Laminar: Practical Fine-Grained Decentralized Information Flow Control

Indrajit Roy Donald E. Porter Michael D. Bond Kathryn S. McKinley Emmett Witchel
Department of Computer Sciences
The University of Texas at Austin
{indrajit, porterde, mikebond, mckinley, witchel}@cs.utexas.edu

Abstract

Decentralized information flow control (DIFC) is a promising model for writing programs with powerful, end-to-end security guarantees. Current DIFC systems that run on commodity hardware can be broadly categorized into two types: language-level and operating system-level DIFC. Language level solutions provide no guarantees against security violations on system resources, like files and sockets. Operating system solutions can mediate accesses to system resources, but are inefficient at monitoring the flow of information through fine-grained program data structures.

This paper describes Laminar, the first system to implement decentralized information flow control using a single set of abstractions for OS resources and heap-allocated objects. Programmers express security policies by labeling data with secrecy and integrity labels, and then access the labeled data in lexically scoped *security regions*. Laminar enforces the security policies specified by the labels at runtime. Laminar is implemented using a modified Java virtual machine and a new Linux security module. This paper shows that security regions ease incremental deployment and limit dynamic security checks, allowing us to retrofit DIFC policies on four application case studies. Replacing the applications' ad-hoc security policies changes less than 10% of the code, and incurs performance overheads from 1% to 56%. Whereas prior DIFC systems only support limited types of multithreaded programs, Laminar supports a more general class of multithreaded DIFC programs that can access heterogeneously labeled data.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls

General Terms Languages, Performance , Security

Keywords Information flow control, Java virtual machine, operating systems, security region

1. Introduction

As computer systems support more aspects of modern life, from finance to health care, security becomes increasingly important. Current security policies and enforcement mechanisms are typically sprinkled throughout an application, making security policies difficult to express, change, and audit. Operating system security abstractions, such as file permissions and user IDs, are too coarse to express many desirable policies, such as protecting a user's financial data from an untrusted browser plug-in. Furthermore, poor integration of programming language (PL) constructs and operating system (OS) security mechanisms complicates the expression and enforcement of security policies. For example, a user's credit card number should not be broadcast on the network, whether the number originates in a file or a data structure. Files and data structures are currently governed by completely distinct security mechanisms, requiring developers to understand both mechanisms. This paper addresses these issues by integrating PL and OS security abstractions, allowing application developers to express uniform security policies that are enforced at all layers of the software stack.

The decentralized information flow control (DIFC) security model [21] expresses policies based on how applications use data and more naturally matches how developers and users think of security policies than traditional security mechanisms. For instance, traditional access control mechanisms are all-or-nothing; once an application has the right to read a file, it can do anything with that file's data. In contrast, DIFC enforces more powerful policies, such as permitting an application to read a file, but disallowing the broadcast of the contents of that file over an unsecured network channel. A DIFC *implementation* dynamically or statically enforces end-to-end user specified security policies by tracking information flow throughout the system.

The DIFC model provides security by allowing users to associate secrecy and integrity labels with data and restricting the flow of information according to these labels. Secrecy guarantees prevent sensitive information from escaping the system, and integrity guarantees prevent external information from corrupting the system.

As an example of the DIFC model, consider Alice and Bob who want to schedule a meeting while keeping their calendars mostly secret. Alice and Bob each place a *secrecy label* on their calendar file, and then only a thread with those secrecy labels can read it. Once a thread has a secrecy label, it has been *tainted* by that label, and can no longer write to an unlabeled output, such as standard output or the network. If the thread has the capability to *declassify* the information, it may remove the secrecy label and then write the data to an unlabeled output. In the calendar example, the program obtains both Alice and Bob's secrecy label to read both calendar files, but then it cannot remove the labels. When the thread is ready to output an acceptable meeting time, it must call a function that then declassifies the result. The declassification function checks that its output contains no secret information. For example, the output is simply a date and does not include Bob's upcoming visit to the doctor.

DIFC provides two key advantages—clear rules for the legal propagation of data through a program, and the ability to localize security policy decisions. In the calendar example, the secrecy labels ensure that any program that can read the data cannot leak the data, whether accidentally or intentionally. The label is tied to the data, and it restricts who may access the data. The decision to declassify is localized to a small piece of code that can be closely audited. The result is a system where security policies are easier to express, maintain, and modify. DIFC can be supported at the language level [21, 22], in the operating system [15, 29, 31], or in the architecture [28, 32]. Each approach has strengths and limitations. Language-based DIFC systems rely on extensive type sys-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'09, June 15–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-392-1/09/06...$5.00

tem changes, wrap standard libraries, augment types throughout the entire program, and modify the program structure. OS-based DIFC systems have trouble enforcing data flow through program data structures because page mappings are an inefficient mechanism to control permissions for most user-defined data structures. Architecture-based solutions track labels on data and CPUs in hardware and signal violations, but still require trusted software to manage the labels. We limit the scope of this paper to DIFC implementations on commodity hardware.

This paper introduces a new DIFC system, Laminar, that provides a common security abstraction and labeling scheme for program objects and OS resources, such as files and sockets. By combining the strengths of PL and OS techniques, while minimizing their limitations, developers express comprehensive security policies and then the Laminar virtual machine (VM) and OS enforce these policies.

Laminar can be incrementally deployed—developers need only modify the security-sensitive portion of their programs. Trusted and untrusted threads, labeled and unlabeled files, sockets, data structures, and so on, coexist in the system. In contrast, existing DIFC languages require pervasive program modifications to label data structures, variables, functions, and return types [20, 26]. Because they rely on whole-program static analysis for enforcement of DIFC rules, they also exclude features such as dynamic class loading and multithreading.

Laminar introduces lexically scoped *security regions*. Security regions limit the scope of non-trivial DIFC enforcement, which makes it easier to develop, deploy, and audit DIFC programs. Security regions also reduce the overhead of dynamic security checks. All operations on labeled data must occur within security regions. A typical security region might read a labeled configuration file and parse it into a labeled data structure. Since all code that manipulates the labeled configuration data structures resides in a security region, it is easier to identify and audit. Code unrelated to the data structure needs no modification.

The contributions of this paper are:

1. The design of Laminar, the first system with unified PL and OS mechanisms for enforcing DIFC. Laminar features a novel division of responsibilities among the programming model, VM, and OS.
2. The introduction of security regions, an intuitive primitive that eases deployment, security programming, implementation, and auditing.
3. An implementation of Laminar that makes modest additions to Jikes RVM [2] (a Java virtual machine), and the Linux operating system.
4. Four case studies that retrofit security policies onto existing code. These case studies require modification of less than 10% of the total code base and incur overheads from Laminar ranging from 1% to 56%.

These advantages and initial results suggest that integrating PL and OS support incurs low overheads and allows the application developer to write fine-grained security policies that encompass program data structures as well as system resources.

2. Related work

Previous DIFC systems have either used only PL abstractions or OS abstractions. Laminar instead enforces DIFC rules for Java programs using an extended JVM and OS. By unifying PL and OS abstractions for the first time with a seamless labeling model, Laminar combines the strengths of previous approaches and further improves the DIFC programming model.

From IFC to DIFC. Information flow control (IFC) stemmed from research in multi-level security for defense projects [11]. In the original military IFC systems [14], an administrator must allocate all labels and approve all declassification requests. Modern mandatory access control (MAC) systems, like security-enhanced Linux (SELinux), also limit declassification and require a static collection of labels and principals. Decentralized information flow control (DIFC) systems allow individual applications to allocate labels and declassify data for their labels, providing a richer model for implementing security policies [21].

Language-based DIFC. Language-based DIFC systems [20, 22, 26] augment the type system to include secrecy and integrity constraints enforced by the bytecode generator. These systems label program data structures and objects at a fine granularity, but require programming an intrusive type system or an entirely new language. These language based systems trust the whole operating system and provide no guarantees against security violations on system resources, like files and sockets.

OS-based IFC. Asbestos [29] and HiStar [31] are new operating systems that provide DIFC properties. Flume [15] is a user-level reference monitor that provides DIFC guarantees without making extensive changes to the underlying operating system.

OS DIFC systems provide little or no support for tracking information flow through application data structures with different labels. Flume tracks information flow at the granularity of an entire address space. HiStar can enforce information flow at page granularity and supports a form of multithreading by requiring each thread to have a page mapping compatible with its label. Using page table protections to track information flow is expensive, both in execution time and space fragmentation, and complicates the programming model by tightly coupling memory management with DIFC enforcement. Laminar supports a richer, more natural programming model in which threads may have heterogeneous labels and access a variety of labeled data structures. For example, all of our application case studies use threads with different labels.

Laminar provides DIFC guarantees at the granularity of lexically scoped code blocks and data structures with modest changes to the VM. It also adds a security module to a standard operating system, as opposed to Asbestos and HiStar, which completely rewrite the OS. Most of Laminar's OS DIFC enforcement occurs in a security module whose architecture is already present within Linux (Linux security modules [30] (LSM)). The Laminar OS does not need Flume's *endpoint* abstraction to enforce security during operations on file descriptors (e.g., writes to a file or pipe), as the kernel-level reference monitor can check the information flow for each operation on a file descriptor.

Laminar adopts the label structure and the label/capability distinction derived from JIF and used by Flume. Capabilities in DIFC systems are formally defined in the next section. They are distinct from the capabilities used in capability-based operating systems. Capability-based operating systems, like EROS [25], use pointers with access control information to combine system and language mechanisms for stronger security. However, capability systems cannot enforce DIFC rules, and programs must be completely rewritten to work with the capability programing model.

Table 1 summarizes the taxonomy of design issues common to DIFC systems. Laminar combines the strengths of PL- and OS-based systems. Laminar handles implicit flows and provides fine-grained information flow control just like PL systems but without resorting to whole-program static analysis. Like OS-based systems, Laminar can enforce security policies on system resources. Laminar makes it easier to deploy and use information flow control systems by introducing the intuitive notion of lexically scoped code blocks called *security regions*.

Integrating language and OS-based security. Hicks et al. observe that security-typed languages can be used to ensure that OS security policies are not violated by trusted system applications,

Issue	PL solution [22, 26]	OS solution [15, 29, 31]	Laminar solution
Main subsystems modified	Compiler and type system	(1) Complete OS [29, 31] (2) User-level reference monitor and kernel module [15]	VM and kernel module
Trusted computing base	Compiler, VM, and OS	OS	VM and OS
Securing individual application data structures	Whole-program static analysis	Either not supported or inefficient because of page table mechanisms	Dynamic analysis and VM enforcement using read/write barriers
Securing files and OS resources	Not handled	(1) Modify entire OS or (2) User-level reference monitor with kernel module support	Kernel module
Termination, timing, probabilistic channels	Not handled	HiStar [31] and Flume [15] handle termination channels by suppressing termination notification	Not handled
Implicit information flow	Static analysis	Not applicable—information flow tracked at granularity of thread [31] or address space [15]	Dynamic analysis using lexically scoped *security regions*
Deployment issues	Code must use new language or type system	Excludes multithreaded applications whose threads have heterogeneous security needs	Incrementally deployable

Table 1. Issues for DIFC systems. Laminar offers better functionality than the combination of PL and OS solutions.

such as `logrotate` [13]. Their framework, called SIESTA, extends Jif to enforce SELinux [18] MAC policies at the language level. The aims of Laminar and SIESTA are orthogonal. SIESTA provides developers with a mechanism to prove to the system that an application is trustworthy, whereas Laminar provides the developer a unified abstraction for specifying application security policies.

Termination, timing and probabilistic channels. Vachharajani et al. argue that implementing DIFC with dynamic checking is as correct as static checking by showing that the program termination channels of static and dynamic DIFC systems leak an arbitrary number of bits [28]. They prove that a correct dynamic DIFC system will over-approximate information flow, rejecting some programs that do not contain actual information flow violations. Laminar is a dynamic DIFC system and its security regions explicitly over-approximate information flow.

DIFC systems attempt to eliminate covert channels, which may be used to leak information, but do not eliminate timing channels [16] or probabilistic channels [24]. Section 4.3.3 defines and discusses implicit information flows.

3. DIFC model

All DIFC systems need a mechanism to denote the sensitivity of information and the privileges of the participating users. This section describes the mechanisms used by Laminar and the DIFC rules that determine safe information flows.

In DIFC systems, the security policy is defined in terms of *principals* that read and write the data in the system. Examples of principals in DIFC systems are users [22], processes [15] and kernel threads [31]. Principals in Laminar are kernel threads.

3.1 DIFC abstractions

Standard DIFC abstractions include tags, labels, and capabilities. Tags are short, arbitrary tokens drawn from a large universe of possible values ($\mathcal{T}$) [15]. A tag has no inherent meaning. A set of tags is called a label.

In a DIFC system, any principal can create a new tag for secrecy or integrity. For example, a web application might create one secrecy tag for its user database and a separate secrecy tag for each user's data. The secrecy tag on the user database will prevent authentication information from leaking to the network. The tags on user data will prevent a malicious user from writing another user's secret data to an untrusted network connection.

Principals assign labels to data objects. Data objects include program data structures (e.g., individual objects, arrays, lists, hash tables) and system resources (e.g., files and sockets). Previous OS-based systems limit principals to the granularity of a process or support threads by enforcing DIFC rules at the granularity of a page. Our system is the first to expand principals to support threads as principals and enforce DIFC at object granularity.

Each data object or principal x has two labels, S_x for secrecy and I_x for integrity. A tag t in the secrecy label S_x of a data object denotes that it may contain information private to principals with tag t. Similarly, a tag t in I_x implies that a data object may contain data *endorsed* by principals with integrity tag t. Data integrity is a guarantee that data exists in the same state as when it was endorsed by a principal. For example, if Microsoft endorses a data file, and the integrity of the file is preserved, then a user can choose to trust the file's contents if she trusts Microsoft. A principal's labels restrict the interaction that the principal can have with other principals and data objects.

A partial ordering of labels imposed by the subset relation forms a lattice [9]. At the bottom of the lattice are unlabeled resources, which have the empty label for security and integrity. An implicit empty label means that the program need not label every data structure, nor does the OS need to label every file in the file system. Allowing implicit empty labels makes Laminar easier to deploy incrementally.

A principal may change the label of a data object or principal if and only if it has the appropriate capabilities, which generalize ownership of tags [21]. A principal p has a capability set, C_p, that defines whether it has the privilege to add or remove a tag. For each tag t, let t^+ and t^- denote the capability to add and remove the tag t. The capability t^+ allows a principal to *classify* data with secrecy tag t, while the t^- capability allows it to *declassify* data. Classification raises data to a higher secrecy level; declassification lowers its secrecy level. Principals can add t to their secrecy label if they have the t^+ capability. If the principal adds t, then we call it *tainted* with the tag t. A principal taints itself when it wants to read secret data. To communicate with unlabeled devices and files, a tainted principal must use the t^- capability to untaint itself and to declassify the data it wants to write. Note that DIFC capabilities are not pointers with access control information, which is how they are commonly defined in capability-based operating systems [17, 25].

DIFC handles integrity similarly to secrecy. The t^+ capability allows a principal to endorse data with integrity tag t, and the t^- capability allows it to drop the endorsement. A principal with integrity tag t is claiming to represent a certain level of integrity. For example, code and data signed by a software vendor could run with that vendor's integrity tag. When the principal drops an integrity tag, for example, to read an unlabeled file of lower integrity, the principal drops the endorsement of the tag.

Note that the capability set C_p is defined on tags. A tag can be assigned to a secrecy or integrity label. In practice, a tag is rarely used for both purposes. C_p^- is the set of tags which principal p

may declassify (drop endorsements), and C_p^+ is the set of tags that p may classify (endorse). Principals and data objects have both a secrecy and integrity label; a data object with secrecy label s and integrity label i is written: $\{S(s), I(i)\}$. An empty label set is written: $\{S(), I()\}$. The capability set of a principal that can add both s and i but can drop only i is written: $\{C(s^+, i^+, i^-)\}$.

3.2 Restricting information flow

Programs implement policies to control access and propagation of data by using labels to limit the interaction among principals and data objects. Information flow is defined in terms of data moving from a source x to a destination y, at least one of which is a principal. For example, principal x writing to file y or sending a message to principal y is an information flow from x to y. If principal x reads from a file y, then we say information flows from source y to destination x. Laminar enforces the following information flow rules for x **to** y:

Secrecy rule. Bell and LaPadula [4] introduced the simple security property and the *-property for secrecy. These properties enforce that no principal may read data at a higher level (*no read up*) or write data to a lower level (*no write down*). Expressed formally, information flow from x to y preserves secrecy if:

$$S_x \subseteq S_y$$

Note that x or y may make a flow feasible by using their capabilities to explicitly drop or add a label. For example, x may make a flow feasible by removing a label L from S_x if it has the declassification capability for L, i.e. $L \in C_x^-$. Similarly, y may use its capabilities in C_y^+ to extend its secrecy label and receive information.

Integrity rule. The integrity rule constrains who can alter information [5] and restricts reads from lower integrity (*no read down*) and writes to higher integrity (*no write up*). In our system, we enforce the following rule:

$$I_y \subseteq I_x$$

Intuitively, the integrity label of x should be at least as strong as destination y. Just like the secrecy rule, x may make a flow feasible by endorsing information sent to a higher integrity destination, which is allowed if x has the appropriate capability in C_x^+. Similarly, y may need to reduce its integrity level, using C_y^-, to receive information from a lower integrity source.

Label changes. According to the previous two rules, a principal can enable information flow by using its current capabilities to drop or add a label. Laminar requires that the principal must *explicitly* change its current labels. Zeldovich et al. show that automatic, or implicit, label changes can form a covert storage channel [31].

In Laminar, a principal p may change its label from L_1 to L_2 if it has the capability to add tags present in L_2 but not in L_1, and can drop the tags that are in L_1 but not in L_2. Formally, this is stated as:

$$(L_2 - L_1) \subseteq C_p^+ \text{ and } (L_1 - L_2) \subseteq C_p^-$$

3.3 Calendar example

Again consider scheduling a meeting between Bob and Alice using a scheduling server that is not administered by either Alice or Bob. Alice's calendar file has a secrecy tag, a, and Bob's calendar file has a secrecy tag, b.

Ensuring secrecy. Focusing on Alice, she gives a^+ to the scheduling server to let it read her secret calendar file, which has label $\{S(a)\}$. A thread in the server uses the a^+ capability to start a security region with secrecy tag a that reads Alice's calendar file. Once the server's thread has the label $\{S(a)\}$, it can no longer return to the empty label because it lacks the declassification capability, a^-. As a result, the server thread can read Alice's secret file, but it can never write to an unlabeled device like the disk, network, or display. If the server thread creates a new file, it must have label $\{S(a)\}$, which is unreadable to its other threads. Before the server thread can communicate information derived from Alice's secret file to another thread, the other thread must add the a tag, and also becomes unable to write to unlabeled channels.

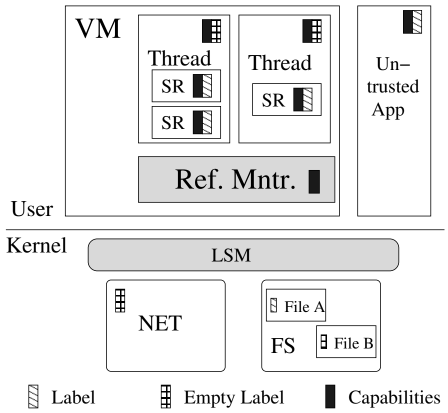

Figure 1. Design of Laminar. Unlabeled objects have an implicit empty label. Trusted components are shaded.

Ensuring integrity. The scheduling server runs its plugins with an integrity tag, i, corresponding to the idea of having an authority vouch for the safety and correctness of plugins, just as `addons.mozilla.org` vouches for plugins by allowing a secure network connection. The server cannot execute or read a plugin that has an integrity label lower than $\{I(i)\}$. The server is assured that plugins and their input files have not been written by any principal with an integrity label lower than $\{I(i)\}$. The DIFC integrity rule gives the server confidence that its code and data files have not been tampered with.

Sharing secrets with trusted partners. Alice and Bob collaborate to schedule a meeting while both retaining fine-grained control over what information is exposed. Both send the server a code module in a file that has integrity label $\{I(i)\}$. For example, they might have access to such an integrity label because they are both employed by the company running the scheduling server. Alice's module has access to both her a^+ and a^- capabilities, so the server calls her code which reads her secret calendar file and selectively declassifies parts of it, for instance, making her availability between 10:00am and 1:30pm on Mondays and Tuesdays publicly available (unlabeled). Alice controls which of her data flows into the scheduler. Bob does the same, and the scheduler can communicate with both of them their possible meeting times.

Discussion. In this example, Alice specifies a declassifier as a small code module that can be loaded into a larger server application, which can be completely ignorant of DIFC and requires no modifications to work with Alice's DIFC-aware module. For previous DIFC systems, this example would be more cumbersome. OS-based DIFC systems would require the declassifier to run as a separate process. Language-based systems would require the entire application to be annotated for DIFC enforcement. By integrating OS and language techniques, Laminar substantially improves the state of the art in DIFC.

4. Design

This section describes how Laminar enforces DIFC in an enhanced VM and OS. Figure 1 illustrates the Laminar architecture. The VM enforces DIFC rules within the application's address space. The OS security module mediates accesses to system resources. Only the VM and the OS are trusted in our model.

For example, Alice may write a program in Java using the Laminar API to label her data. Alice's program uses the same label namespace present in the file system: it can read data from a labeled file into a data structure with the same label. She compiles the code using a standard, untrusted, bytecode generator such as `javac`. The Laminar just-in-time (JIT) compiler and VM execute the bytecode, and the Laminar OS executes the Laminar VM. Laminar ensures that any accesses or modifications to labeled data follow the DIFC rules and occur in a *security region*, a lexically scoped region specified by the program. Security regions make it easier to secure and audit only the portions of a program that need it, making it easier to deploy Laminar incrementally. They also decrease the cost of dynamic security checks by the Laminar runtime.

4.1 Enforcement mechanism

The Laminar OS extends a standard operating system with a Laminar security module for information flow control. The Laminar OS security module governs information flows through all standard OS interfaces, including through devices, files, pipes and sockets. The OS regulates communication between threads of the same or different processes that access the labeled or unlabeled system resources or that use OS inter-process communication mechanisms, such as signals. OS enforcement applies to *all* applications, preventing unlabeled or non-Laminar applications from circumventing the DIFC restrictions.

The Laminar VM regulates information flow between heap objects and between threads of the same process via these objects. These flows are regulated by inserting dynamic DIFC checks in the application code. Because the Laminar VM regulates these flows within the address space, the OS allows data structures and threads to have heterogeneous labels. All threads in multithreaded processes without a trusted VM must have the same labels and capabilities.

4.2 Programming model

Laminar provides language extensions, a new security library, and new security-related system calls. The `secure` keyword is used to lexically scope a security region. Figure 2 depicts the library API, which includes tag creation, declassification, and label queries. The Laminar OS exports security system calls to the trusted VM for capability and label management, as shown in Figure 3. An untrusted application may directly use these system calls to manage its capabilities and labels.

Threads are the only principals in Laminar, but the thread's labels and capabilities are modified when entering and exiting security regions. In Laminar, labeled data objects (files, heap allocated objects etc.) can be accessed only inside security regions. Hence, outside a security region threads always have empty labels. The VM and the OS do not allow code outside the security region to access labeled data objects. During the execution of a security region, the VM gives the thread the labels and capabilities of the security region so that the OS can mediate access to system resources according to the security region's labels. Security regions are not visible to the OS, so the thread itself must have the labels and capabilities. At the end of the security region, the VM restores the thread's original capabilities and labels.

4.3 Security regions

A security region is a lexically scoped code block that has parameters for a capability set, a secrecy label, and an integrity label. The labels dictate which data the program may touch inside the security region. The capabilities dictate how a thread within a security region may add or remove labels. When we say a security region performs an action, we mean a kernel thread executing within the region performs the action.

Only code within a security region can access labeled data. Security regions demarcate the code regions that are security sensitive, easing the programmer's burden when adding security policies to existing programs. The programmer is just required to wrap the pieces of code that touch labeled data in a security region, such as a routine that reads a sensitive file into a data structure. Usually only a small portion of code and data in a program is security sensitive, so security regions also simplify the task of auditing security-sensitive code.

Requiring threads to access labeled data within security regions limits the amount of work the VM and compiler must do to enforce DIFC, provided that a substantial portion of the execution time is spent operating on unlabeled data. Every time the program reads or writes labeled objects or OS resources within a security region, the system must check the information flow with respect to the *current* labels of the thread executing within the region. For example, an assignment $w = r$ inside a security region R is safe if and only if the information flow from r **to** the thread inside R and from the thread **to** w is legal. Note that the Laminar library API (Figure 2) does not include a routine for adding labels to a thread. In order to add labels, a thread must start a security region.

Laminar Application Library API.

`Label getCurrentLabel(LabelType t)`
  Return the current secrecy or integrity label of the security region.
`Tag createAndAddCapability()`
  Create a new tag and add both capabilities to the current principal.
`void removeCapability(CapType c, Tag name, boolean global)`
  Drop the given capability from the current principal.
  Setting the global flag drops a capability permanently, whereas not setting it drops the capability for the scope of a security region.
`Object copyAndLabel(Object o, Label l)`
  Return a copy of the object o with new label l.

Figure 2. Laminar library API. `LabelType` denotes the secrecy or integrity label. `CapType` denotes the plus, minus or both capabilities for a given tag. The API also has wrapper functions(not shown) for the new system calls introduced in Laminar OS.

Laminar System Calls.

`tag_t alloc_tag(capList_t &caps)`
  Return a new tag, add the plus and minus capabilities to the calling principal, and write the new capabilities into *caps*.
`int set_task_label(tag_t l, int op, int type)`
  Set the *type* (secrecy or integrity) label of the current principal.
`int drop_label_tcb(pid_t tid)`
  Drop the current temporary labels of the thread without capability checks. Can be called only by threads with the special integrity tag.
`int drop_capabilities(capList_t *caps, int tmp)`
  Drop the given capabilities from the current principal. Tmp is a flag used to suspend a capability for a security region or during a `fork()`.
`int write_capability(capability_t cap, int fd)`
  Send a capability to another thread via a pipe.
`int create_file_labeled(char* name, mode_t m, struct label *l)`
  Create a labeled file with the given labels.
`int mkdir_labeled(char* name, mode_t m, struct label *l)`
  Create a labeled directory with the given labels.

Figure 3. Laminar system calls. The `tag_t` and `capability_t` types represent a single tag or capability, respectively. The `struct label` type represents a set of tags that compose a label, and the `capList_t` type is a list of capabilities.

Security regions provide several benefits. They make it easier for programmers to add security policies to existing programs. They also make it easier for programmers to audit security code, and limit the effects of implicit information flows (Section 4.3.3). Finally, they make the DIFC implementation more efficient by reducing the amount of code that requires full DIFC checks.

4.3.1 Example

Figure 4 depicts code where the calendar server reads a file belonging to Alice, adds an event to the common calendar, and exports the common meeting schedule for Bob. As shown, the data structure `cal` has the secrecy tags a and b. The thread entering the security region is initialized with the secrecy label $S(a, b)$ and therefore it can read secret data guarded by these tags. Its integrity label i restricts it from reading data that is not tagged with i. The thread has the capability $C(a^-)$ to declassify tag a. The line `L1` is a valid information flow because the label of the thread executing at `L1` is $\{S(a, b), I(i)\}$, which is more restrictive than $\underline{f}$. (We adhere to the convention that $\underline{a}$ is the label of a).

At line `L2`, the VM checks that the write to calendar c is legal. The write is legal because `cal` has the same secrecy label as the thread in the security region at that point. At line `L3`, the common meeting time is computed and stored in `s2`. Note that `s2` has the same labels as the security region. At line `L4`, a nested security region is started to declassify data. The copying and relabeling of `s2`, at `L5`, is legal because the thread has the a^- capability. Notice that if line `L5` were `copyAndLabel(s2, S(), I(i))`, it would result in a VM exception because the thread does not have the b^- capability. In this example, the OS checks the file operations in line `L1` and the VM checks the operations in line `L2`–`L5`.

4.3.2 Security region initialization

Laminar enforces certain rules when a thread enters a security region. Let S_R, I_R, and C_R be the security, integrity, and capability sets of a security region, R. Similarly, let S_P, I_P, and C_P be the sets associated with a kernel thread P, that enters and then leaves R. Laminar supports arbitrary *nesting* of security regions. P could, therefore, already be inside a security region when it enters R. When the thread enters the security region, the following rules hold:

$$S_R \subseteq (C_p^+ \cup S_P) \text{ and } I_R \subseteq (C_p^+ \cup I_p) \tag{1}$$

$$C_R \subseteq C_P \tag{2}$$

The first rule states that the creating principal P must have either the add capability or the label itself for any tag it will have within a security region.The second rule states that the principal P can only retain a subset of its current capabilities when it enters a security region.

The above rules encapsulate the common sense understanding that a parent principal, P, has control over the labels and capabilities it passes to a security region, and that the system will not let the principal create a security region with security properties that the principal itself lacks. The rules also state that security regions nest in the natural way based on the labels and capabilities of the thread entering the nested region.

4.3.3 Implicit information flows

A major benefit of security regions is that they limit the amount of analysis necessary to restrict implicit information flows. Implicit information flow leaks secret data through control flow decisions [10]. For example, the code in Figure 5 shows an implicit flow from the control variable H to the data variable L. By looking at the value of L, a thread can deduce the value of H. Since L is low secrecy and H is high secrecy, this implicit flow is a violation of DIFC rules. The system must therefore detect and prohibit this flow.

```
Calendar cal; // has labels {S(a,b), I(i)}
Output ret;   // has labels {S(b), I(i)}
File f;       // has labels {S(a), I(i)}

    secure ({S(a,b), I(i), C(a⁻)}) {
[L1] Schedule s1 = getScheduleFromFile(f);
[L2] caladdSchedule(s1);
[L3] Schedule s2 = cal.getCommonSchedule();
[L4] secure({S(b), I(i), C(a⁻)}){
[L5]  ret.val = Laminar.copyAndLabel(s2, S(b), I(i));
[L6] }
     ...
```

Figure 4. Example pseudocode to read and update a calendar. The thread (not shown) has the required capabilities a^+, a^-, b^+ and i^+ to initialize the security region.

```
// H has labels {S(h), I()}
// L has labels {S(), I()}
// Invariant: y == 2x
L = false;
secure ({S(h), I(), C()}) {
  x++;
  if (H) L = true;
  y = 2 * x;
  ...
} catch (...) {
  y = 2 * x;
}
```

Figure 5. Pseudocode that shows how implicit flows are handled with secure/catch.

```
// H has labels {S(h), I()}
// L has labels {S(), I()}
L = false;
secure ({S(h), I(), C()}) {
  if (H) while (true) { }
} catch (...) { }
```

Figure 6. Pseudocode that leaks data via a termination channel.

Laminar has a special construct to limit implicit flows—each security region has a required catch block as shown in Figure 5. The catch block executes with the same labels as the security region, and the capability set at the time of the exception. For instance, if H is true and the program attempts assignment to L, then Laminar raises an exception because the security region does not have the right to declassify H. The catch block gives the programmer a chance to restore program invariants before exiting the security region. No flow occurs between H and L because L is never assigned, regardless of whether H is true or false, and control flow continues after the secure/catch blocks. The VM suppresses all exceptions inside a security region that are not explicitly caught, including exceptions within a catch block. The VM continues execution after the security region.

In addition to restricting exception control flow, Laminar limits implicit flows by restricting how a security region returns from a region through non-exceptional control flow. In particular, the VM enforces that security regions must exit via fall through. Security regions cannot use `break`, `return`, or `continue` to exit, except in the trivial case where these expressions cause control flow to continue from the statement immediately after the security region ends.

Laminar thus eliminates implicit flows by hiding the control flow of a security region from code outside of the security region. In Figure 5, code outside of the security region cannot distinguish an execution where H is true from one where it is false. In contrast, DIFC systems that rely on static analysis prevent these flows by detecting them during compilation [21].

Both Laminar and static analysis DIFC systems assume that programs (or in Laminar's case, security regions) terminate. Figure 6 shows an example of an implicit flow via a *termination channel*, that leaks secret information based on whether the application terminates. If control returns from this security region, then unprivileged code can learn that H is false. Similarly, a colluding application might learn that H is true if the application must be explicitly killed. OS-based DIFC systems eliminate termination channels by ensuring that no one with whom communication is prohibited is notified of thread termination. In practice, termination channels have been low bandwidth and difficult to exploit, but in a multithreaded environment, exploitation might be significantly easier. For instance, if H is usually false, a low security thread may deduce that fact with high probability.

In cases where reestablishing a program invariant is difficult, a secure catch block can simply kill the process, for example, by calling `System.exit()`. If high-secrecy data structures become corrupted, programmers may want to terminate the program rather than require a declassifier to notice the corruption. However, exiting the program in the catch block creates a termination channel. A more restrictive model would prevent this termination channel by ensuring that only a security region with full declassification capabilities kills the process.

4.4 VM-OS interface

Security regions are abstractions that are not visible to the OS. For the OS to enforce DIFC rules on system calls made in a security region, the VM must set appropriate labels on the current kernel thread using the `set_task_label` system call. As an optimization, the VM omits setting the labels in the kernel thread if the security region does not perform a system call. When the VM sets the labels on a thread, the OS checks to ensure that the labels are legal given the thread's capabilities.

Acquiring tags and capabilities. Principals in Laminar acquire capabilities in three ways: they allocate a new tag, they inherit them through `fork()`, or they perform inter-process communication. The system carefully mediates capability acquisition, lest a principal incorrectly declassify or endorse data.

A principal can allocate a new tag via the `alloc_tag` system call. The OS security module that allocates tags is trusted and ensures that all tags are unique. The principal that allocates a tag becomes the owner of the new tag. The owner can give the plus and minus capabilities for the new tag to any other principal with whom it can legally communicate. By default, a thread that gains a capability within a security region retains the capability on exit from the region. The thread must explicitly call `drop_capability` to prevent the capability from propagating to the calling context.

Threads and security regions form a natural hierarchy of principals. When a kernel thread forks off a new thread, it can initialize the new thread with a subset of its capabilities. Similarly, when a thread enters a security region, the thread retains only the subset of its capabilities specified by the region. In general, when a new principal is created, its capabilities are a subset of its immediate parent, which the VM and OS enforce.

The passing of all inter-thread and inter-process capabilities is mediated by the kernel, specifically with the `write_capability` kernel call. This system call checks that the labels of the sender and receiver allow communication.

Removing tags and capabilities. The Laminar VM is responsible for correctly setting thread labels and capabilities inside security regions. When a thread enters a security region, the VM first makes sure that the thread has sufficient capabilities to enter the region. If it does, the VM sets the labels and capabilities of the thread to equal those specified by the security region. Similarly, when the thread exits the security region, the VM restores the labels and capabilities it had just before it entered the region. On exiting a nested security region, the VM restores the labels and capabilities of the thread to those of the parent security region.

The Laminar language API provides a method, `removeCapability`, that removes a thread's capability in the VM, which then calls the `drop_capability` system call to notify the kernel. The `removeCapability` method takes an argument, `global`, that allows the user to specify whether the capability should be dropped only for the scope of the security region or permanently (i.e., globally). Globally dropped capabilities are not restored to the parent when the thread exits a security region. The VM uses the `set_task_label` system call to change the label of a thread at the beginning and end of a security region. This function has no user API; it is called solely by the VM at the entry and exit of security regions. Laminar does not allow security regions to change their labels, because the VM relies on labels staying the same throughout lexically scoped regions to prevent leaks through local variables, as discussed in Section 5.1. To change labels in the middle of a security region, a thread may begin a nested security region.

Consider an example when a thread only has the a^+ capability and starts a security region with secrecy label $\{S(a)\}$. The Laminar VM will set the label of the thread as $\{S(a)\}$ when the security region begins. When the security region ends, the thread must drop the secrecy label, even if it does not have the a^- capability. To allow the thread to drop $\{S(a)\}$, the VM contains a thread, that is trusted by the OS, which runs code with a special integrity tag called `tcb`. Using the `drop_label_tcb` system call, this trusted thread may drop all current labels for a thread without having the appropriate capabilities.

A single, high-integrity thread in the VM limits exposure to bugs because the OS enforces that only the thread with the `tcb` tag may drop labels within a single address space. The VM cannot drop the labels on other applications. Only a small, auditable portion of the VM is trusted to run with this special label.

Capability persistence and revocation. Capability persistence and revocation are always issues for capability-based systems, and Laminar does not innovate any solutions. However, its use of capabilities is simple and stylized. The OS stores the persistent capabilities for each user in a file. On login, the OS gives the login shell all of the user's persistent capabilities, just as it gives the shell access to the controlling terminal. If a user wishes to revoke access to a resource for which she has already shared a capability, she must allocate a new capability and relabel the data. Because tags are drawn from a 64-bit address space, tag exhaustion is not a concern.

4.5 Labeling data

Data objects are labeled as part of their allocation to avoid races between creation and labeling. The VM labels objects allocated within a security region with the label of that region. The `create_labeled` and `mkdir_labeled` kernel calls create labeled files and directories. Other system resources use the label of their creating thread.

Like most other DIFC systems, Laminar uses immutable labels. To change a label, the user must copy the data object. Dynamic relabeling in a multithreaded environment requires additional synchronization to ensure that a label check on a data-object and its subsequent use by principal A are atomic with respect to the relabel by principal B. Without atomicity, an information flow rule may be violated. For example, A checks the label, B changes the label to be more secret, B writes secret data, and then A uses the data. Atomic relabeling can prevent this unauthorized flow from B to A. Instead, Laminar uses immutable labels to avoid extra synchronization.

4.6 Compatibility challenges

Although Laminar is designed to be incrementally deployed, some implementation techniques are incompatible with any DIFC system. For instance, a library might memoize results without regard for labels. If a function memoized its result in a security region with one label, a later call with a different label may attempt to return the memoized value. Because the memoized result is secret, the attempt to return it will be prevented by the system. Such code must be modified to work in any DIFC system.

4.7 Trusted computing base

To implement Laminar, we added approximately $2,000$ lines of code to Jikes RVM,[1] a $1,000$ line Linux security module, and 500 lines of modifications to the Linux kernel. This relatively small amount of code means that Laminar can be easily audited.

We rely on the standardization of the VM and the OS as the basis of Laminar's trust. In addition to trusting the base VM, Laminar requires that the VM correctly inserts the appropriate read and write instrumentations called barriers for all accesses and optimizes them correctly. Read and write barrier insertion is localized and standard in many VMs. In Linux, Laminar assumes that the kernel has the proper hooks to call into Linux security modules (LSM). Because many projects rely on LSMs, the Linux code base is under constant audit to make sure all necessary calls are made.

5. Implementation

This section describes our implementation of Laminar, which modifies Jikes RVM and the Linux operating system to provide DIFC.

5.1 JVM support

We implement Laminar's trusted VM in Jikes RVM 3.0.0,[1] a high-performance Java-in-Java virtual machine [2]. Our implementation of Laminar is publicly available on the Jikes RVM Research Archive.[2] As of August 2008, Jikes RVM's performance compared well with commercial VMs: the same average performance as Sun HotSpot 1.5; and 15–20% worse than Sun HotSpot 1.6, JRockit, and J9 1.9[3].All subsequent uses of the term JVM refer to the Laminar-enhanced version of Jikes RVM.

The JVM controls information flow by ensuring that all program access to labeled data occur in security regions. The JVM adds instrumentation called *barriers* at every object read and write; these barriers check at run time that accesses conform to the DIFC rules in Section 3.

Starting a security region. When a thread starts a security region, the JVM checks whether it has the capabilities to initialize the security region with the specified labels and capabilities, as described in Section 4.3.2. Thread capabilities are stored in the kernel. The JVM then caches a copy of the current capabilities of each thread to make the checks efficient inside the security region.

Restricting information flow for locals and statics. The JVM enforces information flow control for accesses to three types of application data: locals, which reside on the stack and in registers; objects, which reside in the heap; and statics, which reside in a global table.

Because the lifetime of local variables is typically short, and tracking their labels would be expensive, our prototype restricts the programming model. Laminar *statically* (during JIT compilation) enforces the following restrictions on local variables: (1) a local variable written inside a security region may not later be read outside that security region if the region has secrecy labels, and (2) a local variable already written outside a security region may not be read inside the region if the region has integrity labels. Because a security region's labels are dynamic, for simplicity our implementation requires both properties for every security region. An important exception to these restrictions is that local variables that are references may be *dereferenced*; however the references themselves may not be read or written (e.g., `obj.f` is allowed, but `if (obj == null)` is not). These accesses to the referenced object is checked by read and write barriers, which are discussed below.

The Laminar prototype implementation requires that a security region be in its own method to simplify static checking of these restrictions. Thus, the JVM only needs to ensure statically that any method security region (1) does not return a value if the region has security labels, (2) takes only reference-type parameters, and (3) does not read or write the values of its parameters (dereferencing its parameters is allowed). However, our prototype implementation does not currently check these rules but instead requires programs to adhere to them. A production implementation of Laminar could decouple security regions from methods by enforcing local variable restrictions as part of bytecode verification.

The JVM restricts information flow to and from static variables. The Laminar prototype implementation prevents security regions with secrecy labels from writing static variables, and prevents regions with integrity labels from reading statics. Compiler-inserted barriers at each access inside a security region enforce these restrictions. A production implementation could support labeling statics with modest overhead because static accesses are relatively infrequent compared to field and array element accesses. The applications in Section 7 do not need labeled static variables.

Supporting information flow for objects. The JVM tracks information flow for objects that live in the heap. At allocation time, objects may be assigned secrecy and integrity labels that are immutable. By default, objects allocated inside security regions are assigned the labels of the region at the allocation point. The program may specify alternate labels, as long as they conform to DIFC rules. To change an object's labels, our implementation provides an API call, `copyAndLabel`, that clones an object with specified labels. The label change must conform to the label change rule (Section 3). The JVM allocates labeled objects into a separate *labeled object space* in the heap, allowing instrumentation to quickly check whether an object is labeled. We modify the allocator to add two words to each object's header, which point to secrecy and integrity labels.

Our implementation encapsulates labels into immutable, opaque objects of type `Labels` that support operations such as `isSubsetOf()` and `union()`. For efficiency, `Labels` objects may be shared by objects, security regions, and threads because they are immutable; mutating operations such as `union()` return a new object, if needed. Internally, `Labels` uses a sorted array of 64-bit integers to hold tags. Because `Labels` is opaque, applications cannot observe the individual values of the tags, so they can read and use labels without creating a covert channel.

The JVM's compiler inserts read and write *barriers* [8] (instrumentation at every operation) into application code to enforce DIFC rules. Inside security regions, the compiler inserts barriers at *labeled object allocation* (before the constructor call) to set the labels and check that they conform to DIFC rules. It inserts barriers at every *read from* and *write to* an object field or array element. Inside security regions, barriers load the accessed objects' secrecy and integrity `Labels` and check that they conform to the current secu-

[1] http://www.jikesrvm.org

[2] http://www.jikesrvm.org/Research+Archive

[3] http://jikesrvm.anu.edu.au/ dacapo/

```
  // credentials = {S(s1,s2),I(),C(s1-,s2-)}
  // credentialsNew = {S(),I(),C(s1-,s2-)}
  // newLabel  = {S(),I()}
[L1] secure(credentials){
[L2]  int m1 = student1.marks;
[L3]  int m2 = student2.marks;
[L4]  MyObject obj = new MyObject(m1+m2);
[L5]  secure(credentialsNew){
[L6]   ret.val = Laminar.copyAndLabel(obj, newLabel);
       ...  }
```

Figure 7. Example code to read the marks of two students. The `student1` and `student2` objects are labeled. The object `credentials` contains the secrecy, integrity, and capabilities sets with which the security region is initialized.

rity region's labels and capabilities. Outside security regions, read and write barriers check that the accessed objects are *unlabeled* (or equivalently, have the empty label). The compiler inserts barriers inside security regions at *static* accesses to verify that static reads (writes) occur only in regions without integrity (secrecy) labels.

The compiler inserts different barriers at an access depending on whether the access occurs inside or outside a security region. Choosing the right barrier at compile time can be difficult because a method may be called by code inside of and outside of a security region. In our prototype implementation, when a method first executes and the compiler compiles it, the compiler checks whether the thread is in a security region and inserts barriers accordingly. (Subsequent recompilation at higher optimization levels reuses this decision.) This approach, which we call *static barriers*, fails if a method is called from both within and without a security region. Thus, we also support a configuration where the compiler adds *dynamic barriers* that check whether the current thread is in a security region or not, and then execute the correct barrier. A production implementation would use cloning to compile two versions of methods executed from both contexts; the same approach is used in prior work on software transactional memory [23]. Static barriers add the same overhead that cloning would achieve.

Because object labels are immutable and security regions cannot change their labels, repeated barriers and checks on the same object are redundant. We implement an intraprocedural, flow-sensitive data-flow analysis that identifies redundant barriers and removes them. A read (or write) barrier is redundant if the object has been read (written), or if the object was allocated, along every incoming path. Although the optimization is intraprocedural, the compiler already inlines small and hot methods, increasing the scope of redundancy elimination.

Example. The example illustrated in Figure 7 computes the sum of the marks obtained by two different students. The `student1` and `student2` objects are labeled and have different secrecy values associated with them. The object `credentials` contains a set of labels and capabilities. If the capabilities and labels inside `credentials` do not conform to the current thread's capabilities, then the program terminates at `L1`. Once the security region starts, the thread's current labels become those present in `credentials`. Lines `L2` and `L3` are reads of labeled objects that will result in an error if the flow from `student1.marks` or `student2.marks` to the thread in the security region is not allowed. In line `L4`, the JVM allocates object `obj` with the labels present in `credentials`. The programmer can write to the reference `obj` because it is used only inside the security region. In line `L6`, the thread attempts to change the labels of the object inside a nested security region. The JVM allows this change because the thread has the required capabilities. Note that the inner security region has empty secrecy label.

5.2 OS support

We have implemented support for DIFC in Linux version 2.6.22.6 as a Linux Security Module (LSM) [30]. LSM provides hooks into the kernel to allow custom authorization rules. We also added a set of system calls to manage labels and capabilities (Figure 2). Some LSM-based systems, such as SELinux [18], manage access control settings through a custom filesystem similar to `proc`. This method is isomorphic to adding new system calls. The Laminar security module contains about 1,000 lines of code, and about 500 lines of modifications to the kernel to support the new system calls.

Tags, labels, and capabilities. Tags are represented by 64-bit integers and allocated via the `alloc_tag()` system call. Labels and capabilities are stored in the opaque security field of the appropriate Linux objects (`task_struct`, `inode`, `file`, etc.). Secrecy and integrity labels for files are persistently stored in the file's extended attributes. Most of the standard local filesystems for Linux support extended attributes, including `ext2`, `ext3`, `xfs`, and `reiserfs`.

Files. Using LSM, Laminar intercepts `inode` and `file` accesses, which are used to perform operations on unopened files and file handles (including sockets and pipes), respectively. The Laminar security hooks perform a straightforward check of the rules listed in Section 3.2. The label of an `inode` protects its contents and its metadata, except for the name and label, which are protected by the label of the parent directory.

In a typical filesystem tree, secrecy increases from the root to the leaves. Creating labeled files in a DIFC system is tricky because it involves writing a new entry in a parent directory, which can disclose secret information. For example, we disallow a principal with secrecy label $\{S(a)\}$ from creating a file with secrecy label $\{S(a)\}$ in an unlabeled directory, because it can leak information through the file name. Instead, the principal should pre-create the file before tainting itself with the secrecy label.

More formally, we allow a principal with non-empty labels $\{S_p, I_p\}$ to create a labeled file or directory with labels $\{S_f, I_f\}$ if: (1) $S_p \subset S_f$ and $I_f \subseteq I_p$; (2) the principal has capabilities to acquire labels $\{S_p, I_p\}$; and (3) the principal can write to the parent directory with its current label. This approach prevents information leaks during file creation while maintaining a usable interface.

Applying integrity labels to a filesystem tree is more complex than secrecy. The intuitive reason for integrity labels on directories is to prevent an attacker from tricking a program into opening the wrong file, for instance using symbolic links. The practical difficulty with integrity for directories is that a task with integrity label I_A cannot read any files or directories without this label, potentially including `/`. If system directories, such as `/home`, have the union of all integrity labels, then an administrator cannot add home directories for new users without being given the integrity labels of all existing users. Flume solves this problem by providing a flat namespace that applications can use to store data with integrity labels.

Applying integrity labels to a traditional Unix directory structure brings out a fundamental design tension in DIFC OS's, between usability and minimizing trust in the administrator. Laminar finds a middle ground by labeling system directories (e.g., `/`, `/etc`, `/home`) with a system administrator integrity label when the system is installed. A user may choose to trust the system administrator's integrity label and read absolute paths to files, or she may eschew trust in the system administrator by exclusively opening relative paths. In the worst case, she creates her own `chroot` environment. Simple relative paths were sufficient for all of the case studies in this paper. Laminar's approach supports incremental deployability by allowing users to choose whether to trust the system administrator at the cost of extra work for stronger integrity guarantees.

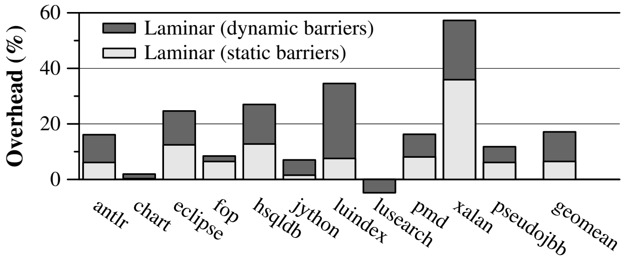

Figure 8. Laminar VM overhead on programs without security regions.

Benchmark	Linux	Laminar	% Overhead
stat	0.92	0.94	2.0
fork	96.40	97.00	0.6
exec	300.00	302.00	0.6
0k file create	6.29	6.56	4.0
0k file delete	2.54	2.68	6.0
mmap latency	6,877.00	7,035.00	2.0
prot fault	0.24	0.26	7.0
null I/O	0.13	0.17	31.0

Table 2. Execution time in microseconds of several lmbench OS microbenchmarks, and overhead incurred by using Laminar. Lower is better.

Pipes. Laminar mediates inter-process communication (IPC) over pipes by labeling the inode associated with the pipe message buffer. A process may read or write to a pipe so long as its labels are compatible with the label of the pipe. Message delivery over a pipe in Laminar is unreliable. An error code due to an incorrect label or a full pipe buffer can leak information, so messages that cannot be delivered are silently dropped. Unreliable pipes are common in OS DIFC implementations [15, 29].

Reads from a pipe in Laminar must be nonblocking to prevent illegal information flow. Standard pipes deliver an `EOF` to readers when a writer exits. When the exiting process does not have appropriate write labels, sending an `EOF` violates DIFC rules. Thus, reads should be non-blocking and readers cannot depend on an explicit `EOF` if the writer can change labels. In the common case where all applications in a pipeline have the same label, traditional Unix pipe behavior can be approximated with a timeout. Using pipes in programs with heterogeneous, dynamic labels may require modification for a DIFC environment.

6. Laminar overhead

This section reports the overhead of Laminar's subsystems. The performance loss on Java benchmarks without security regions is 6% or 17%, depending on whether the dynamic compiler compiles separate versions of methods called both from inside and outside security regions. The Laminar OS incurs an overhead of less than 8% on `lmbench`. All experiments, including those in the next section, were conducted on a machine with a quad-core Intel Xeon 2.83 GHz processor. All experiments configure Jikes RVM to run on two cores. All results are normalized to values obtained on unmodified Linux 2.6.22 and Jikes RVM 3.0.0.

6.1 JVM overhead

Figure 8 shows the overhead of our Laminar-enabled JVM for the DaCapo benchmarks [7] and a fixed-workload version of SPECjbb2000 called `pseudojbb` [27]. Each experiment executes two iterations of the benchmark: the first includes compilation, and the second disables compilation and runs only the application. We report the running time of the second iteration. Because compilation decisions are nondeterministic, running times vary, so we execute 10 trials of each experiment and take the median.

The darker bar shows the overhead of dynamic barriers, which check dynamically if they are in a security region. Dynamic barriers add 17% overhead on average. The lighter bar is the overhead of using static barriers, 6% on average. As discussed in Section 5.1, a mature implementation of Laminar would use method cloning and eliminate all dynamic barriers. Because method cloning has comparable overheads to static barriers, code outside of a security region is expected to have an average overhead of 6%.

We also measure compilation time and find that, on average, static barriers double it, and dynamic barriers triple it. However, compilation time is not our primary concern, especially for long-running programs. For these benchmarks, compilation time accounts for just 8-12% of running time on average, making barrier compilation's effect on run time comparable to barriers' effect on application execution time. The overhead is high in large part because we instruct the compiler to inline the barriers aggressively, which bloats the code and slows downstream optimizations. To lower compilation time without increasing run time substantially, an implementation could choose to inline less aggressively.

6.2 OS overhead

We use the lmbench [19] suite of benchmarks to measure the overheads imposed on unlabeled applications when running on Laminar OS. A selection of the results is presented in Table 2.

In general, the overhead of the Laminar OS modifications are less than 8%, which is similar to previously reported overheads for Linux security modules [30]. The only performance outlier is the "null I/O" benchmark, which has an overhead of 31%. This benchmark represents the worst case for Laminar in that the system call being measured does little work to amortize the cost of the label check. For the sake of comparison, Flume adds a factor of 4-35× to the latency of system calls relative to unmodified Linux [15].

7. Application case studies

This section describes four case study applications and how we retrofit them with DIFC security policies. Table 3 summarizes the details of the applications. Figure 9 shows the overhead of running the modified version with Laminar. The retrofitted applications implement more powerful security policies than their unmodified counterparts, and all modifications are at most 10% of the source.

The figure breaks down the overhead of Laminar into four parts. *Start/end SR* is the overhead of application modifications to support DIFC, including the starting and ending of security regions and other security operations, such as `copyAndLabel`. The *Alloc barriers* configuration denotes the extra time for allocating labeled objects and assigning their label sets. *Static barriers* is the overhead from read and write barriers when the security context is known at compile time. Finally, *Dynamic barriers* is the extra overhead from barriers that check context at run time. We note that Gradesheet and Battleship run correctly with static barriers, but Calendar and FreeCS require dynamic barriers because some methods are called from both inside and outside security regions. Method cloning would obviate the need for dynamic barriers (Section 5.1).

In all our experiments, we disabled the GUI, as well as other I/O and network-related operations so that the Laminar overheads are not masked by them. Hence, the slowdown in deployed applications would be less than what is reported in our experiments. For comparison, Flume [15] adds 34–43% slowdown on the MoinMoin wiki application. Flume labels data at the granularity of an address space, and cannot enforce DIFC rules on heterogeneously labeled objects in the same address space.

7.1 GradeSheet

GradeSheet is a small program that manages the grades of students [6]. It uses three types of principals: professors, TAs and stu-

Application	LOC	Protected Data	LOC Added	% time in SRs
GradeSheet	900	Student grades	92 (10%)	6%
Battleship	1,700	Ship locations	95 (6%)	54%
Calendar	6,200	Schedules	290 (5%)	1%
FreeCS	22,000	Membership properties	1,200 (6%)	<1%

Table 3. Details of the various applications, including lines of code, the data that needs to be secured, the lines of code that had to be added to secure the application using Laminar and the fraction of time spent in security regions.

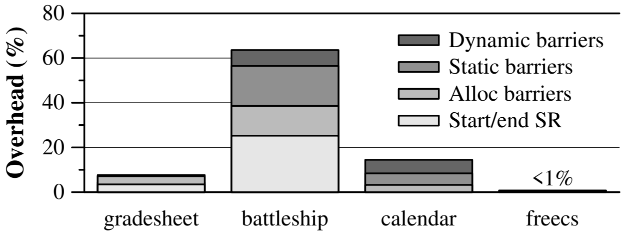

Figure 9. Overhead of executing applications retrofitted with Laminar.

Name	Security Set
GradeCell(i,j)	S={s_i}, I={p_j}
Student(i)	C={s_i^+, s_i^-}
TA(j)	C={$\bigcup_{i=1}^{i=n} s_i^+, p_j^+, p_j^-$}
Professor	C={$\bigcup_{i=1,j=1}^{i=n,j=m} (s_i^+, s_i^-, p_j^+, p_j^-)$}

Table 4. The security sets associated with the principals and data objects in GradeSheet. S, I and C stand for security, integrity and capability sets. Student(i) and TA(j) refer to the i^{th} student and j^{th} teaching assistant, respectively.

dents. The main data structure is a two-dimensional object array `GradeCell`. The $(i, j)^{th}$ object of `GradeCell` stores the information about student i and her marks in project j. A sample policy states that (1) the professor can read/write any cell, (2) the TA can read the marks of all students but only modify the ones related to the project that she graded, and (3) students can only view their own marks, but for any project.

Table 4 shows how this policy can be expressed by assigning labels and capabilities to the data and the principals respectively. Specifically, we guard the $(i, j)^{th}$ entry in the `GradeCell` with the secrecy tag s_i and the integrity tag p_j. Each student i has the capability to add or remove s_i, so students can read their own marks in any project. Each TA j has the capability to add tags s_i and the integrity tag for the project that she graded (p_j). This tag ensures that TAs can read the marks of all students, but the integrity constraint prevents them from modifying grades for projects that they did not grade.

Interestingly, Laminar found an information leak in the original policy. The policy allowed a student to calculate and read the average marks in a project, which leaks information about the marks of other students. After integration with Laminar, only the professor is allowed to calculate the average and declassify it.

Our experiments measure the time taken by the server to process queries from different users. The Laminar-enabled version has a 7% slowdown compared to the unmodified version.

7.2 Battleship

Battleship is a common board game played between two players. Each player secretly places her ships on the grid in her board. Play proceeds in rounds; in each round, a player shoots a location on the opponent's grid. The player who first sinks all the opponent's ships wins the game.

We started with `JavaBattle`, which is a 1,700-line Battleship program available on SourceForge. Each player P_i allocates a tag p_i and labels her board and the ships with it. The capability p_i^- is not given to anyone else, ensuring that only the player can declassify the locations of her ships. In the original implementation, players directly inspect the coordinates of a shot to determine whether it hit or missed an opponent's boat. Under Laminar, each player sends her guess to her opponent, who then updates his board inside a security region. The opponent then declassifies whether the guess was a hit or a miss and sends that information back to the first player. We had to add less than 100 lines of code to secure the program to run with Laminar.

In our experiments, the game is played between computers on a 15×15 grid without a GUI. Figure 9 shows that the secured version adds 56% overhead with static barriers. The overhead is high because the benchmark spends almost 54% of its time inside security regions. In a deployed Battleship, which would display the intermediate state of the board to the players, the overhead would be significantly less. In an experiment where we display the shot location after each move, the run time increases, and Laminar overhead drops to 1%.

7.3 Calendar

Like in the examples from earlier in the paper, we modified the `k5nCal`[4] multithreaded desktop calendar to label all data structures and `.ics` files that store a user's calendar information with the user's secrecy tag.All functions that access this data are wrapped inside security regions, including a scheduler that finds available meeting times for multiple users. In the original program, a user could view the calendar of other users, a feature we disabled.

Our experiments measure the time to schedule a meeting, which includes reading the labeled calendars of Bob and Alice, finding a common meeting date, and then writing the date to another labeled file that Alice can read. The scheduling code is executed in a thread that has the capability to read data for both Alice and Bob, but can only declassify Bob's data. The output file is protected by the label of Alice. Our experiment schedules 1,000 meetings. Figure 9 shows that the secured version of Calendar runs 14% slower than unmodified Calendar.

We note that for Calendar, idle time was high when running the DIFC version with two cores. We have not yet diagnosed this issue. However, we found that the problem is specific to our Xeon machine, so we report results for Calendar on a Core 2 Quad 2.4 GHz processor running our modified kernel.

7.4 FreeCS chat server

FreeCS is an open-source chat server written in Java[5]. Multiple users connect to the server and communicate with each other. FreeCS supports 47 commands, such as creating groups, inviting other users, and changing the theme of the chat room. The original security policy consists of an authorization framework that restricts what commands can be used by a user. All these policies are written in the form of `if..then` checks. These authorization checks are actually checks on the *role* of a user. For example, a user who is in the role of a `VIP` and has superuser power on a group can `ban` another user in the group.

We improve the security code in FreeCS by labeling sensitive data structures and accessing them inside security regions. We made most of our modifications in two classes—`Group` and `User`.

[4] http://k5ndesktopcal.sourceforge.net

[5] http://freecs.sourceforge.net

With Laminar, we localized all security checks to these classes. The abstraction of a role maps naturally onto integrity labels. For example, we protect the `banList` data structure with two tags, one that corresponds to the notion of VIP and the other for the group's superuser. Now, only users who have the add capability for these two tags can use the `ban` command. We also changed the authentication module to ensure that users are given the right capabilities when they log in. Our experiments measure the time to process requests from $4,000$ users, each invoking three different commands. Laminar's overhead is less than 1% (Figure 9).

7.5 Summary

The four case studies reveal a pattern in the way applications are written. First, most applications have only a few key data structures that need to be secured, like the array of student grades in GradeSheet or the playing boards in Battleship. Second, the interface to access these data structures is quite narrow. For example, `InternalServer` in GradeSheet and `DataFile` in Calendar contain the functions used to access the important data. These observations support our hypothesis that only localized changes are needed to retrofit DIFC onto many types of applications. Third, most of the data structures require heterogeneous labeling—the single data structure `GradeCell` has different labels corresponding to different students. Heterogeneous labeling is impractical in OS-based systems [15, 29, 31], since they support a single label on the whole address space or require the programmer to map application data structures onto labeled pages. The Laminar VM easily solves this problem with fine-grain tracking of labels on the data structure, for example, individual array elements and objects in GradeSheet.

8. Conclusion

Laminar is the first DIFC system to unify PL and OS mechanisms for information flow control. It provides a natural programming model to retrofit powerful and auditable security policies onto existing, complex, multithreaded programs.

Acknowledgments

We thank Eddie Kohler, Maxwell Krohn, and Nickolai Zeldovich for their feedback on earlier drafts of this paper. We also thank the anonymous referees for their constructive comments.

References

[1] *DaCapo Benchmark Regression Tests*. `http://jikesrvm.anu.edu.au/~dacapo/`.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, Susan Flynn Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, pages 333–348, 2008.

[4] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[5] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.

[6] A. Birgisson, M. Dhawan, Úlfar Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *CCS*, 2008.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[8] Stephen M. Blackburn and Antony L. Hosking. Barriers: Friend or foe? In *ACM International Symposium on Memory Management*, pages 143–151, 2004.

[9] D. E. Denning. A lattice model of secure information flow. *CACM*, 19(5):236–243, May 1976.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *CACM*, 20(7):504–513, July 1977.

[11] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD (The Orange Book) edition, December 1985.

[12] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 367–384, 2008.

[13] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. From trusted to secure: Building and executing applications that enforce system security. pages 205–218, 2007.

[14] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Trans. Softw. Eng.*, 17(11), 1991.

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

[16] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[17] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[18] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX*, 2001.

[19] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Usenix*, 1996.

[20] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[21] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, pages 129–142, October 1997.

[22] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. Software release. http://www.cs.cornell.edu/jif, July 2001.

[23] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA*, pages 195–212, 2008.

[24] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21, 2003.

[25] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *SOSP*, 1999.

[26] V. Simonet and I. Rocquencourt. Flow Caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, pages 152–165, 2003.

[27] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.

[28] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *MICRO*, 2004.

[29] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[30] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. K. Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symposium*, 2002.

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

[32] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI*, 2008.