Laminar: Practical Fine-Grained Decentralized Information Flow Control

Indrajit Roy Michael D. Bond Donald Porter Kathryn S. McKinley Emmett Witchel
Department of Computer Science
UT-Austin
{indrajit, mikebond, porterde, mckinley, witchel}@cs.utexas.edu

Abstract

This paper describes Laminar, the first system to implement decentralized information flow control (DIFC) 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 accessing this labeled data in lexically scoped *security regions*. Laminar enforces the security policies specified by the labels at run time. 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 policies changes less than 10% of the code, and incurs performance overheads from 4% to 84%. Whereas prior systems only supported limited types of multithreaded programs, Laminar supports a more general class of multithreaded DIFC programs that can access heterogeneously labeled data.

1. Introduction

As computer systems support more aspects of modern life, from finance to health care, there is an increasing need for security. 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, like 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 plugin. 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 it is 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 automatically enforced at all layers of the software stack.

The decentralized information flow control (DIFC) security model [20] 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 can enforce the more powerful policy that allows an application to read a file, but disallows broadcasting the contents of that file over an unsecured network channel. DIFC tracks information flow throughout the system, automatically restricting access based on simple information flow rules provided by the programmer.

DIFC systems provide security by allowing users to associate secrecy and integrity labels with data and regulating 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. To understand how DIFC works, 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. A thread *taints* itself with a secrecy label, and a secrecy label can have many categories of taint. Once a thread has a secrecy label, it 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 can declassify the result. The declassification function checks that its output contains no secret information; e.g., that 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 flows to any principal that accesses 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 [20, 21], in the operating system [14, 28, 30], or in the architecture [27, 31] Each approach has strengths and limitations. Language-based DIFC systems have relied on extensive type system changes, requiring the wrapping of standard libraries, augmentation of types throughout the entire program, and modification of the program's structure. OS-based DIFC systems have trouble enforcing data flow through program data structures because they support only a single label for an application's address space. 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, like files and sockets. By combining the strengths of PL and OS techniques, Laminar makes it possible to express a comprehensive security policy. This policy is then enforced using a combination of a virtual machine (VM) and the operating system.

Laminar is incrementally deployable—developers need only modify the security-sensitive portion of their programs. Trusted and untrusted threads, labeled and unlabeled files, sockets, data structures, etc. coexist in the system. In contrast, existing DIFC languages require pervasive program modification to label data structures, variables, functions, and return types [19, 25]. They also exclude features, such as dynamic class loading and multithreading, because they rely on whole-program static analysis.

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 in Laminar. 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 an unified PL and OS mechanisms for enforcing DIFC, which 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 (a Java virtual machine (JVM)), and the Linux operating system.
4. Four case studies that retrofit security policies onto existing code. These case studies require less than 10% modifications to the overall code base and incur overheads from Laminar ranging from 4% to 84%.

These advantages and first results suggest that integrating PL and OS support is a promising direction for supporting the security needs of modern application.

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. Table 1 summarizes the taxonomy of design issues common to DIFC systems. Technical definitions for all terms appear in the next section. 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 [13], 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 [20].

Language-based DIFC. Language-based DIFC systems [19, 21, 25] 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 systems do not support multithreaded programs because of the difficulty and tractability of combining whole-program concurrency and type analysis. Laminar supports multithreaded programs by performing dynamic checks in the VM, thus obviating this heavyweight static analysis.

OS-based IFC. Asbestos [28] and HiStar [30] are new operating systems that provide DIFC properties. Flume [14] is a user level reference monitor that provides DIFC guarantees without making extensive changes to the underlying operating system.

As Table 1 shows, no OS DIFC system protects flows between individual application data structures. Threads accessing data structures with different labels in the same address space are not supported by any OS-based system, because the OS does not have an efficient way to mediate accesses to application-level data structures. The mechanism of page table protections is too coarse and expensive to effectively track information flow through application memory. 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 our application case studies use threads with heterogeneous security classes.

Laminar provides DIFC guarantees at the granularity of threads and data structures with modest changes to the VM and by adding 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 [29] (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 not pointers with access control information, as they are in capability-based OSes [16, 24]. Capability-based operating systems, like EROS [24] combine system and language mechanisms for strong security. However, capability systems cannot enforce DIFC rules, and programs must be completely rewritten to work with the capability programing model.

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 channel of static and dynamic DIFC systems leak an arbitrary number of bits [27]. 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 [15], or probabilistic channels [23]. Section 4.3.3 defines and discusses implicit information flows.

3. DIFC Model

All DIFC systems need some mechanism to denote the sensitivity of information and the privileges of the participating users. In this section we describe the mechanisms used by Laminar. We also describe the rules that DIFC systems use to determine whether an information flow is allowed.

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 ($\mathcal{T}$) [14]. A tag has no inherent meaning, and a new tag can be created by any thread. A set of tags is called a label.

Labels are assigned to data objects and principals. Data objects include data structures and system resources like files and sockets. Principals include threads and security regions. Principals, and only principals, can modify labels on data Principals, and only principals, can modify labels on data objects and other principals. Previous systems limit principals to the granularity of a process or address space. Our system is the first to expand principals to include threads and security regions.

Each data object 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

Issue	**PL solution** [21,25]	**OS solution** [14,28,30]	**Laminar solution**
Main subsystems modified	Compiler and type system	(1) Complete OS [28, 30] (2) User-level reference monitor and kernel module [14]	VM and kernel module
Trusted computing base	Compiler, VM, and OS	OS	VM and OS
Securing individual application data structures	Whole-program static analysis	Not supported Page table mechanisms would be inefficient	Dynamic analysis and VM enforcement using read/write barriers
Securing files and OS resources	Not supported	(1) Modify entire OS or (2) User-level reference monitor with kernel module support	Kernel module
Termination, timing, probabilistic channels	Not handled	Not handled	Not handled
Implicit information flow	Static analysis	Entire address space has single label	Dynamic analysis using lexically scoped *security regions*
Barrier to entry	New language or type system	Multithreaded applications whose threads have heterogeneous security needs are excluded	Incrementally deployable

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

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 it trusts Microsoft.

In a decentralized information flow 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 can be used to prevent authentication information from ever leaking to the network, and the tags on user data can prevent a malicious user from writing another user's secret data to an untrusted network connection.

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. Allowing unlabeled resources means every data structure in a program does not need to be labeled and every file in the file system does not need to be labeled.

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 [20]. 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 [16,24].

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, tags are rarely used for both purposes. We will refer to C_p^- as the set of tags for which principal p has the ability to declassify (drop endorsement), while C_p^+ refers to the tags that p can classify (endorse). Principals and data objects have both a secrecy and integrity label; a data object with 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 who 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. Information flow from x **to** y is allowed only if the following two rules are satisfied:

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:

$$S_x - C_x^- \subseteq S_y \cup C_y^+$$

The capability C_x^- ensures that principal x can use its declassification capability to send information to y. Similarly, C_y^+ means that y may receive information if it has the privilege to taint itself by extending its secrecy label.

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:

$$I_y - C_y^- \subseteq I_x \cup C_x^+$$

Intuitively, the integrity label of x should be at least as strong as destination y. x may need to endorse 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 be used to form a covert storage channel [30].

In Laminar, a principal p can 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 stated as:

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

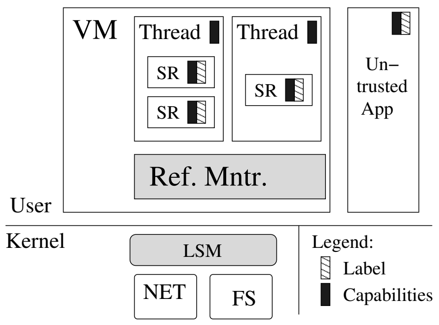

Figure 1. Design of Laminar. Trusted components are shaded.

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.

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 that 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, byte-code generator like `javac`. The byte codes are executed by the Laminar JIT and VM, which in turn is executed on the Laminar OS. Given the labeling, Laminar ensures that any accesses or modifications to labeled data follows the DIFC rules and occur in a security region.

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 process 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 a multi-threaded 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 keyword `secure` 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.

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 what data the program can touch inside the security region. The capabilities dictate how a security region may declassify data it encounters.

Security regions are the only principals allowed to read or write labeled data. This restriction 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 security region. For example, an assignment $w = r$ inside 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)
    Drop the given capability from the current principal.
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.

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 writes 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)
    Drop the given capabilities from the current principal.
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 types `tag_t` and `capability_t` represent a single tag or capability, respectively. The type `struct label` represents a set of tags that compose a label, and `capList_t` is a list of capabilities.

R is safe if and only if the information flow from r **to** R and from R **to** w is legal given the current labels of R.

Restricting labeled data to security regions also eases the burden of the programmer in adding security policies to existing programs. The programmer need only 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. Because of their limited size, security regions also simplify the task of auditing security-sensitive code.

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 `calendar` has the secrecy labels a and b. The security region is initialized with the secrecy labels a and b and can therefore read secret data guarded by these labels. Its integrity label i restricts it from reading data that is not tagged with i. The security region has the capability C(a$^-$) to declassify label a. The line `L1` is a valid information flow because the security region label at `L1` is $\{S(a, b), I(i)\}$, and is equal to $\underline{f}$. (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 c has the same secrecy label as the security region at that point. At line `L4`, the copying and relabeling of `s2` is legal because the security region has the a^- capability. Notice that if line `L4` were `copyAndLabel(s2,S(),I(i))`, it would result in a VM exception since the security region does not have the b^- capability. In this example, OS checks the file operations in line `L1` and the VM checks the operations in line `L2–L4`.

```
Calendar c;  //has labels {S(a,b),I(i)}
Schedule s3; //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]  c.addSchedule(s1);
[L3]  Schedule s2=c.getCommonSchedule();
[L4]  s3=Laminar.copyAndLabel(s2,S(b),I(i));
      ... }
```

Figure 4. Example pseudo-code to read and update a calendar.

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

Figure 5. Example of an implicit information flow.

4.3.2 Security region initialization

Laminar enforces certain rules on the properties used to initialize the security region. Let S_R,I_R and C_R be the security, integrity and capability set of a security region, R. If the thread containing R has the capability set C_T, then at initialization of R the following two rules should hold:

$$(x \in S_R \vee x \in I_R) \Rightarrow x \in C_T^+ \quad (1)$$

$$C_R \subseteq C_T \quad (2)$$

The first rule states that the creating thread must have the add capability for any tag given to a security region's label, as threads are unlabeled. The second rule says that the thread can only pass its own capabilities to the security region. These rules encapsulate the common sense understanding that a thread has control over the labels and capabilities it passes to a security region, and that the system will not let the thread create a security region with security properties that the thread itself lacks.

4.3.3 Implicit flows

A major benefit of security regions is that they limit the amount of analysis necessary to prevent implicit information flows. Implicit information flow [10] goes from control flow to data. 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, we 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.

At run time, Laminar prevents this flow because the write to `L` is illegal given the current label $\{S(h)\}$ of the security region. By checking writes against the security region labels, Laminar conservatively assumes that each write is an outward flow at the current label value of the security region. Similarly, each read is checked against the current label of the security region. Security regions over-approximate information flow to remain secure (like all dynamic DIFC systems [27]).

Laminar does execute the code in Figure 5 if H is false. If H is true, the program terminates to prevent the implicit flow from H to L. This policy does leak information via program termination. However, previous work has shown that neither static nor dynamic IFC systems can limit the amount of information leaked through a termination channel [3, 27]. In practice, these channels are low-bandwidth and difficult to exploit to learn sensitive data.

4.4 VM-OS interface

Security regions are abstractions that are visible to the VM but not 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 through the `set_task_label` system call, which is optimized out 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. There are three ways by which the Laminar OS allows principals to acquire capabilities: through the allocation of a new tag, as part of a `fork()`, and through 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. It can give the plus and minus capabilities for the new tag to any other principal with whom it can legally communicate.

Threads and security regions form a natural hierarchy of principals. When a kernel thread forks, it can initialize the new thread with a subset of its capabilities. Similarly, when a thread enters a security region, the thread specifies the subset of its capabilities that will be available. In general, when a new principal is created, its capabilities are a subset of its immediate parent, which is enforced by the VM and OS.

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 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 `set_task_label` system call is used by the VM to change the label of a thread at the beginning and end of a security region. Laminar does not allow security regions to change their labels because the VM relies on labels staying the same throughout lexically scoped regions in order to prevent leaks through local variables, as discussed in Section 5.1. We plan to add support for nested security regions, which will achieve the same functionality as allowing regions to change their labels.

A thread may start a security region with secrecy label $\{S(a)\}$ because it has the a^+ capability, but when the security region ends, the thread must drop the secrecy label, even if it does not have the a^- capability. The VM contains a thread running code at an integrity level known to the OS that allows it to drop all current labels for a thread without having the appropriate capabilities using the `drop_label_tcb` system call.

Having a single, high-integrity thread in the VM limits exposure to bugs as the OS only allows the thread to drop labels within a single address space; the VM cannot drop the labels on other applications. Second, only a small, auditable portion of the VM is trusted to run with this label.

Capability persistence and revocation. Capability persistence and revocation are always issues for capability-based systems, and Laminar does not innovate any new 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. Similarly, if a user wishes to revoke access to a resource she has already shared a capability for, 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 any object allocated within a security region with the label of that region. The `create_labeled` and `mkdir_labeled` kernel calls create labeled files and directories.

Like most other DIFC systems, Laminar uses immutable labels. To change a label, the user must copy the data object. Supporting 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 another 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. This leads to an unauthorized flow from B to A.

4.6 Compatibility challenges

Although Laminar is designed for incremental deployability, 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 and terminate the program. Such code must be modified to work in any DIFC system.

4.7 Trusted computing base

For Laminar, we added approximately $2,000$ lines of code to Jikes RVM, a $1,000$ line Linux security module, and 500 lines of modifications to the Linux kernel itself. 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 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. 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]. 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 1.6, JRockit, and J9 1.9 [1]. All subsequent uses of the term JVM refer to the Laminar enhanced version of Jikes RVM.

The JVM controls information flow by allowing labeled objects to be accessed only inside 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. Whenever 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

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

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.

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 resides in the heap; and statics, which reside in a global pool.

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 restrictions that (1) a local variable written in 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 to outside a security region may not be read inside the region if the region has integrity labels. The Laminar prototype implementation simplifies enforcement by requiring that security regions start a new method. The JVM ensures at run time that any method with a security region (1) does not return a value if the region has security labels and (2) takes no parameters if the region has integrity labels. 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. These restrictions are enforced dynamically using barriers inserted at static accesses inside security regions. A production implementation could support labeling of statics with modest overhead because static accesses are relatively infrequent compared to field and array element accesses. Labeled static variables are not needed for the applications in Section 7.

Supporting information flow for objects. The JVM tracks information flow for objects, which live in the heap. At allocation time, objects may be assigned immutable secrecy and integrity labels. 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 label, 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 label sets.

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

The JVM's compiler inserts *barriers* [8] (instrumentation at every operation of some type, e.g., at every read) into application code to enforce DIFC rules. Inside security regions, the compiler inserts barriers at *labeled object allocation* (but 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, these barriers load the accessed objects' secrecy and integrity `LabelSets` and check that they conform to the current security region's labels and capabilities. Outside security regions, read and write barriers check that the accessed objects are *unlabeled*. The compiler also inserts barriers inside security regions at *static* accesses to verify that static reads (writes) occur only in regions without integrity (secrecy) labels.

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

Figure 6. Example code to read the marks of two students. The `student1` and `student2` objects are labeled. The `principal` object contains the secrecy, integrity, and capabilities sets with which the security region is initialized.

The compiler inserts different barriers at an access depending on whether the access occurs inside 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 [22]. Static barriers add the same overhead that cloning would achieve.

Because object labels are immutable and security regions cannot change their labels, repeated barrier checks of the same object are redundant. We implement an intraprocedural, flow-sensitive dataflow analysis that identifies instructions for which a barrier would be redundant. 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. Figure 6 contains example code for computing 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 `principal` contains a set of labels and capabilities. If the capabilities and labels inside `principal` do not conform to the current threads capabilities, then the program terminates at `L1`. Once the security region starts, its current labels become those present in `principal`. Lines `L2` and `L3` are reads of labeled objects that will result in an error if the flow from $\underline{\text{student1}}$ or $\underline{\text{student2}}$ to $\underline{\text{principal}}$ is not allowed (recall that $\underline{x}$ is the label of x). In line `L4`, the JVM gives the unlabeled `obj` object the label $\underline{\text{principal}}$. In line `L5`, the security region attempts to change the labels of the object. The JVM allows this change as it has the required capabilities.

5.2 OS support

We have implemented support for DIFC in Linux version 2.6.22.6 as a Linux Security Module (LSM) [29]. 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 [17], 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, and integrity decreases. 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. Rather 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.

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 [14, 28].

Laminar allows pipes to be relabeled when empty, avoiding races with message delivery and the label on the pipe. Thus, a process attempting to change a label may block until the pipe is drained, and subsequent writers may block on the label change. Because the label on a pipe can be used to form a covert channel, Laminar does not allow the label of a file handle to be read. An application that is changing the label on a pipe should first send a message indicating the new label. The label on the pipe will not change until this message is retrieved.

6. Laminar overhead

This section measures the performance overhead of Laminar's subsystems. The performance loss on Java benchmarks without security regions is 10%. The Laminar OS incurs overhead 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. The results are normalized to values obtained on unmodified Linux 2.6.22 and Jikes RVM 3.0.0.

6.1 JVM overhead

Figure 7 shows the overhead of our Laminar-enabled JVM for the DaCapo benchmarks [7] and a fixed workload version of SPECjbb-2000 called `pseudojbb` [26]. We use a deterministic methodology called replay compilation [12] to control run-to-run variation due to timer-based sampling. The darker bar shows the overhead of dynamic barriers, which check dynamically if they are in a security region. Dynamic barriers add 21% overhead on average. The lighter bar is the overhead of using static barriers, 10% 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, 10% average overhead is representative of expected performance of code outside of a security region.

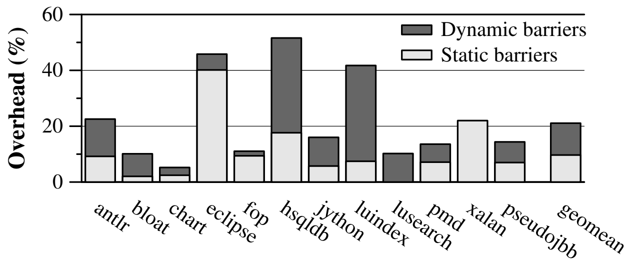

Figure 7. Laminar VM overhead on programs without security regions.

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	6877.00	7035.00	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.

6.2 OS overhead

We use the lmbench [18] 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 [29]. The only performance outlier is the "null I/O" benchmark, which has 31% overhead. This 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 application relative to unmodified Linux [14].

7. Application case studies

This section describes our case study applications and how we retrofit them with DIFC security policies. Table 3 summarizes the details of the applications that we use in our case study, and Figure 8 shows the overhead of running the modified version with Laminar. The retrofitted applications implement more powerful security policies than their unmodified counterparts, and all security-related code accounts for 10% or less of the source.

The figure breaks down the overhead of Laminar into four components. *Start/end SR* is the overhead of all application-code modifications to support DIFC, including the starting and ending of security regions and other user-called operations such as `copyAndLabel`. The *Alloc barriers* configuration is extra time for allocating labeled objects and assigning their label sets. Finally, *Static barriers* is the overhead from read and write barriers where the

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.

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.

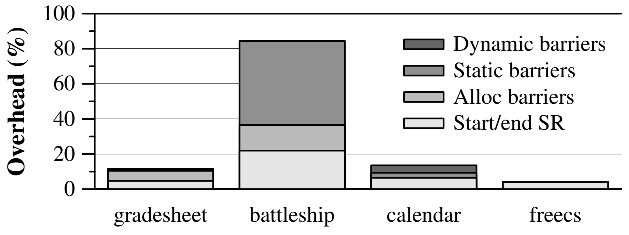

Figure 8. Overhead of executing applications retrofitted with Laminar.

security context is known at compile time, and *Dynamic barriers* is extra overhead from barriers that check context at run time. We note that all the applications run correctly with static barriers except Calendar, which requires dynamic barriers because some methods are called both from 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, 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 [14] adds 30–40% slowdown on the MoinMoin wiki application. But Flume does not support multithreaded applications and all data structures within a process share the same label.

7.1 GradeSheet

GradeSheet is a small program that manages the grades of students [6]. This program uses three types of principals: professors, TAs and students. 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 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. This means they 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 12% 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 shoots down 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. This ensures that the locations can be declassified by only the player. The original implementation had a player 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. Figure 8 shows that the secured version adds 84% overhead with static barriers. The high overhead is 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

We modified the `k5nCal` multithreaded desktop calendar to label all data structures and `.ics` files which store a user's calendar information with the user's secrecy tag, much like the examples from earlier in the paper[2]. All functions that access this data are wrapped inside security regions, including a scheduler that finds available meeting times for multiple users. The default behavior was to allow a user to view all other users' calendars, which 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 experiments schedule 1,000 meetings Figure 8 shows that the secured version of Calendar runs 14% slower than unmodified Calendar, which includes overhead due to dynamic barriers.

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

7.4 FreeCS chat server

FreeCS is an open-source chat server written in Java[3]. It allows multiple users to connect to the server and communicate with each other. FreeCS supports 47 commands such as creating groups, inviting other users, changing the theme of the chat room and so on. The existing security policy consists of an authorization framework that dictates who can use which command. 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. The roles make a natural case for using integrity tags to secure the commands in FreeCS. 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. Laminar improves the security code in FreeCS by localizing the checks to the data structure. The programmer only has to label the data structure and access it inside a security region. Most of our modifications were done in two classes—`Group` and `User`. We also changed the authentication module to ensure that users are given the right capabilities when they log in. Our experiment for FreeCS consists of $4,000$ users, each invoking three different commands. We measure the time taken by the server to process all the requests. Figure 8 shows that Laminar's overhead is at most 4%.

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, e.g., array of student grades in GradeSheet and 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. This supports our hypothesis that only local 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 to achieve in OS-based systems [14, 28, 30], since they support a single label on the whole or large segments of the address spaces, whereas Laminar can easily solve this problem with the help of its VM.

8. Conclusion

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

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.

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

[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 *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 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] 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.

[14] 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.

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

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

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

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

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

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

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

[22] 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.

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

[24] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: a fast capability system. In *SOSP*, 1999.

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

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

[27] 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.

[28] 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.

[29] 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.

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

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