable overhead. To that end, 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,

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

Table 6.3: 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.

	Add member/ connect port	Debug message generation			
		label (to sender/rcvr)	syscall	exit	*send()*
cycles	2291 / 1275	31625 / 26316	12548	5606	at least 31140

Table 6.4: 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. For reference we have included the cost of an Asbestos *send()* system call (used to send a message) in the less "costly" case (no payload, and no discretionary labels attached to message).

connected on a 1Gbps switch. We ran throughput measurements and compared our results to the unmodified AWS. As shown in Table 6.3, 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 [VEK07].)

We ran micro-benchmarks to measure the average cost of some major kernel operations for debugging. The results, presented in Table 6.4, 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. Table 6.4 also presents the cost of the base case for the *send()*

system call, for reference. Notice that the cost of sending a message depends on various factors: for example the cost is almost linear to the size of the discretionary labels attached to the message. The base case presented on Table 6.4 corresponds to zero sized payload and no discretionary labels attached to the message.[1]

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, 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 the Muenster uploader of untrusted worker processes that would restart an AWS worker 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.

[1]For instance, if we use all four discretionary labels with 10, 100, 200 and 500 tags in each of them the cost increases to 25035, 242890, 488224 and 1234840 cycles respectively.

6.6.1 Discussion

We implemented and tested debug domains in Asbestos, but we believe that the benefits of systematically modelling debugging privilege can improve other DIFC systems' programming model. Even DIFC systems whose design reduces the amount of system state management challenges compared to Asbestos (e.g. HiStar and Flume) lack a systematic way to represent and manage debugging privilege.

6.7 Summary

Policy bugs are a serious problem in DIFC systems: not only do they interfere with the application's normal operation, but they may also pose serious threats to application security. Debugging requires exposure of system information related to the problem and that often contrasts the security guarantees of the system: unchecked release of system state information almost always leads to leaking information from a compartment, effectively violating information flow. Additionally, an application may hold insufficient privilege to inspect system state necessary for debugging.

To address the policy debugging challenges in Asbestos, we introduced the debug domain kernel mechanism that formally models debugging privilege. Debug domains implement a decentralized debugging primitive that adheres to the information flow policies enforced by Asbestos. A debug domain represents privilege to declassify debugging information out of a set of compartments represented by a set of member tags. A second set of tags associated with the debug domain represents the communication ports that to which the declassified debugging information will be released.

We have used debug domains to implement tolls such as a debugger, system call and label tracing libraries and we have found that they are a flexible mechanism that can simplify development and improve the programming model, through a number of features:

- developers can use debug domains to implement DIFC-safe debugging tools
- debugging privilege is managed in a decentralized fashion—in the spirit of DIFC
- developers are able to perform fine-grained management of debugging privilege (at the granularity of compartments—as opposed to using an excessively privileged debugger)
- debug domains are flexible enough to implement multiple different types of debugging, such as label error debugging, system call tracing, label history tracing and exiting process notification
- each process can create and manage an arbitrary number of debug domains, with disjoint or overlapping member tags and debugging roles (e.g. one or more label error debug domains)
- the runtime overhead of debug domains is below 2% and the cost of debug domain operations (add/remove members, generate debug messages etc) is reasonable

CHAPTER 7

Conclusion

This thesis examined how we can improve the the programming model of decentralized information flow control (DIFC) systems by providing DIFC-safe system management mechanisms.

Our work was motivated by our experiences with the Asbestos DIFC system, but our proposed solutions can be used to improve system management in other DIFC systems as well.

Asbestos makes non-discretionary access control mechanisms available to unprivileged users by implementing DIFC through the Asbestos labeling system. Developers are given fine-grained, end-to-end control over the flow of information in the system—without requiring any type of special system privilege—which they can use to define application policy. Application policy specifies the rules that govern information flow for a specific application and lies at the heart of any DIFC application. The restrictions imposed by each separate policy as well as the combination of all policies in conjunction, affect system management tasks and introduce new challenges for developers.

In this thesis we investigated and proposed solutions for two important system management challenges in Asbestos: policy management and debugging. We proposed a policy description language able to express a wide variety of policies in a human-friendly way. We have developed tools that translate high-level policy de-

scriptions 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. To that end, 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.

7.1 Open Research Problems

This work is an important first step towards a better DIFC programming model, that identified a fundamental problem: system management problems render the DIFC programming model challenging for developers. We were able to investigate and propose solutions for two pressing system management issues, but making DIFC easier to work with and adopt requires further work that will help improve the programming model.

Our policy description language is a first step towards better DIFC policy management. The language interface could be improved in many interesting ways that would give developers better control over policy description. For instance, 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. It would be useful 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).

Our debug domain abstraction is able to represent system state management privilege and allowed us to implement some preliminary debugging mechanisms and tools. It would be very useful 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.

Although policy management and debugging are two very important system management problems, there are certain other interesting system management issues that require a solution. For instance, resource management is another sensitive system management issue that Asbestos is currently not dealing with. Addressing this problem in a DIFC-safe manner is a challenging task. Furthermore it would be interesting to investigate whether the debug domain abstraction is versatile enough to be used to model solutions for different classes of system management problems, such as resource management.

Relative to policy management, it would also be interesting 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). Being able to inspect runtime policy at a higher level would be particularly useful for policy debugging, especially in the case of bugs that appear only at runtime due to the interaction of the application 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.

Although this work is only a first step, hopefully these and other programma-

bility improvements will bring the security benefits of DIFC to a wider community of developers.

APPENDIX A

Policy Examples

A.1 The Asbestos Web Server Policy

```
# Global default for compartments is "<>"
default <>
# Database compartment, fully isolated
comp DB {
        default !
}
# Compartments for thedemux, the ID daemon, and the DB proxy
comp DEMUX IDD DBV {
        default <>
}
# Compartment for the AWS worker base process
comp W {
        default <
}
# External network daemon. Belongs to external, isolated compartment
# initialized localy through environment variables.
exec netd {
        belongs (env NET_TS NET_TR default !)
        # netd's external (pre-existing) port, initilized localy
        # through an environment variable.
        port NETROOT {
                env NETDHANDLE
        }
}
# Database executable
exec database {
# Path to binary
        bin /okdb
# The databese executable should belong to the CDB compartment
        belongs DB
# Database port, used between for communication between the
# the database and the DB proxy. It is a restricted port
# (i.e. privilege is required to send messages to it) and its
# label belongs to CDB.
        port DBP {
                type restricted
                name "database port"
                belongs DB
        }
# Environment variables made available to the database.
# Notice that "env*" means that the database will also be
# granted privilege with respect to the relevant ports
# while plain "env" does not grant privilege.
```

```
        env* OKDBHANDLE=port:DBP
        env* YOURPORT=port:DBP
}
# Database initializer process. Fills the database with dummy user data.
exec dbinit {
        bin /okdb-init -a
        belongs DB
        env* OKDBHANDLE=port:DBP
}
# Identity daemon. Used to authenticate users.
exec id_daemon {
        bin /idd -db
        belongs IDD
        port HSYSTEM {
                type restricted
        }
        port HANYONE {
                type open
        }
        env HSYSTEM=port:HSYSTEM
        env HANYONE=port:HANYONE
}
# Database proxy.
exec dbv {
        bin /okws-dbv
        belongs DB
        belongs IDD
        port DBVPORT {
                type open
                name "okws-dbv: self tag"
        }
        port DBVPRIVATE {
                type restricted
                belongs DB
        }
        env HSYSTEM=port:HSYSTEM
        env HANYONE=port:HANYONE
        env MYPORT=port:DBVPORT
        env MYPORTPRIV=port:DBVPRIVATE
}
# Five different AWS worker base processes.
# Each of them provides a Web service (Web-shell, Sql front-end,
# login facility, user profile edit and profile view).
# Each base process uses event processes to server different users.
exec worker1 worker2 worker3 worker4 worker5 {
        bin /okws-shell sh
        bin /okws-sql sql
        bin /okws-login login
        bin /okws-edit edit
        bin /okws-view view
        belongs W
        port WP {
                type open
        }
        port WV {
                # port that will be used as worker verify tag
                type open
        }
        port DP {
                type restricted
# This port belongs to demux. It is declared here
# because we need one of these per worker.
```

```
                owner demux
        }
        dynexec USER {
                source demux
                belongs (env USER_S USER_R default <)
                file user-tmpl
                port WORKER\_PORT {
                        type restricted
                }
                port UG {
                        type restricted
                        owner parent
                        belongs (env USERX USERXP default <)
                }
                env WRPORT=port:WORKER\_PORT
                env* DPORT=port:DEMUX_USERP
                env* UG=port:UG
}
        env NETROOT=port:NETROOT
        env* SELFPORT=port:WP
        env* DBP=port:DBP
        env  UGRANT=port:UG
        env DEMUXPORT=port:DEMUX_USERP
        env* DP=port:DP
}
exec demux {
        bin /okws-demux sh sql login edit view
        belongs DEMUX
        port DEMUX_USERP {
                type open
        }
        port DEMUXPORT {
                type open
        }
        # this actually translates to manny WPORTS:
        # WPORT1=port:WP1, WPORT2=port:WP2 etc
        env WPORT=port:WP
        env DEMUXUSERP=port:DEMUX_USERP
        env SELFPORT=port:DEMUXPORT
        env* WORKERCTL1=control:worker1
        env* WORKERCTL2=control:worker2
        env* WORKERCTL3=control:worker3
        env* WVERIFY=port:WV
        env  NETDHANDLE=port:NETROOT
        #env HSYSTEM=port:HSYSTEM
        env  IDHANDLE=port:HANYONE
        env  DP=port:DP
}

# Pair-wise communication Rules
DB <> DBV
W <> netd
W <> DEMUX
USER <> netd
USER <> DBV
USER <> DEMUX
```

A.2 HiStar's ClamAV Policy

```
# The global default for compartments (i.e. the default default)
default <>
# We first declare the three compartments
comp USER {
        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
        belongs (env USER_S USER_R default !)
        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
}

# Pair-wise communication Rules
PRINTER <> AV
PRINTER <> USER
AV < USER
```

A.3 Policy similar to HiStar's VPN isolation

```
# We declare the five compartments
comp VPN INTERNET {
default <
```

```
}
comp INTERNET_IPSTACK VPNCLIENT {
default <>
}
comp NETD {
        default !
}
# Both the vpn browser and the vpn IP stack belong to CVPN
exec browser_vpn ipstack_vpn {
        belongs VPN
}
exec ipstack_internet {
        belongs INTERNET
}
exec browser_internet {
        belongs INTERNET
        belongs INTERNET_IPSTACK
}
exec vpn_client {
        belongs VPN
}
exec netd {
        belongs NETD
        belongs INTERNET
}

# Pair-wise communication Rules
VPNCLIENT <> VPN
VPNCLIENT <> INTERNET
INTERNET_LWIP <> NETD
```

A.4 Policy similar to Jif's hospital example

```
# The external compartment the "hospital" process belongs to
comp H {
        env HOSPITAL_S HOSPITAL_R default <>
}
# Compartments for data extractor, researchers, statistics package, and DB
comp E R SP DB {
        default !
}
# Compartment for the process that outputs results of study
comp OUT {
        env OUT_S OUT_R default <>
}

# Pair-wise communication Rules
H <> E
E > R
R <> SP
SP <> DB
R > OUT
```

References

[BCZ03] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. "Capriccio: Scalable Threads for Internet Services." In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 268–281, Bolton Landing, Lake George, NY, October 2003.

[BEK07] Micah Brodsky, Petros Efstathopoulos, Frans Kaashoek, Eddie Kohler, Maxwell Krohn, David Mazieres, Robert Morris, Steve VanDeBogart, and Alexander 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`.

[CLM07] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. "Secure Web Applications Via Automatic Partitioning." In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.

[CVM07] Stephen Chong, K. Vikram, and Andrew C. Myers. "SIF: Enforcing Confidentiality and Integrity in Web Applications." In *Proceedings of the USENIX Security Symposium 2007*, 2007.

[Den76] Dorothy E. Denning. "A Lattice Model of Secure Information Flow." *Communications of the ACM*, **19**(5), May 1976.

[Dep85] Department of Defense. *Trusted Computer System Evaluation Criteria (Orange Book)*, December 1985. DoD 5200.28-STD.

[EK08] Petros Efstathopoulos and Eddie Kohler. "Manageable Fine-Grained Information Flow." In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Galsgow, UK, April 2008.

[EKV05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. "Labels and Event Processes in the Asbestos Operating System." In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, England, October 2005.

[Faca] "Facebook Developers." `http://developers.facebook.com/`.

[Facb] "Facebook Privacy Breach Exposed Users' Hidden Dates Of Birth." http://www.sophos.com/pressoffice/news/articles/2008/07/facebook-birthday.html.

[Facc] "Security Lapse Exposes Facebook Photos." http://www.msnbc.msn.com/id/23785561/.

[Fra00] Timothy Fraser. "LOMAC: Low Water-Mark Integrity Protection for COTS Environments." In *Proceedings of 2000 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[GJS05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, 2005.

[Gol73] R. P. Goldberg. "Architecture of virtual machines." In *Proceedings of AFIPS National Computer Conference*, volume 42, June 1973.

[Har88] Norman Hardy. "The Confused Deputy (or why capabilities might have been invented)." *Operating Systems Review*, **22**(4):36–38, October 1988.

[KC03] Samuel T. King and Peter M. Chen. "Operating System Support for Virtual Machines." In *Proceedings of 2003 USENIX Annual Technical Conference*, San Antonio, TX, June 2003.

[KEF05] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. "Make Least Privilege a Right (Not a Privilege)." In *Proceedings of 10th Hot Topics in Operating Systems Symposium (HotOS-X)*, Santa Fe, NM, June 2005.

[KH84] Paul A. Karger and Andrew J. Herbert. "An Augmented Capability Architecture to Support Lattice Security and Traceability of Access." In *Proceedings of 1984 IEEE Symposium on Security and Privacy*, Oakland, CA, April 1984.

[Kro04] Maxwell Krohn. "Building Secure High-Performance Web Services with OKWS." In *Proceedings of 2004 USENIX Annual Technical Conference*, Boston, MA, June 2004.

[KYB07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. "Information Flow Control for Standard OS Abstractions." In *Proceedings of the*

21th ACM Symposium on Operating Systems Principles (SOSP '07), Stevenson, Washington, October 2007.

[KZB90] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. "A VMM Security Kernel for the VAX Architecture." In *Proceedings of 1990 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1990.

[Lan81] Carl E. Landwehr. "Formal Models for Computer Security." *ACM Computing Surveys*, **13**(3):247–278, September 1981.

[Lem05] Robert Lemos. "Payroll site closes on security worries.", February 2005. `http://news.com.com/2102-1029_3-5587859.html`.

[Lem06] Robert Lemos. "UCLA alerts 800,000 to data breach.", December 2006. `http://www.securityfocus.com/news/11429`.

[liv] "Livejournal S2 manual." `http://www.livejournal.com/doc/s2/`.

[LM02] Jeff Levinger and Rita Moran. "Oracle Label Security Administrator's Guide.", March 2002. `http://tinyurl.com/hu4qz`.

[LS01] Peter Loscocco and Stephen Smalley. "Integrating Flexible Support for Security Policies into the Linux Operating System." In *Proceedings of 2001 USENIX Annual Technical Conference—FREENIX Track*, June 2001.

[MBM06] Karl MacMillan, Joshua Brindle, Frank Mayer, David Caplan, and Jason Tang. "Design and Implementation of the SELinux Policy Management Server." In *Proceedings of 2006 Security Enhanced Linux Symposium*, Baltimore, MA, March 2006.

[ML00] Andrew C. Myers and Barbara Liskov. "Protecting Privacy using the Decentralized Label Model." *ACM Transactions on Computer Systems*, **9**(4), October 2000.

[MMN90] Catherine Jensen McCollum, Judith R. Messing, and LouAnna Notargiacomo. "Beyond the Pale of MAC and DAC—Defining New Forms of Access Control." In *Proceedings of 1990 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1990.

[MR92] M. Douglas McIlroy and James A. Reeds. "Multilevel Security in the UNIX Tradition." *Software—Practice and Experience*, **22**(8), August 1992.

[Mye99] Andrew 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*, New York, NY, USA, 1999.

[New05] News10. "Hacker Accesses Thousands of Personal Data Files at CSU Chico.", March 2005. `http://tinyurl.com/2syo7x`.

[Ope] "OpenSocial." `http://code.google.com/apis/opensocial/`.

[Ora07] "Oracle Label Security For Privacy and Compliance.", June 2007. "An Oracle White Paper".

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. "Flash: An efficient and portable Web server." In *Proceedings of 1999 USENIX Annual Technical Conference*, pp. 199–212, Monterey, CA, June 1999.

[RB04] Walid Rjaibi and Paul Bird. "A Multi-Purpose Implementation of Mandatory Access Control in Relational Database Management Systems." In *Proceedings of 30th Very Large Data Bases Conference (VLDB '04)*, Toronto, Canada, August 2004.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. "The Protection of Information in Computer Systems." *Proceedings of of the IEEE*, **63**(9), September 1975.

[SSF99] Jonathan S. Shapiro, Jonathan Smith, and David J. Farber. "EROS: A fast capability system." In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pp. 170–185, Kiawah Island, SC, December 1999.

[SSL99] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. "The Flask Security Architecture: System Support for Diverse Security Policies." In *Proceedings of the 8th USENIX Security Symposium*, Washington, DC, August 1999.

[Tro06] Rebecca Trounson. "Major Breach of UCLA's Computer Files." *Los Angeles Times*, December 12, 2006, 2006. `http://www.latimes.com/news/local/la-me-ucla12dec12,0,7111141.story`.

[VEK07] Steve VanDeBogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. "Labels and Event Processes in the Asbestos Operating System." *ACM Transactions on Computer Systems*, **25**(4):11:1–11:43, November 2007.

[VMw01] VMware. "VMware and the National Security Agency Team to Build Advanced Secure Computer Systems.", January 2001. `http://www.vmware.com/pdf/TechTrendNotes.pdf`.

[Wet99] David Wetherall. "Active network vision and reality." In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[Wik] "Wikipedia, The Free Encyclopedia." `http://www.wikipedia.org/`.

[WMV03] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. "The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0." In *Proceedings of 2003 USENIX Annual Technical Conference*, San Antonio, TX, June 2003.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. "Scale and Performance in the Denali Isolation Kernel." In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pp. 195–210, Boston, MA, December 2002.

[ZBK06] Nickolai B. Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. "Making Information Flow Explicit in HiStar." In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.