Practical Fine-Grained Information Flow Control Using Laminar

DONALD E. PORTER, Stony Brook University
MICHAEL D. BOND, Ohio State University
INDRAJIT ROY, Hewlett Packard Labs
KATHRYN S. MCKINLEY, Microsoft Research
EMMETT WITCHEL, The University of Texas at Austin

4

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 solutions provide no guarantees against security violations on system resources such as files and sockets. Operating system solutions mediate accesses to system resources but are either inefficient or imprecise at monitoring the flow of information through fine-grained program data structures. This article describes Laminar, the first system to implement DIFC using a unified set of abstractions for OS resources and heap-allocated objects. Programmers express security policies by labeling data with secrecy and integrity labels and access the labeled data in *security methods*. 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 article shows that security methods ease incremental deployment and limit dynamic security checks by retrofitting 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 5% to 56%. Compared to prior DIFC systems, Laminar supports a more general class of multithreaded DIFC programs efficiently and integrates language and OS abstractions.

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

Additional Key Words and Phrases: Information flow control, Java virtual machine, operating systems, security method

ACM Reference Format:
Donald E. Porter, Michael D. Bond, Indrajit Roy, Kathryn S. McKinley, and Emmett Witchel. 2014. Practical fine-grained information flow control using Laminar. ACM Trans. Program. Lang. Syst. 37, 1, Article 4 (October 2014), 50 pages.
DOI: http://dx.doi.org/10.1145/2638548

This research was supported in part by NSF grants CAREER-0644205, CNS-0905602, CAREER CNS-1149229, CNS-1161541, CNS-1228839, CNS-1228843, CAREER-1253703, CSR-1218695, SHF-0910818, NIH LM011028-01, and the Office of the Vice President for Research at Stony Brook University.

Q1

Authors' addresses: D. E. Porter, Computer Science Department, Stony Brook University; M. D. Bond, Department of Computer Science and Engineering, Ohio State University; I. Roy, Hewlett Packard Labs; K. S. McKinley, Microsoft Research, Redmond; E. Witchel, Department of Computer Science, The University of Texas at Austin.

DOI: http://dx.doi.org/10.1145/2638548

1. INTRODUCTION

As computer systems support more aspects of modern life, from finance to health care to energy, the security of these systems 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 a program downloaded from the Internet.

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 policy against sending a user's credit card number on the network should be enforced whether the number originates from a file or an application data structure. In current systems, the OS governs the security of files, and application-specific logic governs the security of data structures; because these mechanisms are completely distinct, developers must understand both mechanisms and ensure that they interoperate correctly. This article describes Laminar, which integrates PL and OS security mechanisms under a common set of programmer abstractions and uniformly enforces programmer-specified security policies at all levels of the software stack.

Laminar builds on the Decentralized Label Model (DLM) [Myers and Liskov 1997], which expresses more powerful, sophisticated, and intuitive security policies than traditional security models. The enforcement of DLM restrictions is called Decentralized Information Flow Control (DIFC). DIFC is more expressive than traditional access control. For instance, traditional access control models 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, a DIFC policy may give an application the right to read a file and simultaneously forbid it to broadcast the contents of the file over an unsecured network channel. A DIFC implementation dynamically or statically enforces user-specified security policies by tracking information flow throughout the system.

In the decentralized label model, users create *tags*, which represent secrecy or integrity concerns. A set of tags is called a *label*, and all data and application threads have an associated secrecy label and an integrity label. The system restricts the flow of information according to these labels. Secrecy guarantees prevent sensitive information from escaping the system (no illegal reads),[1] and integrity guarantees prevent external information from corrupting the system (no illegal writes).

As an example, suppose Alice and Bob want to schedule a meeting without disclosing other appointments on their calendars. In the DLM model, Alice and Bob each place a tag in the *secrecy label* on their calendar files. Alice and Bob can give the calendar application permission to read these files but only if the application *taints* its own secrecy label with the secrecy tags of each file. A tainted application thread may no longer write to less-secret outputs, such as the terminal or the network. In our example, the tainted thread may read each calendar file and select an agreeable meeting time, but the thread can only write output to a file or data structure labeled with both Alice's and Bob's secrecy tags. In order for the calendar application to output a nonsecret meeting time, Alice and Bob must provide a *declassifier* with the capability of removing their tags from a datum's secrecy label. The declassifier is a piece of code responsible for checking that its output conforms to a secrecy policy associated with a tag; the declassifier may write acceptable data to a less-secret output. In the calendar example, Alice and Bob might both provide declassifiers; each declassifier can generate output

[1]The literature uses both *confidentiality* and *secrecy* for this guarantee. We use S for secrecy, I for integrity, and C for capabilities to avoid ambiguity.

without that user's tag in the secrecy label. For instance, Bob's declassifier might read the labeled meeting time and check that the output is simply a date and does not include mention of his upcoming vacation to Las Vegas. Note that DIFC exists in addition to traditional access control; for example, a web server would not be allowed to open either calendar file due to standard OS-level permission checks.

Similarly, Alice may use an *integrity label* on her calendar file to ensure that any updates to the file respect certain invariants. Suppose Alice's calendar is stored as a chronologically sorted list of appointments. Untrusted code that adds appointments to Alice's calendar might serialize her appointments into the on-disk format and store the pending data in a memory buffer. Alice could then run this buffer through an *endorser*, which ensures that the pending data write meets the specifications of her calendar format, such as checking that all appointments are sorted chronologically. Just as secrecy labels can be removed from the output of a computation by a declassifier, an endorser is trusted with the capability to add a tag to the integrity label of inputs that it validates. Once the endorser has validated that its input is trustworthy, the endorser adds Alice's integrity tag to its integrity label and writes a new version of her calendar file.

DIFC provides two key advantages: precise rules for the legal propagation of data through an application 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 the label modulates how data can flow through threads and data containers (e.g., files and data structures). The decision to declassify data is localized to small pieces of code that programmers may closely audit. The result is a system where security policies are easier to express, maintain, and modify than with traditional security models.

Combining the Strengths of Language and Operating System Enforcement. Laminar is a new DIFC system design that features a common security abstraction and labeling scheme for program objects and OS resources such as files and sockets. The Java Virtual Machine (VM) and OS coordinate to comprehensively enforce rules within an application, among applications, and through OS resources.

Prior DIFC systems are implemented at the language level [Chandra and Franz 2007; Myers and Liskov 1997; Myers et al. 2001; Nair et al. 2008] in the operating system [Krohn et al. 2007; Vandebogart et al. 2007; Zeldovich et al. 2006], or in the architecture [Tiwari et al. 2009a; Vachharajani et al. 2004; Zeldovich et al. 2008]. Each approach has strengths and limitations. Language-based DIFC systems can track information flow through data structures within a program but have little visibility into OS-managed resources, such as files and pipes. In contrast, OS-based DIFC systems track labels at the coarse granularity of pages or a process's virtual address space rather than on individual data structures. Information flow rules are enforced on OS-level abstractions, such as sockets and files. For many simple applications, these coarse-grained rules simplify DIFC adoption. However, OS protection mechanisms are not a good fit for managing information flow on data structures within an application because the OS's primary tool is page-level protections. Although an application developer could group objects with similar labels on similarly labeled pages, this undermines developer productivity and application efficiency. Thus, we believe that coordinating language and OS mechanisms will maximize security and programmability.

We limit the scope of this article to DIFC implementations on commodity hardware. Architecture-based solutions track data labels on various low-level hardware features, such as CPU registers, memory, cache lines, or even gates, but require similar coordination with trusted software to manage the labels.

Language-based DIFC systems can be further categorized by how they enforce DIFC rules: static analysis [Myers and Liskov 1997; Myers et al. 2001], dynamic analysis

[Shroff et al. 2007], or a hybrid [Chandra and Franz 2007; Nair et al. 2008]. Static systems generally introduce a type system that is expressive and powerful but difficult to program or retrofit onto existing code. Because static systems do most security analysis at compile time, they introduce little runtime overhead; static systems may insert dynamic checks for properties that cannot be established at compile time. Dynamic systems generally enforce information flow rules by mediating every operation at runtime but with relatively high performance overheads. Purely dynamic systems also struggle to regulate implicit flows (discussed further in Section 6.4) and can ultimately reject safe programs or leak sensitive data [Russo and Sabelfeld 2010].

Most language-level systems are actually a hybrid of static and dynamic analysis. Each design strikes a balance among changes to the programming language to facilitate static analysis, runtime overheads, and security guarantees. The Laminar design restricts the programming model slightly, ensuring that all security properties can be checked dynamically. Laminar does employ intraprocedural static analysis at Just-in-Time (JIT) compilation time to optimize security checks.

Limiting the Scope of Analysis. A second key contribution of Laminar is the design of a language-level feature, called a *security method*, which strikes a unique balance between programmability and efficiency. Developers place all security-sensitive program logic in security methods. The Laminar VM requires that all operations on labeled data or system resources occur within security methods, according to developer-specified policies. In addition, all methods dynamically invoked by a security method, directly or transitively, are security methods. Code that attempts to manipulate security-sensitive data outside of a security method will fail.

Laminar enforces stringent requirements on transitions to and from security methods, restricting both control and data flow. These restrictions are enforced dynamically by VM instrumentation. Security methods reduce the overhead of dynamic security checks because only code within security methods requires complex DIFC checks.

Security methods also minimize the code changes required to adopt DIFC. In our case studies, changes to adopt security methods account for 10% or fewer of the total lines of code, which suggests that pervasive program modifications are unnecessary to use DIFC with Laminar.

Contributions. The contributions of this article are as follows:

Q2

(1) We present the design and implementation of Laminar, the first system to unify PL and OS mechanisms for enforcing DIFC. Laminar features a novel division of responsibilities between the VM and OS.
(2) We introduce security methods, an intuitive security primitive that reduces the work required to convert an application to use DIFC, makes code auditing easier, and makes the DIFC implementation simpler and more efficient.
(3) We present the design and implementation of Laminar in the Linux OS and Jikes RVM, a Java research VM.
(4) We evaluate 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 5% to 56%.
(5) Based on our experiences, we substantially modified the conference publication that introduced this research [Roy et al. 2009]. We replace security regions with security methods to simplify our implementation. We use only dynamic analysis to simplify the enforcement security policies. We identify and fix a covert channel bug arising from the interaction of termination and concurrency. Furthermore, we improve the programming model for initializing and using security labels.

(6) We describe strengths and limitations of the Laminar model, its open challenges, and potential solutions. In particular, Laminar is one of the few DIFC systems to attempt multithreading support, which is prone to high-bandwidth timing channels. Laminar cannot prevent all of these timing channels, but we outline how subsequent work by others [Askarov et al. 2010; Askarov and Myers 2012; Zhang et al. 2011] could strengthen the Laminar threading model.

Initial results suggest that integrating PL and OS DIFC enforcement is practical and incurs low overheads. Our experience with Laminar shows that it prevents some termination information channels, but it cannot yet make guarantees on some timing channels. We believe restrictions on the programming model within security methods can solve some of these problems, but this research leaves open the definition of such a formalism and accompanying proofs. Laminar provides a first step for application developers to write expressive abstractions with fine-grained, powerful, and useful security policies that span program data structures and system resources.

2. DIFC MODEL

This section describes how the DIFC model specifies and enforces safe information flows and how Laminar embodies the DIFC model. All DIFC systems denote the sensitivity of information and the privileges of the participating users, as well as describe application-specific policies that map between users and sensitive information. The security policy is defined in terms of *principals* that read and write data in the system. Examples of principals in DIFC systems are users [Myers et al. 2001], processes [Krohn et al. 2007], and kernel threads [Zeldovich et al. 2006]. Principals in Laminar are kernel threads, which ultimately work on behalf of human users or other application-level actors.

2.1. DIFC Abstractions

Standard DIFC security abstractions include tags, labels, and capabilities. Tags are short, arbitrary tokens drawn from a large universe of possible values ($\mathcal{T}$) [Krohn et al. 2007]. Programmers use tags to denote a unique secrecy or integrity property, but a tag has no inherent meaning. Programmers may create tags dynamically and may persist tags beyond execution of an application. 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, and 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. Laminar is the first to support threads as principals and enforce DIFC at object granularity.

Each data object and 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 u in I_x indicates that the owner of integrity tag u *endorses* the data. Data integrity is a guarantee that data exist in the same state as when they were endorsed by a principal. For example, if Microsoft endorses a data file, then a user can choose to trust the file's contents if she trusts Microsoft. With integrity enforcement, only Microsoft may modify the integrity-labeled file. However, Microsoft may choose to remove the integrity label, or some other application may write the file, but without the Microsoft integrity label. In either case, the file's consumer will no longer trust the contents as coming from Microsoft. In general, a principal's labels restrict the interaction that the principal has with other principals and data objects.

A partial ordering of labels imposed by the subset relation forms a lattice [Denning 1976]. Secrecy and integrity may be treated separately, as asymmetric duals. The bottom of the secrecy lattice is the least restricted label (public): Any principal can read it. The bottom of the integrity lattice is the most secure (trusted): All principals can trust it. Adding secrecy tags to a label restricts the use of the data, moving higher in the lattice. The most restricted data at the top of the secrecy lattice includes all secrecy tags. The bottom of the integrity lattice is the most secure (trusted) and includes all integrity tags. Removing integrity tags moves the label higher in the lattice and the data are less trusted. The top of the integrity lattice has no integrity tags—no principal endorses it.

Some other DIFC explanations put an empty integrity label at the bottom of the lattice so that adding tags moves up the lattice, as opposed to the preceding description that places the label with all integrity tags at the bottom, so that moving up the lattice adds restrictions. Both representations are functionally equivalent. For clarity, this article generally discusses the secrecy and integrity labels separately, but occassionally some explanations treat principals and data as having a single label and capability set for ease of exposition.

Because Laminar's threat model includes code that may be contributed to the application by an adversary, all application data are assigned an empty label (public and untrusted) by default. Data from the JVM itself are public and trusted. The programmer need not label every data structure, nor does the OS need to label every file in the file system. Code that executes with a nonempty integrity label must sanitize untrusted data before a read. Default empty labels make Laminar easier to deploy incrementally, but introduce some asymmetry in the treatment of secrecy and integrity tags.

A principal may change the label of a data object or principal if and only if the principal has the appropriate capabilities, which generalize ownership of tags [Myers and Liskov 1997]. A principal p has a capability set C_p that defines whether the principal has the privilege to add or remove a tag. For each tag t, let t^+ and t^- denote the capabilities to add and remove the tag t.

If tag t is used for secrecy, a principal with the capability t^+ may *classify* data with secrecy tag t. Classification raises data to a higher secrecy level. Given the t^- capability, a principal may *declassify* these data. Declassification lowers the secrecy level. Principals may 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 [Levy 1984; Shapiro et al. 1999].

DIFC handles integrity similarly to secrecy. A principal with integrity tag t is claiming to represent a certain level of integrity; the system prevents the principal from reading data with a lower integrity label, which could undermine the integrity of the computation. Given the t^+ capability, a principal may *endorse* data with integrity tag t, generally after validating that the input data meet some requirements. Given the t^- capability, a principal may drop the endorsement and read untrusted data. For example, code and data signed by a software vendor could run with that vendor's integrity tag. If the program wants to load an unlabeled, third-party extension, 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 that 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: $\langle S(s), I(i)\rangle$. An empty label set is written: $\langle S(), I()\rangle$. The capability set of a principal that can add both s and i but can drop only i is written: $\langle C(s^+, i^+, i^-)\rangle$.

2.2. 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, principal x sending a message to principal y, and principal y reading a file x are all information flows from x to y. If principal x writes to a file y, then we say information flows from source x to destination y. Laminar enforces the following information flow rules for x to y:

Secrecy rule. Bell and LaPadula introduced the simple security property and the *-property for secrecy [Bell and LaPadula 1973]. The simple security property states that no principal may read data at a higher level (*no read up*), and the *-property states that a principal may not 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 tag t from its label S_x if it has the declassification capability for t (i.e., $t^- \in C_x^-$). 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 and restricts reads from lower integrity (*no read down*) and writes to higher integrity (*no write up*) [Biba 1977]. Laminar enforces 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 tags from its 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 [Zeldovich et al. 2006].

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. This is formally stated as:

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

2.3. Calendar Example

Again, consider scheduling a meeting between Bob and Alice using a calendar server that is not administered by either Alice or Bob. Alice's calendar file has a secrecy tag, a, and integrity tag i; 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 $\langle S(a) \rangle$. A thread in the server uses the a^+ capability to start a security method with secrecy tag a that reads Alice's calendar file. Once the server's thread has the label $\langle S(a) \rangle$, 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 $\langle S(a) \rangle$, 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 it also becomes unable to write to unlabeled channels.

Ensuring integrity. Alice also places an integrity label on her calendar file, which is propagated to the heap data structures representing her calendar. In order for any thread to update Alice's calendar, the thread must add the i integrity tag to its label. In general, the capability to add this tag would be restricted to code that is trusted to check that inputs or updates uphold application invariants. In this example, much of the calendar code may run with an empty integrity label, but once a meeting request is ready to be added to Alice's calendar, the meeting request is checked by Alice's endorser. If the checks pass, Alice's endorser adds the i tag to the meeting request data structure. The code that writes the updated calendar to disk must also run with the i tag, preventing data from untrusted heap objects from inadvertently being written to the calendar file.

Sharing secrets with trusted partners. Alice and Bob collaborate to schedule a meeting while both retain fine-grained control over what information is exposed. After the scheduler has read Alice's and Bob's calendar files, the output data are labeled with the a and b secrecy tags. Alice's module has access to her a^- capability, so the server calls her code, which validates that the output does not disclose unintended information to Bob. Alice's module then removes the a tag from the output data, publishing the meeting time to Bob. Alice controls which of her data flow out of the scheduler. Bob does the same, and the scheduler can communicate with both of them and coordinate 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 is more cumbersome. OS-based DIFC systems require the declassifier to run as a separate process. Language-based DIFC systems require programmers to annotate the entire application. By integrating OS and language techniques, Laminar simplifies incremental DIFC adoption.

2.4. Goals and Threat Model

This subsection describes our threat model and its rationale at a high level. We revisit these security properties in Section 6, after describing our system design and implementation. Section 10 surveys related work in more detail, but here we summarize key categories of DIFC systems and challenges in DIFC adoption. DIFC systems can be roughly categorized by how they enforce flows: static analysis, dynamic language-level analysis, or OS-level enforcement.

Incremental Adoption. A key design goal of Laminar is facilitating incremental adoption of DIFC on a large body of code. The ease with which a programmer can adopt DIFC is an issue for most DIFC designs. DIFC based on static analysis often requires substantial annotations of the program with a new type system. OS-based DIFC requires substantial reorganization of the application code in order to segregate data pages and code by label. It is unclear whether a language-level dynamic analysis is any easier to adopt. Although there has been some work in this area, it has generally enforced only simple policies on outputs [Chandra and Franz 2007] or had problems with "label creep," which requires error-prone, manual analysis by the programmer [Nair et al. 2008].

The insight underlying Laminar's security-method-based design is that many applications already handle sensitive data only in relatively small portions of their code. For instance, web server authentication code is generally small relative to all of the code that generates and transmits web content. Thus, Laminar is designed so that the programmer audits only these relatively small portions of preexisting code for correct handling of sensitive data. Sensitive code is placed in security methods, and the system dynamically checks that all information flows according to the restrictions imposed by the developer and end users.

Laminar enforces DIFC rules using a combination of dynamic analysis and programmer annotations (i.e., security methods). Compared to other dynamic or hybrid language-level systems, Laminar is generally more efficient than previous systems because of careful implementation choices and limited scope of analysis. As discussed earlier, all DIFC systems require some measure of work to adopt, and our experience is that security methods minimize the effort without sacrificing functionality.

Integration of OS and PL Abstractions. Laminar integrates OS and PL DIFC abstractions to implement uniform policies and label management across resources. Existing systems cannot easily integrate these abstractions. For instance, a PL system might enforce all-or-nothing policies about output or might make educated guesses about information flow through OS abstractions, but it cannot ensure that these rules are followed once data leaves the application.

Threading. A key aspect of incremental deployability is tracking information flow *through* a program, including with multiple, concurrent threads. OS systems mediate multiprocess concurrency through explicit channels and at page granularity. In practice, these systems cannot track fine-grained information flow through traditional thread packages without major modifications to the application. PL systems have generally avoided multithreading because it increases the risks of covert channels. Laminar does permit multithreading, but cannot prevent all timing channels attacks. This article identifies some threats and points to solutions developed after the initial publication of this work [Roy et al. 2009] that could be integrated into security methods to mitigate these channels.

Threat Model. In a DIFC system, the primary concern is limiting the ability of one principal to access another principal's data. So, in our threat model, the attacker may have contributed code to the application and is executing as principal (thread) A. Laminar does not allow principal A to explicitly read or write another principal B's data (e.g., by explicit assignment in the program) without acquiring appropriate secrecy and integrity labels. Any other user controls access to her data by controlling which principals she gives the capabilities to add and remove tags associated with her data.

Limitations. Like most DIFC systems, the Laminar VM and OS mediate all explicit assignments of labeled data, as described in Sections 4 and 5. Laminar prevents implicit information flows by restricting the visibility that untrusted code has into the control flow of a security method, including restrictions on input and output variables (discussed further in Section 6.4).

Eliminating all timing, termination, and other covert channels are open problems [Denning and Denning 1977; Lampson 1973] and beyond the scope of this article.

In particular, it is well established that preventing all these channels on a general-purpose programming model is tantamount to solving the halting problem [Denning and Denning 1977]. In order to eliminate information leaks due to unbounded execution, more recent work has investigated highly restricted programming models (e.g., without unbounded loops [Tiwari et al. 2009b]) or bounding the execution time of code that manipulates sensitive data [Tiwari et al. 2009a].

To facilitate incremental adoption, Laminar places capabilities in threads, rather than statically mapping them to functions. The underlying tradeoff is that the programmer can more easily invoke standard libraries from a security method. For example, programmers may therefore manipulate secure objects using standard implementations of arrays, lists, and sets. Code invoked from a security method executes as if it were in the security method. This choice introduces some risk for a confused deputy problem and requires trusting the caller of a security method to manage capabilities. The Laminar design mitigates the risk of capability management errors by requiring that all endorsers and declassifiers be declared `final` and that non-endorser/declassifier security methods do not accept capabilities as arguments. These issues are discussed further in Section 6.6.

Two key innovations of Laminar are support for multiple threads and the ability of a single thread to transition between different trust levels—facilitating incremental adoption but also introducing new opportunities for covert channels based on the timing of these transitions. Section 6 describes the new classes of timing and termination channels that these features could introduce and how Laminar mitigates them. To summarize, Laminar restricts the ability to create a channel based on control flow within a security method by requiring a single exit point from a security method and carefully mediating any OS- or VM-level storage channel, such as the thread's capabilities. The article also discusses how more recent work, such as predictive timing models [Askarov et al. 2010], could be applied to the Laminar prototype to further reduce covert channels, especially through thread synchronization. These issues are discussed further in Section 6.

3. DESIGN

This section describes the Laminar programming model and how Laminar enforces DIFC in an enhanced VM and OS.

3.1. Overview

Figure 1 illustrates the Laminar architecture. The OS kernel reference monitor mediates accesses to system resources. The VM enforces DIFC rules within the application's address space. Only the OS kernel and VM are in the Laminar trusted computing base. The OS kernel and VM trust each other as well.

The Laminar OS kernel extends a standard OS kernel with a Laminar security module for information flow control. Users and programmers invoke the Laminar kernel security APIs to create tags, store capabilities for their tags, and label their data in files. Users launch processes with a subset of their tags and capabilities. The Laminar OS kernel governs information flows through all standard kernel interfaces, including through devices, files, pipes, and sockets. DIFC rules are enforced by the kernel on all threads, whether the threads are of the same or different processes. Resources and principals without an explicit label have empty secrecy and integrity labels, facilitating incremental adoption. Our prototype uses the Linux Security Modules [Wright et al. 2002] framework, although the design could be extended to any OS that provides similar hooks to an in-kernel reference monitor to label kernel objects and mediate system calls that could create an information flow.

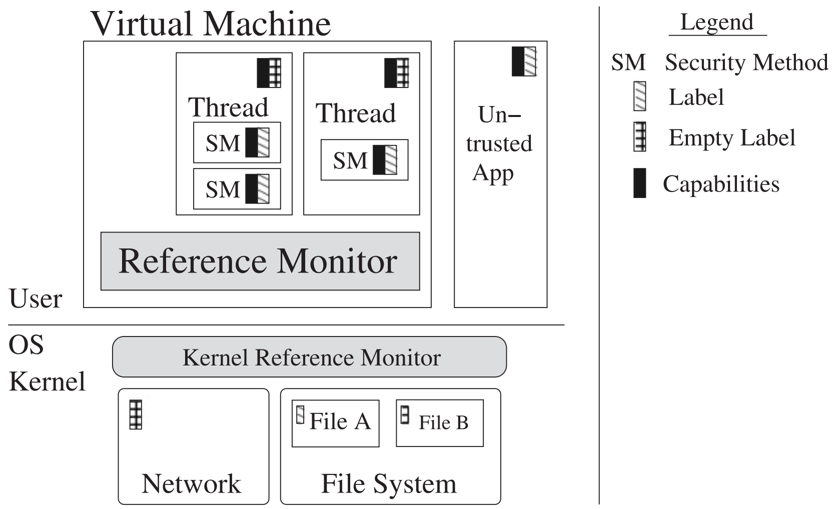

Fig. 1. Design of Laminar. An OS kernel reference monitor and VM reference monitor enforce information flow control. All data is labeled, including objects in the VM, as well as OS abstractions, such as files and sockets. Objects without explicit labels default to empty secrecy (public) and integrity (untrusted) labels. Threads have capabilities and an empty label. A thread enters a Security Method (SM) to acquire a nonempty label. A security method may optionally receive capabilities from the calling thread.

To regulate information flows within an application, Laminar extends the runtime system of a standard Java VM. By default, the Laminar OS kernel requires all threads within a process to have the same secrecy and integrity labels. The OS relaxes this restriction for threads running on the trusted Laminar VM. The Laminar VM binary is labeled with a special TCB integrity tag, which indicates to the OS that this application is trusted to control information flows within its address space. Although the kernel trusts the Laminar VM to regulate flows within the address space, the kernel still checks all accesses to system resources.

The Laminar VM regulates information flow between heap objects within a thread and between threads of the same process via these objects. The Laminar VM inserts dynamic DIFC checks to regulate DIFC flows.

The key abstraction for manipulating labeled data is the *security method*. Programmers explicitly declare security methods. In addition, any method invoked directly or transitively from a declared security method is implicitly defined as a security method. Outside of a security method, a thread has empty secrecy and integrity labels and may only read or write data with empty secrecy and integrity labels. The VM terminates the program if it attempts to read or write any labeled data outside a security method. If a thread has the capability to add a tag to its secrecy or integrity labels, the thread may change its labels by entering a security method. Within a security method, a thread may read or write data with nonempty labels as long as the reads and writes constitute a legal information flow according to the capabilities specified by the parameters. A thread typically runs with a subset of the user's capabilities, and a security method specifies a subset of the thread's capabilities. Security methods may nest. Each security method may only have a subset of the parent security method's capabilities and may only change its labels as permitted by the parent's capabilities.

For example, Alice writes a program in Java with the Laminar programming model and uses the Laminar API (see next paragraphs and Table I). Alice compiles the code using a standard, untrusted bytecode generator such as `javac`. The Laminar JIT compiler and VM execute the bytecode, and the Laminar OS kernel executes the Laminar

Table I. Laminar API

These functions manipulate labels and capabilities. `LabelType` denotes whether a request is for a secrecy label or integrity label. `CapType` denotes plus, minus, or both capabilities for a tag. The `getCurrentLabel` and `copyAndLabel` functions may be invoked inside of a security method, but tags and capabilities may only be created and destroyed outside of a security method, using `createAndAddCapability` and `removeCapability`, respectively. The API has wrapper functions (not shown) for the new Laminar system calls. `Label` stores a set of secrecy or integrity tags (Section 5.1).

Function	Description
`Label getCurrentLabel(LabelType t)`	Return the current secrecy or integrity label of the security method as an opaque object. Label objects cannot be enumerated.
`Object copyAndLabel(Object o, Label S, Label I)`	Return a copy of the object `o` with new secrecy label `S` and integrity label I.
`CapSet getCurrentCapabilities()`	Return the current capability set of the thread as an opaque object.
`Label createAndAddCapability()`	Create a new tag and add capabilities to the current thread. Must be used outside of a security method.
`void removeCapability(CapType c, Label l)`	Drop the capabilities listed in `c` (plus and/or minus) associated with the tags in `l` from the current thread. Must be used outside of a security method.

VM. During execution, the program labels data with security and integrity tags that it obtains from the kernel API. The OS kernel and VM thus use the same tag namespace for the system resources and objects. For example, the application reads data from a labeled file into a data structure with the same labels. The Laminar VM ensures that any accesses or modifications to labeled data follow the DIFC rules and occur in a security method, a labeled method specified by the programmer.

Restricting security policies to security methods makes it easier to add security policies to existing programs. Furthermore, auditing security methods will generally be easier than auditing the entire program. These features should facilitate incremental deployment of Laminar in existing systems. Security methods also decrease the cost of performing dynamic security checks in the Laminar runtime.

Laminar does not track information flows through local variables. Because labeled data are manipulated in security methods, locals in an untrusted parent are out of scope inside the security method and vice versa. With additional static analysis on information flow through locals, one might be able to safely implement security methods as arbitrary, lexically scoped regions, as originally proposed [Roy et al. 2009]. We expect that the additional static analysis required to support lexically scoped regions would be easiest to implement in the Java compiler (`javac`), but these properties might also be checked by the JVM during bytecode verification. We found mediating flows through locals at method boundaries to strike a good balance between implementation complexity for the application programmer and JVM developer.

3.2. Programming Model, VM, and OS Interaction

Laminar provides language extensions, a security library, and security system calls. Table I depicts the Java APIs, which include methods that perform tag creation, declassification, label queries, and capability queries. The Laminar OS kernel exports security system calls to the trusted VM for capability and label management, as shown in Table II. These system calls are used by the Laminar VM internally to implement security methods and are not directly exposed to Laminar applications. An application not running on the Laminar VM may directly use these system calls to manage its capabilities and labels, excluding `drop_label_tcb`, which can only be issued by the trusted Laminar VM. The Laminar OS securely stores all of the persistent capabilities of a user so that these capabilities can be used across user sessions. On login, the OS kernel gives the capabilities of the user to the login shell. Laminar does not innovate in

Table II. 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.

System Call	Description
`tag_t alloc_tag(capList_t &caps)`	Return a new tag, add plus and minus capabilities to the calling principal, and write new capabilities into `caps`.
`int add_task_tag(tag_t t, int type)`	Add a tag to the current principal's secrecy or integrity label (secrecy or integrity selected by `type`), as allowed by the principal's capabilities.
`int remove_task_tag(tag_t t, int type)`	Remove a tag from the current principal's secrecy or integrity label (secrecy or integrity selected by `type`).
`int drop_label_tcb(pid_t tid)`	Drop the current temporary labels of the thread without capability checks, invoked 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 by the VM to suspend a capability only for a security method 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 *S, struct label *I)`	Create a labeled file with labels `S` and `I`.
`int mkdir_labeled(char* name, mode_t m, struct label *S, struct label *I)`	Create a labeled directory with labels `S` and `I`.

capability persistence but rather adopts a simple and stylized model. Asbestos develops a more robust model for persistent storage of tags [Vandebogart et al. 2007].

The `secure` keyword applies to methods used as security methods. The VM and kernel enforce the rule that the program may only access labeled data objects (e.g., files, heap-allocated objects, arrays) inside security methods, which includes all methods directly or transitively invoked from a declared security method. Outside security methods, threads always have empty labels but may hold capabilities that determine whether the thread may enter a security method. Threads are the only principals in Laminar, and the VM modifies the thread's labels and capabilities when it enters and exits a security method. When a thread enters a security method, it dynamically passes the desired secrecy label and integrity label as arguments to the method, using the opaque `Label` object. If the security method endorses or declassifies data, it may also accept the necessary capabilities as an argument, as a `CapSet` object. During the execution of a security method, the VM internally uses these labels and capabilities for DIFC enforcement, and the kernel mediates thread accesses to system resources according to the security method's labels. Because security methods are not visible to the kernel, the VM proxies the security method by tainting the thread with the correct labels and capabilities. At the end of the security method, the VM restores the thread's original capabilities and labels.

3.3. Security Methods

A *security method* is a special method type that has parameters for a secrecy and integrity label. A security method that can endorse or declassify data also has a parameter for a capability set. The labels dictate which data the program may touch inside the security method. Labels on secure method parameters must satisfy the data flow constraints of the labels on the security method, and the label on the returned data must satisfy the data flow equations for the labels of the calling context. In the Laminar implementation, these labels and capabilities are represented as sets that can be variably sized and assigned at runtime. Label and capability sets are stored as opaque objects, which cannot be enumerated (see Section 5.1).

Only code within a security method can access data with nonempty labels. Security methods demarcate the methods that are security sensitive, thus easing the programmer's burden when adding security policies to existing programs. The programmer must place all code that references labeled data in a security method, such as a routine that reads a sensitive file into a data structure. In our experience, only a small portion of code and data in a program is security sensitive and will belong in a security method, which simplifies the task of auditing security-sensitive code. This design also limits the amount of work the VM must do to enforce DIFC.

Dynamic DIFC Enforcement with Barriers. The VM inserts *barriers* that ensure no reference outside a security method reads or writes labeled data, and all references inside a security method follow the DIFC rules. A barrier is a snippet of code the VM executes before every read and write to an object and is a standard implementation feature in VMs for garbage-collected languages [Blackburn and Hosking 2004]. The Laminar VM inserts barriers at every object read and write. Outside the security method, the barrier throws an exception if the program tries to read or write data with a nonempty secrecy or integrity label. Inside a security method, every time the program reads or writes objects or kernel resources, the barrier checks that the information flow follows the policies specified by the *current* labels of the security method.

For example, an assignment `w=r` inside a security method `M` is safe if and only if the information flow from `r` to `w` is legal for the thread inside `M`. Note that the Laminar library API (Table I) does not include a routine for adding labels to a thread. In order to add labels, threads must start a security method.

Security methods have the added benefit that they make the DIFC implementation more efficient because the barrier checks outside a security method are simpler than checks inside a security method. Outside a security method, the barrier simply checks if data have nonempty labels and throws an exception if they do, since any access to secure data is forbidden. Inside a security method, the DIFC barriers must compare DIFC labels of references to ascertain if the information flow is legal.

In summary, security methods make it easier for programmers to add security policies to existing programs. They make it easier for programmers to audit security code. They limit the effects of implicit information flows (Section 6.4), and they make the implementation more efficient.

Inputs, Return Values, and Container Objects. Security methods have two default parameters: a secrecy label and an integrity label. A declassifier or endorser includes a third parameter: the capability set. Security methods may take other parameters as inputs and/or return an output value so long as the input or return is a valid information flow. These input and output values may be primitives (`int`, `boolean`, etc.) or object references. Generally speaking, security methods with a nonempty secrecy label cannot return a value, and security methods with a nonempty integrity label cannot read inputs without being wrapped in an endorser. Section 5.2 details the specific rules.

A key abstraction in Laminar that improves programmability is the stylized use of *container* objects. The programmer allocates objects with secret labels outside of a secret security method by invoking `new` and passing the appropriate labels. Labeled object creation is explained further in Section 5.2. The program then passes this object to security methods. Each security method may update the contents of the object, but outside of the security method, the object's contents and any modifications are opaque. Code outside of a security method may not dereference references to objects with nonempty labels.

Example. Figure 2 presents the calendar example from the introduction. A calendar server calls code provided by Bob that reads the `Calendar` object belonging to Alice,

```
public class MeetingScheduler {
     Calendar AliceCal; // has labels ⟨S(a), I()⟩
     Calendar BobCal; //  has labels ⟨S(b), I()⟩
     // Alice Calendar file "alice.cal" has labels ⟨S(a), I(i)⟩

     void ScheduleMeeting() {
         CapSet C = getCurrentCapabilities(); // C =  (a^+, b^+, b^-, i^+)
         CapSet CapDeclassify = C.minus(i^+);
         CapSet CapEndorse = C.minus(b^+, b^-);

         // Create a Meeting object with a secrecy label, to act as a security container
[L1]     Meeting Mtg = new Meeting (S(a), I(), C(a^+)) ();

         // Bob computes a mutually agreeable meeting time, declassifies to Alice
[L2]     Mtg.BobFindMeetingTime(S(a,b), I(), CapDeclassify) (AliceCal, BobCal);

         // Alice's endorser
[L7]     Mtg.AliceCheckMeeting (S(a), I(), CapEndorse) ();
     }
}

public class Meeting {
     MeetingTime val;  // class variable

     secure (Label S, Label I, CapSet c)
     final void BobFindMeetingTime (Calendar Other, Calendar Bob) {
[L3]      MeetingTime tmp = getMeetingTime(Other, Bob);

          // tmp now has labels ⟨S(a,b), I()⟩
[L4]      Label newS = S.minus(S(b));
[L5]      if ( /* bob's tests */ ) { BobDeclassify (newS, I, c) (tmp); }
     }

     secure (Label S, Label I, CapSet c)
     final void BobDeclassify (MeetingTime t) {
[L6]      this.val = Laminar.copyAndLabel(t, newS, I);
     }

     secure (Label S, Label I, CapSet c)
     final void AliceCheckMeeting () {
[L8]      Label AliceFileIntLabel = I.union(I(i));

          // Check that the MeetingTime is well-formed
[L9]      if (AliceIntegrityCheck(this.val) {
[L10]          this.val = Laminar.copyAndLabel(this.val, S, AliceFileIntLabel);
          }

          // this.val now has labels ⟨S(a), I(i)⟩
[L11]     AliceWriteCalendar(S, AliceFileIntLabel) ();
     }

     secure (Label S, Label I)
     void AliceWriteCalendar () {
[L12]     FileOutputStream AliceCalFile = new FileOutputStream("alice.cal");
          // Calculate offset into the file
[L13]     AliceCalFile.Write(this.val, offset, this.val.length);
[L14]     AliceCalFile.Close();
     }
}
```

Fig. 2. Example security methods that read and write a secure calendar. Bob provides the first two security methods. `BobFindMeetingTime` executes with both Alice and Bob's secrecy tags (a and b, respectively). This method selects a meeting time such that Alice and Bob are both available and places it in a `MeetingTime` object with label $\langle S(a, b), I()\rangle$. `BobDeclassify` then removes Bob's secrecy tag (b). The calendar application then executes Alice's endorser (`AliceCheckMeeting`), which checks that the `MeetingTime` object is well-formed, and then adds the i integrity tag and writes the meeting time to a file with label $\langle S(a), I(i)\rangle$. The label on Alice's calendar ensures both secrecy of her calendar data, as well as that all updates have been checked by trusted code. Execution order is indicated with LX, where X is the line number if the code were inlined into `ScheduleMeeting`.

creates a `Meeting` object, exports the meeting time to Bob, and writes the meeting into Alice's calendar file. The code begins at L1 by allocating a container object, `Mtg`, which is passed to subsequent security methods to store the secret meeting time. When the thread enters the security method `BobFindMeetingTime` to read the calendars (line `L2`), the VM sets the thread's secrecy label to $S(a, b)$ and therefore the thread can read secret `Calendar` data guarded by tags a and b. The code that reads each calendar (line `L3`) is a valid information flow because the thread's labels are more restrictive than either Calendar's labels (e.g., $\langle S(a, b), I()\rangle \supseteq \langle S(a), I()\rangle$). The thread has the capability $C(b^-)$ to declassify tag b, which is used to enter the security method `BobDeclassify`, at line `L5`. Entering the nested declassifier at line `L5` may be conditioned on additional checks to prevent information leaks. Before writing the meeting time into Alice's calendar file, the thread must acquire integrity label $I(i)$ by calling an endorser function, `AliceCheckMeeting`, which checks that the data to be written meet Alice's invariants, such as prohibiting conflicting meeting times.

The VM inserts barriers that check the information flow safety at every object read and write. Locals are limited to method scope and implicitly have the same label as the security method. The code at Line `L3` computes the common meeting time and stores it in the container object referred to by `Mtg`. For instance, the read barrier code tests if reading fields of objects `Other` and `Bob` are valid information flows and whether the writes into the newly created `Meeting` object are legal. The writes into the `Meeting` object are legal because the object has the same secrecy label as the thread in the security method at that point. A nested security method declassifies the meeting to Alice (`L5`/`L6`), updating the meeting time in `this.val`. By replacing the object referenced by `this.val` with a copy with a lower secrecy level (Line `L6`), this code effectively removes the tag b from the output. Copying and relabeling `tmp` at `L6` is legal because the method has the b^- capability and can declassify data protected by the secrecy tag b. Notice that if line `L6` performed `copyAndLabel(tmp,` $S(), I()$`)` to remove all tags from the secrecy label, the VM would throw an exception because when the thread is in the security method, it does not have the a^- capability and cannot remove the a tag from the data. In this example, the kernel checks the file operations in line `L12` and `L13` that write to Alice's calendar file, and the VM checks the other operations on application data structures.

Security Method Initialization. Laminar enforces the following rules when a thread enters a security method. Let S_R, I_R, and C_R be the secrecy label, integrity label, and capability sets of a security method, 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 methods. P could, therefore, already be inside a security method when it enters R. When the thread P enters the security method R, the VM ensures that the following rules hold:

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

$$(I_R - I_P) \subseteq C_p^+ \text{ and } (I_P - I_R) \subseteq C_p^- \tag{2}$$

$$C_R \subseteq C_P \tag{3}$$

The first two rules state that, in order for principal P to enter method R, P must have the required capabilities to change its labels to R's labels. The third rule states that the principal P can only retain a subset of its current capabilities when it enters a security method. While the security method executes, the sets associated with P change to S_R, I_R, and C_R.

These rules encapsulate the common-sense understanding that a parent principal, P, has control over the labels and capabilities it passes to a security method and that the VM will prevent the principal from creating a security method with security properties that the principal itself lacks. The rules also state that security methods nest in the natural way based on the labels and capabilities of the thread entering the nested method.

3.4. VM and OS Interface

Our design trusts the Laminar VM and OS kernel for DIFC enforcement. The VM is trusted to enforce DIFC policies on application data structures and implements security methods without kernel involvement. The kernel is responsible for enforcing DIFC rules on OS kernel abstractions, such as files and pipes. If a security method does not perform a system call, the VM does all the enforcement and does not involve the kernel. For example, if code inside a security method with secrecy tags tries to write to a public object, the VM will throw an exception that will end the security method. As an optimization, the VM does not notify the kernel of changes to the thread's labels until the VM needs to issue a system call on behalf of the application. The kernel enforces DIFC rules on each system call according to the thread's labels and the labels of any other objects involved (e.g., writing data to a file or the network). For standard system calls, such as `read`, the labels of the thread and file handle are implicit system call arguments. The VM communicates security metadata to the kernel via the Laminar system calls (Table II). For instance, the VM changes the labels on the current application thread (embodied as a kernel thread) executing within a security method using the `add_task_tag` system call. The kernel ensures that the labels are legal given the thread's capabilities.

Acquiring Tags and Capabilities. Principals (threads) in Laminar acquire capabilities in three ways. They allocate a new tag, they inherit them through `fork()`, or they perform interprocess communication. A thread working on behalf of one user may call security methods provided by another user; for instance, Alice's thread may call Bob's declassifier with the capability to read Alice's calendar. Another thread running on Bob's behalf can only acquire Alice's capability if Alice shares it over an IPC channel. The system carefully mediates capability acquisition lest a principal incorrectly declassify or endorse data. Laminar assumes a one-to-one correspondence between application and kernel threads. Application threads use the Laminar language API, which in turn invokes the system calls for managing tags and capabilities.

A principal allocates a new tag in the kernel via the `alloc_tag` system call, which is used to implement the language API function `createAndAddCapability`. As a result of the system call, the kernel security module will create and return a new, unique tag. 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. A thread explicitly selects which capabilities it will pass to a security method, and the trusted VM can temporarily remove the capability from the thread using the `drop_capabilities` system call.

Threads and security methods 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 method, the thread retains only the subset of its capabilities specified by the method. In general, when a new principal is created, its capabilities are a subset of its immediate parent, which the VM and kernel enforce.

The passing of all interthread and interprocess 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 methods. When a thread enters a security method, the VM first makes sure that the thread has sufficient capabilities to enter the method. If the thread may enter the security method, the VM sets the labels and capabilities of the thread to equal those specified by the security method. The VM sets the thread's capabilities to the empty set when it enters a security method that is not passed capabilities (i.e., not a declassifier or endorser). Similarly, when the thread exits the security method, the VM restores the labels and capabilities the thread had just before it entered the method. On exiting a nested security method, the VM restores the labels and capabilities of the thread to those of the parent security method.

The Laminar language API provides a method, `removeCapability`, that removes a thread's capability in the VM, preventing use as an argument to a later security method. To prevent threads from using the capability set as a covert channel, capabilities must be created and removed outside of a security method (Section 6.6). The `removeCapability` VM call uses the `drop_capability` system call to drop the capability from the OS kernel thread.

Similarly, if a security method issues a system call, the VM first invokes the `add_task_tag` or `remove_task_tag` system calls to change the thread's labels in the OS kernel. As an optimization, the VM postpones setting the thread's labels in the kernel until just before the first system call and at the end of the security method. This system call has no user API; it is used solely by the VM.

The Laminar VM prohibits security methods from changing their labels; labels stay the same throughout the security method to prevent leaks through local variables (Section 5.2). Labels are stored as opaque objects that cannot be enumerated. To change labels in the middle of a security method, a thread begins a nested security method.

Consider an example when a thread only has the a^+ capability and starts a security method with secrecy label $\langle S(a)\rangle$. The Laminar VM sets the secrecy label of the thread to $\langle S(a)\rangle$ when the security method begins. When the security method ends, the VM forces the thread to drop the secrecy label, even if it does not have the a^- capability. To drop $\langle S(a)\rangle$ from a thread, the VM contains a high-integrity thread, running with a special integrity tag called `tcb` that is trusted by the kernel. 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 kernel 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 kernel 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 identifier space, tag exhaustion is not a concern.

3.5. Security Discussion

The Laminar OS mediates information flow on OS resources, such as files and pipes. The Laminar JVM mediates information flows within the application using barriers, by restricting the programming model, and constraining how data enter and leave a security method. Implicit flows are mediated by masking the control flow within the security method (Section 6.4). Updating the capability set of a thread is treated as a public write, preventing covert channels through this abstraction. We allow security methods to execute concurrently. Our threat model assumes that security methods will terminate and will not leak information through timing channels, including the execution time of a security method; Sections 6.4 and 6.5 discuss techniques that could be adopted in a production Laminar deployment to uphold these assumptions.

Although the security method design facilitates incremental adoption because threads manage capabilities, this choice unfortunately places a measure of trust in the code that calls security methods. To limit the risk of capability mismanagement, only security methods that endorse or declassify data are passed capabilities, and these methods must be declared `final`. Section 6.6 discusses this issue in more detail.

3.6. Labeling Data

The VM labels data objects at allocation time to avoid races between creation and labeling. The VM labels objects allocated within a security method with the secrecy and integrity labels of that method. The `create_file_labeled` and `mkdir_labeled` kernel calls create labeled files and directories. Other system resources use the labels of their creating thread.

Similar to most other DIFC systems, Laminar uses immutable labels. To change a label, the user must copy the data object. Section 5.4 discusses implementation details and the interaction of object labels with the Java memory model. 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 relabeling 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. Laminar currently supports only immutable labels on files. It may be possible to safely relabel files using additional synchronization.

3.7. 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 memoizes its result in a security method with one label, a later call with a different label may attempt to return the memoized value. Because the memoized result is secret, Laminar will prevent the attempt to return it. Programmers or the VM must modify such code to work in any DIFC system.

3.8. Trusted Computing Base

To implement Laminar, we added approximately 2,000 lines of code to Jikes RVM [Alpern et al. 2000],[2] added a 1,000-line Linux security module, and modified 500 lines of the Linux kernel. This relatively small amount of code means that Laminar can be audited.

The Laminar design does not trust `javac` to enforce information flow rules but does trust `javac` to provide valid bytecode that faithfully represents the Java source. Jikes RVM does not include a bytecode verifier—a feature of a secure, production VM that should reject malformed bytecode.

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 (Section 3.3) for all accesses and optimizes them

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

correctly. Read and write barrier insertion is standard in many VMs [Blackburn and Hosking 2004]. In Linux, Laminar assumes that the kernel has the proper mechanisms to call into Linux Security Modules (LSM) [Wright et al. 2002]. Because many projects rely on LSMs, the Linux code base is under constant audit to make sure all necessary calls are made.

4. OS SUPPORT TO CONTROL INFORMATION FLOW

We have implemented support for DIFC in Linux version 2.6.22.6 as an LSM [Wright et al. 2002]. LSM provides hooks into the kernel that customize authorization rules. We added a set of system calls to manage labels and capabilities, as listed in Table II. Some LSM systems, such as SELinux [Loscocco and Smalley 2001], manage access control settings through a custom filesystem similar to `/proc`. A custom filesystem is isomorphic to adding new system calls. The Laminar security module contains about 1,000 lines of new code, and we modified about 500 lines of existing kernel code to implement the Laminar OS.

Tags, Labels, and Capabilities. Tags are represented by 64-bit integers and allocated via the `alloc_tag()` system call. The OS stores labels and capabilities for system resources in the opaque security field of the appropriate Linux objects (e.g., `task_struct`, `inode`, and `file`). The OS persistently stores secrecy and integrity labels for files in the files' extended attributes. Most of the standard local filesystems for Linux support extended attributes, including `ext2`, `ext3`, `xfs`, and `reiserfs`. A mature implementation of Laminar could adopt a similar strategy to Flume for filesystems without extended attributes, encoding a label identifier in the extra bits of the user and group identifier fields of a file's `inode` [Krohn et al. 2007].

Files. Using LSM, Laminar intercepts `inode` and `file` accesses, which perform all operations on unopened files and file handles, respectively. The `inode` and `file` data structures are used to implement a variety of abstractions, such as sockets and pipes. The Laminar security hooks perform a straightforward check of the rules listed in Section 2.2. The secrecy and integrity labels of an `inode` protect its contents and its metadata, except for the name and labels, which are protected by the labels of the parent directory.

For instance, if a process with secrecy label $\langle S(a)\rangle$ tries to read directory `foo` with the same secrecy label, the process will be able to see the names and labels of all files in `foo`. If file `foo/bar` has secrecy label $\langle S(a, b)\rangle$, any attempt to read the file's attributes, such as its size, will fail, as size of the file could otherwise be used to leak information about the file's contents.

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 prevent a principal with secrecy label $\langle S(a)\rangle$ from creating a file with secrecy label $\langle S(a)\rangle$ 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.

A principal may use the newly introduced `create_labeled` and `mkdir_labeled` system calls to create a file or directory with secrecy and integrity labels different from the principal's current labels. Informally, a principal may create a differently labeled file if its current labels permit reading and writing the parent directory, and it has capabilities such that it can change its labels to match the new file. More formally, we allow a principal with labels $\langle S_p, I_p\rangle$ to create a labeled file or directory with labels $\langle S_f, I_f\rangle$ if (1) $S_p \subseteq S_f$ and $I_f \subseteq I_p$, (2) the principal has capabilities to acquire labels $\langle S_f, I_f\rangle$, and

(3) the principal can read and write the parent directory with its current secrecy and integrity label. This approach prevents information leaks during file creation while maintaining a logical and useful 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 elides this problem with hierarchical directory traversal and simplifies application-level data storage with integrity labels [Krohn et al. 2007].

Applying integrity labels to a traditional Unix directory structure brings out a fundamental design tension in DIFC OSes 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 article. 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.

Pipes. Laminar mediates Interprocess 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 labels of the pipe. Message delivery over a pipe in Laminar is unreliable. An error code due to an incompatible 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 [Krohn et al. 2007; Vandebogart et al. 2007]. Linux does not include LSM hooks in the pipe implementation; Laminar adds LSM hooks to the pipe implementation in order to mediate reads and writes to pipes.

To prevent illegal information flows in Laminar, a pipe does not deliver an end-of-file (`EOF`) notification when the writer exits or closes the pipe if the writing thread cannot write to the pipe at the time it exits. This lack of termination implies that, if a process exits inside of a security method, the JVM must ensure that the thread's label is visible to the kernel (Section 3.4) before issuing an `exit` system call, so that the appropriate policies are applied when the OS closes the open file descriptors. Thus, Laminar, like many OS DIFC implementations, only delivers `EOF` notifications if writing the notification constitutes a legal flow.

Thus, the practical implication of unreliable delivery and eliminating `EOF` notification is that reads from a pipe should be nonblocking. Otherwise, an application may hang waiting for an `EOF` notification. In the common case where all applications in a pipeline have the same labels, 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.

Network Sockets and Other IPC. The Laminar OS prototype treats network sockets and other IPC channels as having empty secrecy and integrity labels. Thus, input from the network must be read by code with empty secrecy and integrity labels, and the data must be labeled in a security method that validates the input. Managing information flows across systems is beyond the scope of this work, but has been addressed in other systems including DStar [Zeldovich et al. 2008]. The `inodes` associated with other Linux IPC abstractions, such as System V IPC, could be labeled similarly to pipes but would likely require additional analysis of any potential information flows resulting from idiosyncratic behavior.

5. JAVA VM SUPPORT TO CONTROL INFORMATION FLOW

We implement Laminar's trusted VM in Jikes RVM 3.0.0,[1] a well performing Java-in-Java VM [Alpern et al. 2000]. Our Laminar implementation is publicly available on the Jikes RVM Research Archive and on GitHub.[3] All subsequent uses of the term VM refer to the Laminar-enhanced implementation in Jikes RVM.

When a thread starts a security method, the VM inserts a check that determines if the thread has the capabilities to initialize the security method with the specified labels and capabilities, as described in Section 3.3. Thread capabilities are stored in the kernel. The VM caches a copy of the current capabilities of each thread to make the checks efficient inside the security method.

The VM enforces information flow control for accesses to three types of application data: objects, which reside in the heap; locals, which reside on the stack and in registers; and statics, which reside in a global table.[4] This section describes how the VM enforces the DIFC rules on objects, local variables, and static variables.

5.1. Controlling Information Flow on Objects

The VM interposes on every read and write to an object or static by transparently adding *barriers* before the operation. Barriers are not visible to the programmer and cannot be avoided, thus creating a natural point to mediate explicit data flows. The VM uses barriers to ensure that all accesses to data with nonempty labels occur within a security method and that references inside a security method conform to the DIFC rules in Section 2.

Heap Objects. The VM tracks information flow for labeled heap objects. When an object is allocated, the VM assigns immutable secrecy and integrity labels to the object. We modify the allocator to take secrecy and integrity labels as parameters; the allocator adds two words to each object's header, which point to secrecy and integrity labels. The VM assigns objects allocated inside security methods the labels of the method at the allocation point. 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 2). The VM allocates labeled objects into a separate *labeled object space* in the heap, which we exploit to optimize the instrumentation that checks whether an object is labeled or not.

Each object acts as a *security container* for its fields, and the object's labels protect the fields from illegal access. The Laminar prototype requires that all fields of an object have the same labels. For example, consider an object pointed to by the reference `o`. The object has two fields, primitive integer `x` and reference `y`. When the program reads or writes `o.x` or `o.y`, the VM enforces DIFC rules based on the labels of the object referenced by `o`. If the program has labels that allow it to read the object referenced by `o`, then it may read or copy `o.x` and `o.y`. However, the object that `o.y` references may have the same or different labels. Thus, the programmer may assign distinct labels

[3]http://www.jikesrvm.org/Research+Archive and https://github.com/ut-osa/laminar.

[4]Although objects and nonvolatile statics may be register-allocated and nonescaping objects can be scalar-replaced, objects appear to be in the heap and statics appear to be in the global table when the VM compiler adds the needed barriers.

to the object referenced by `o.y`. If the application performs the dereference `o.y.foo`, the VM must verify that the security method may read and dereference the reference `o.y` based on the labels of the object referenced by `o` and then separately check the read of reference `foo` based on the labels of the object referenced by `o.y`. The security container model simplifies the task of labeling objects at allocation time, which is easier for programmers to reason about and cheaper for the VM to enforce compared to labeling individual object fields.

Labels. Applications do not have direct access to labels on data or principals, which are used internally by the VM to enforce DIFC rules. Recall that a label may contain one or more tags.

The Laminar API provides two functions that return a label. The functions return the label in an immutable, opaque object of type `Label`. The instantiations of `Label` support operations such as `isSubsetOf()`, `minus()`, and `union()`. The function `createAndAddCapability` invokes the `alloc_tag` Laminar OS system call, which creates a new tag and adds the associated capabilities to the current thread, and returns a `Label` object containing the single new tag to the application. The `getCurrentLabel()` function returns the secrecy or integrity label of the enclosing security method.

For efficiency, `Label` objects may be safely shared by objects, security methods, and threads because they are immutable; operations such as `minus()` and `union()` return a new object instead of modifying an existing `Label`. `Label` objects are not used internally by the VM for DIFC enforcement. Internally, the VM implements `Label` as a sorted array of 64-bit integers to hold tags. Because a `Label` object is opaque, applications cannot observe the individual values of the tags. Moreover, because object labels are immutable, any attempt to change the labels on an object requires writing a reference to the new object somewhere, which is an explicit, regulated information flow. Thus, a program cannot create a covert channel by creating a `Label` with irrelevant tags.

Similar to any other object, the VM associates secrecy and integrity labels with the instances of `Label`. An application may create a `Label` object using the `new` keyword or by using trusted Laminar API functions. When a `Label` object is created, it has the secrecy level of the thread at the time it was created. The integrity level of the `Label` object depends on which function created it: `Label` objects created by `new` also have the integrity of the thread at the time of creation, whereas `Label` objects created by the Laminar API functions are given the highest integrity ($\top$, representing the set of all possible integrity tags) because we trust the API and the VM. In general, `Label` objects have high integrity and empty secrecy and can be used as parameters to any security method.

VM Instrumentation. To enforce DIFC rules, the VM's compiler inserts barrier instrumentation just prior to every read and write in the application (Section 3.3). Inside security methods, the compiler inserts barriers at a *labeled object allocation* (before the compiler invokes the application's constructor) that sets the labels. It inserts barriers at every *read from* and *write to* an object field or array element. Inside security methods, barriers load the accessed objects' secrecy and integrity `Labels` and check that they conform to the current security method's labels and capabilities. Outside security methods, read and write barriers check that the accessed objects are *unlabeled* (i.e., have empty secrecy and integrity labels).

The compiler inserts different barriers depending on whether the access occurs inside or outside a security method. If a method is called both from inside and outside security method contexts, the compiler will produce two versions of the method. In our prototype implementation, when a method first executes, the JVM invokes the compiler, and it checks whether the thread is executing a security method 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 outside a security method. Thus, we also support a configuration in which the compiler adds *dynamic barriers*. The barriers check whether the current thread is in a security method or not and then execute the correct barrier. A production implementation should use cloning to compile two versions of methods called from both contexts, and each call site can call the appropriate version based on context. (Some software transactional memory implementations use a similar approach [Ni et al. 2008].) Because which version to call is statically knowable at each compiled call site, the overhead one would attain with a method cloning implementation should match what we measure for static barriers.

Because object labels are immutable, and security methods cannot change their labels, repeating barriers on the same object is redundant. We implemented 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 this optimization is intraprocedural, the VM's dynamic optimizing compiler inlines small and hot methods by default, thus increasing the scope of redundancy elimination.

Example. Figure 3 computes the sum of the grades obtained by two different students. The `student1` and `student2` objects are labeled and have different secrecy values associated with them. Once the security method starts, the VM assigns the thread the secrecy and integrity labels specified by `S` and `I`, respectively. Lines `L2` and `L3` read labeled objects and result in a security exception if the flow from `student1.grades` or `student2.grades` to the thread in the security method is not allowed. Line `L4` stores the value in a new labeled `Integer` object and stores the reference in the labeled `avgHolder` object. At lines `L5`–`L6`, the thread calls a declassifying security method, passing it the capability to add and remove the secrecy tags by making an unlabeled copy of the `avgHolder.value` object. If the `CapSet` passed to the security method is not a subset of the current thread's capabilities, then the program throws a security exception at `L5`, which the end of the security method may catch; this is followed by returning from the security method. Security exceptions are a category of Java language exceptions and may be caught by the security method author. The VM does not propagate exceptions out of a security method (Section 6.2). Because the declassifier runs with an empty label, it may assign the new reference into the unlabeled `outputHolder.value` field. In practice, a declassifier such as `declassifyAverage` would be nested inside a security method with a nonempty secrecy label that first checked the potential output, as in Figure 2, and the application of rules in the VM would be similar.

5.2. Restricting Information Flow for Locals and Parameters

Laminar does not track labels on local variables because they cannot be used outside the scope of the current method, thus precluding an information flow to or from a security method. Laminar assumes that locals have the secrecy and integrity labels of the enclosing security method or empty labels outside of a security method. All security methods take as input two parameters: the secrecy label and integrity label. Declassifiers and endorsers may take a third parameter: the capability set. For clarity, these are indicated in examples with separate argument parentheses on the `secure` keyword.

If the explicit flow is legal, security methods in Laminar can accept additional inputs and return outputs of primitive values (`int`, `boolean`, etc.) and references, which are passed-by-value. A security method with a nonempty integrity label may only accept input if the calling function is also in a security method with higher integrity or the capability to add all missing integrity tags (i.e., an endorser). A security method with

```
        // threadCaps = C(s_1^+, s_1^-, s_2^+, s_2^-)
        // S_T is the thread's current secrecy label, initially empty
        // I_T is the thread's current integrity label, initially empty
        // Label S_Empty = S(), I_Empty = I()
                                               {VM operations}
   [L1] secure (Label S, Label I)              {? changeLabel(S_T, S, threadCaps) }
        void computeAverage                    {? changeLabel(I_T, I, threadCaps) }
        (IntHolder avgHolder) {                {save S_T, I_T, threadCaps}
                                               {S_T = S}
                                               {I_T = I}
                                               {threadCaps = C()}
   [L2]   int m1 = student1.grades;            {? secrecy-label(student1) ⊑ (S_T = (s_1,s_2))}
                                               {? (I_T = ()) ⊑ integrity-label(student1)}
   [L3]   int m2 = student2.grades;            {? secrecy-label(student2) ⊑ (S_T = (s_1,s_2))}
                                               {? (I_T = ()) ⊑ integrity-label(student2)}
          int avg = (m1 + m2) / 2;
   [L4]   avgHolder.value = new Integer(S, I)(avg);
                                               {? S_T ⊑ secrecy-label(new Integer)}
                                               {? integrity-label(new Integer) ⊑ I_T}
                                               {? S_T ⊑ secrecy-label(avgHolder)}
                                               {? integrity-label(avgHolder) ⊑ I_T}
           return;                             {restore S_T, I_T, threadCaps}
        }

   [L5] secure (Label S, Label I, CapSet C) {? C ⊆ threadCaps}
        final void declassifyAverage           {? changeLabel(S_T, S, C) }
        (IntHolder avgHolder,                  {? changeLabel(I_T, I, C) }
         IntHolder outputHolder) {             {save S_T, I_T,  threadCaps}
                                               {S_T = S}
                                               {I_T = I}
                                               {threadCaps = C}
   [L6]   outputHolder.value =                 {? S_T ⊑ secrecy-label(outputHolder)}
                                               {? integrity-label(outputHolder) ⊑ I_T}
            Laminar.copyAndLabel               {? changeLabel(S_T,
              (AvgHolder.value,                               secrecy-label(avgHolder.value),
               S_Empty, I_Empty);                             C)}
                                               {? changeLabel(I_T,
                                                              integrity-label(avgHolder.value),
                                                              C)}
                                               {? changeLabel(secrecy-label(avgHolder.value)
                                                              S_Empty, C)}
                                               {? changeLabel(integrity-label(avgHolder.value)
                                                              I_Empty, C)}
           return;                             {restore S_T, I_T,  threadCaps}
        }

        // Label S == S(s_1,s_2), I == I();
        // CapSet C_Declassify = C(s_1^+, s_1^-, s_2^+, s_2^-)
        IntHolder avgHolder = new IntHolder (S, I) (); // labeled
        IntHolder outputHolder = new IntHolder (); // unlabeled
[L1-L4] computeAverage(S, I) (avgHolder);
[L5-L6] declassifyAverage(S_Empty, I, C_Declassify)) (avgHolder, outputHolder);
        System.out.println("Average is " + outputHolder.value.toString());
```

Fig. 3. Securely computing the average grades of two students. The `student1` and `student2` objects are labeled. The object `credentials` contains the secrecy, integrity, and capabilities sets with which the security method is initialized. The statements on the right side are the checks that are performed by the VM. The symbol ? indicates an assertion, $\sqsubseteq$ indicates an information flow check, and the internal function `changeLabel(Label to, Label from, CapSet caps)` checks whether a label change would be permitted given the input capabilities (caps).

an empty integrity label may read any input. Similarly, a security method may only return a value if it has an empty secrecy label or the value is returned to a more secret security method. Because declassifiers tend to be nested, most declassification examples write the output to an object or security container with an empty secrecy label that is passed as input to the method.

To enable containers for secure data, Laminar permits creation of objects with an empty integrity label and nonempty secrecy label outside of a security method. By creating objects with a nonempty secrecy label outside of a security method, the creation itself and the return of the reference cannot be dependent upon any secret information and thus cannot create any information flow. Once the initial secret object reference is created, it can be passed to multiple security methods that operate on secret data, acting as a container for secret data. Note that the first security method the reference is passed to is the object's *constructor*. If a labeled constructor throws an exception, `new` must still return the labeled but uninitialized object. Because critical regions with an empty secrecy label but a nonempty integrity label can return a value, allocation of objects with integrity tags can always be wrapped in an endorser without any secrecy tags. The endorser may allocate an object with both secrecy and integrity tags in its label so long as it drops its secrecy label before returning the object.

Because a method with a nonempty secrecy label cannot return a value, the security container abstraction serves as a means to facilitate passing secret, intermediate values among security methods. The security container abstraction also neatly integrates with common Java patterns of using the implicit `this` input parameter. In other words, security methods may construct container objects and operate on them, as illustrated in Figure 2.

5.3. Static Variables

Static (global) variables in the Laminar prototype have empty secrecy and integrity labels. By inserting barriers at static variable accesses inside security methods, security methods with an empty secrecy label may write static variables, and security methods with an empty integrity label may read static variables.

We expect that a production implementation could support nonempty labels on statics with modest overhead because static accesses are relatively infrequent compared to field and array element accesses. Good security programming practices, like general-purpose programming practices, recommend sparse use, if any, of statics. We did not find this functionality necessary, and none of the applications in Section 9 needed labeled static variables.

5.4. Instantiating Labels

Some care must be taken when creating objects to prevent race conditions between assigning the object label and concurrent attempts to dereference the object. In the Laminar implementation, the label fields of each object are hidden from the programmer (VM-internal) and are assigned between object allocation and calling the object's constructor. From the perspective of the Java memory model [Manson et al. 2005; Pugh 2005], the label fields should be treated similar to `final` fields. In the Java memory model, `final` fields are visible to all threads before the constructor returns. The VM must prevent reordering these assignments outside of the constructor, and the constructor writer must not make external assignments of the `this` object. In order to protect against a malicious constructor writer, a production Laminar VM would strengthen this requirement slightly: The label assignments must be visible to all threads before the constructor is called.

6. SECURITY IMPLICATIONS AND INFORMATION FLOW ENFORCEMENT MECHANISMS

This section summarizes the major classes of information flows that Laminar mediates and the security implications of the Laminar design. This section pays particular attention to changes in the programming model introduced by Laminar, including security methods and thread capabilities. This section also discusses security issues that are

Table III. Laminar's Programming Requirements and the Attacks They Prevent

Requirement	Attack Prevented
Explicitly labeled objects in the JVM and OS.	Illegal explicit information flow through objects.
Restrict information flow through explicit function arguments and return values.	Illegal explicit information flow through arguments and return values. Prevents information flow through locals, which are out of scope in a security method.
Static fields have empty security and integrity labels.	Illegal explicit information flows through static fields.
A security method may only have one exit point, including exceptions. All exceptions will be caught at the end of a security method.	Implicit information flows based on security method control flow.
A security method will execute for a fixed amount of time (not implemented).	Limits the bandwidth of timing and termination channels, which would otherwise be increased by multithreaded synchronization.
Dropping or creating a capability is treated as a write to the thread's capability set and requires an empty secrecy label.	Prevents information flow through the thread's capability set.
A security method that takes a capability set as a third parameter must be declared `final`.	Prevents passing capabilities to unintended functions via inheritance.

not addressed in the Laminar prototype and how subsequent research could mitigate these concerns. This section connects implementation details described previously in Sections 4 and 5 with the system's security properties. Table III summarizes the key programming abstractions and requirements that Laminar places on the programmer and the attacks they prevent, all of which are discussed later in more detail.

In each example and figure in this section, we use the following notation for labels. The value of a secrecy label with tags a and b is represented as $S(a, b)$. In Java, this label is stored in a `Label` object. Similarly, an integrity label with tag i is represented $I(i)$. Finally, a capability set with the ability to add a and remove i is represented as $C(a^+, i^-)$.

6.1. Explicit Information Flows

An explicit information flow occurs when a program moves data from one variable to another or from program memory into an OS-managed data sink, such as a file. The Laminar JVM and OS kernel collaborate to track explicit information flows and prevent illegal information flows. This subsection reviews the strategy for each major programming abstraction and provides backward references for the implementation details.

OS abstractions (Section 4). Laminar extends the Linux 2.6.22.6 kernel with an LSM that adds secrecy and integrity labels to a `task` (OS-visible thread) and file `inodes`, which include most IPC abstractions, such as pipes. The Laminar LSM interposes on all file handle reads and writes to validate the flow, as well as other system calls such as creating files and directories. We extend the Linux kernel with a few additional system calls and security hooks.

Java objects (Section 5.1). Objects in the Laminar VM are explicitly labeled, and the VM checks the labels of an object before all reads from and writes to an object field. The Laminar VM extends Jikes RVM, which provides *barriers* that interpose each object read and write.

Local variables (Section 5.2). Laminar does not label or track information flows through local variables. Because labeled data are accessed in security methods, locals

```
// Object o has labels ⟨S(h), I()⟩
//   and members
// L has labels ⟨S(), I()⟩
// Invariant: y == 2/x
static boolean L = false;
secure  (Label S = S(h), Label I = I())
void explicit (Object o) {
  o.x++;
  L = o.H;
  o.y = 2 / o.x;
  ...
} catch (ArithmeticException e) {
  o.y = 2;
  o.x = 1;
} catch (...) {
  o.y = 2 / o.x;
}
```

Fig. 4. Catch blocks handle illegal flows. Programmer may handle security exceptions separately from other runtime errors (e.g., divide by zero). Label and capability values are inlined for clarity.

in an untrusted parent are out of scope inside the security method and vice versa. With additional static analysis on information flow through locals, one could safely implement security methods as arbitrary, lexically scoped blocks within a method, as originally proposed [Roy et al. 2009], but we found the implementation was much more complex.

Arguments and return values (Section 5.2). Laminar permits primitives and references as input values to a security method as long as reading the input values would not violate an integrity rule (e.g., no read down). Note that even object references are passed by value in Java, so manipulating any input variables will not affect a local in the calling frame. The VM will mediate all accesses to an object with barriers. Similarly, the programming model is restricted such that a security method may only return a value in the calling context if the write would not violate a secrecy rule (no write down).

In the case of nested security methods, a more secret calling method may pass an input to a less secret inner security method if the outer method has appropriate declassification capability. Similarly, a higher integrity method may accept input from a lower integrity parent if the outer method has the appropriate endorsement capability. These rules for nested security methods are necessary to facilitate declassification and endorsement.

Static variables (Section 5.3). The Laminar prototype treats all static fields as having empty labels. The Laminar VM interposes on all static field accesses and prevents illegal information flows to statics. In general, static fields are used infrequently, and our application case studies did not require nonempty labels for static variables.

6.2. Handling Illegal Flows

When code in a security method attempts an illegal explicit information flow, the VM creates an exception that transfers control to the end of the security method. As a programmer convenience, the security method may catch exceptions in order to restore program invariants. Any exceptions uncaught by the programmer will be caught by the VM before the security method ends, thus hiding the control flow of the security method from the caller.

For example, the code in Figure 4 shows an illegal explicit flow. The code attempts to copy and thus leak the value of secret variable `o.H`, which it may not declassify, to the static, nonsecret variable `L`. Laminar raises an exception because the security method does not have the right to declassify `o.H`. The value of `L` does not change. The `catch` block gives the programmer a chance to restore program invariants before exiting the security method.

If a thread tries to enter a security method for which it lacks appropriate capabilities, or if the thread passes illegal inputs to the security method, the VM raises an exception and transfers control to the security method's terminating `catch` block. Essentially, entering a security method with invalid credentials will effectively skip execution of the security method without revealing any information to the calling thread.

If a system call is attempted that would generate an illegal information flow, the OS returns a unique error code to the VM. The VM treats this error as a security exception; that is, the same way as an illegal flow through application-level variables.

An attempt to access labeled data outside of a security method will terminate the application. To prevent covert channels by testing whether an object is labeled at all, assignments to references must be treated as explicit information flows, described in the next subsection.

6.3. Information Flow through Object References

When a labeled object is created in a security method, Laminar restricts how the object stores references in order to prevent information leaks. One option is that an object reference can be written to a `static` variable, which must have empty secrecy and integrity labels. Therefore, only a method with the capability to drop its labels (i.e., a declassifier) can store a labeled object reference in a static. Similarly, security methods with an empty secrecy label can return an object reference to the caller.

A security method may store a reference to one object inside of another. For instance, suppose a security method writes a reference to newly created object `x` into object `o`'s field `o.p`. This assignment is an explicit flow from the security method into object `o`, and the VM-inserted barriers check the information flow. If `o`'s labels are $\langle S(o), I()\rangle$ and the security method's labels are $\langle S(o, x), I()\rangle$, this assignment is an illegal flow that would violate the secrecy rule, and it triggers a security exception.

Programmers may find it helpful to pass an object reference as input to multiple security methods. This convention does not leak data because object references are passed by value in the Java calling convention. As discussed earlier, returning an initial reference to an object or storing the reference in a `static` requires the capability to declassify the secret. Subsequent reads of the reference will not leak secret data. A subsequent security method cannot update a `static` reference unless it can declassify all of its secret data. Similarly, overwriting an input parameter in a security method does not propagate information to the caller because object references are passed by value in Java.

This pattern for passing secret data among security methods can be generalized by creating *security container* objects—an object whose reference is public which stores a set of secret data or object references. Security methods with the same secrecy label as the security container may conveniently write to the object and accept its reference as input. This convention does not leak any information because the public reference is never changed, and the contents of the container are protected by VM barriers.

To facilitate this pattern, we permit `new` to operate as a security method that can return a newly constructed object. Because the object is actually allocated from the heap and labeled before the constructor is called, a labeled object can always be returned without leaking secret information. If the constructor fails or throws an exception, the exception is masked, just as with any other security method, and a partially initialized, but labeled, object is returned. Objects with integrity tags must be allocated inside of

```
        public static void main (..) {
           MyObj m, k;
           // Label S = S(a), I_b = I(b), I_Empty = I();
           // CapSet C = C(b+)

           // Create an object with labels ⟨S(a), I(b)⟩
[L1]       m = new MyObj(S, I_b) ();
[L2-4]     manipulateObj(S, I_b, C) (m);
[L5]       updateSecret(S, I_empty) (m);

[L6]       k = m; /* Legal copy of a local object reference */
[L7]       k.val = 0; /* Runtime error, since k points to a labeled object */

        }

          secure (Label S = S(a), Label I = I(b), CapSet C = C(b+))
          void manipulateObj(m) {
             // Endorse reference m
[L2]         manipulateObjInternal(S, I) (m);
          }

          // Use m in a security method
          secure (Label S = S(a), Label I = I(b))
          void manipulateObjInternal(m) {
[L3]         MyObj n = new MyObj();
[L4]         m.val = n.val + 5;
          }

          secure (Label S = S(a), Label I = I())
          void updateSecret(m) {
[L5]          m.secret.x = computeNewSecret(); // Internal function, not shown
          }
```

Fig. 5. Allocating and passing objects among security methods using local references. Runtime values of labels and capability sets are inlined for clarity.

an endorser security method. Nested security methods can allow an endorser with the capability to add a secrecy tag to create an object with both secrecy and integrity tags in its label. This approach makes it easier for the programmer to create a security container and pass it among security methods, without creating data leaks.

Local reference example. Figure 5 shows an example in which a local object reference `m` is passed among security methods. The constructor for the new `MyObj` creates a labeled object at line `L1`. This object is assigned to local reference `m` and passed to the security method `manipulateObj`, where it is modified (`L4`). Outside of the security method, the reference `m` may be safely copied to another reference `k`. An attempt to dereference either reference outside of a security method will result in a runtime exception, since both point to a labeled object.

Integrity example. Figure 5 also illustrates how Laminar guarantees integrity. In line `L1` we create an object and label it with integrity label `b`. This object is returned to the calling thread and assigned to `m`. This reference `m` is passed to a security method (`manipulateObj`) but because the local reference itself is untrusted, the reference must be endorsed (`L2`) and then passed to the nested, high-integrity security method. The reference `k` is also assigned outside the security method to a high-integrity object at line `L6`. Since Laminar does not track labels of references, such an assignment outside the security method is valid. However, Laminar would prevent low-integrity code from modifying the high-integrity object. For example, the VM will raise an exception at line

```
// Object o has labels ⟨S(h), I()⟩
//   and members
// L has labels ⟨S(), I()⟩
static boolean L = false;
secure  (Label S = S(h), Label I = I())
void implicit (Object o) {
  if (o.H) L = true;
  ...
} catch (...) {
  ...
}
```

Fig. 6. An example implicit flow. Label and capability values are inlined for clarity.

`L7` when the value of the object pointed to by `k` is dereferenced outside of a security method.

Secrecy example. Figure 5 illustrates how an object can also be used as a security container. As discussed earlier, reference `m` points to a secret object, which cannot be dereferenced outside of a security method. This object may store other secrets, such as object reference `x`, which can be read and modified inside high-secrecy security methods, illustrated in Line `L5`.

6.4. Implicit Information Flows

Security methods limit implicit flows by hiding the control flow within the security method and preventing exceptional control flow from leaving the method. An implicit information flow leaks secret data through control flow decisions [Denning and Denning 1977]. To deal with implicit flows due to exceptional control flow, the VM requires every security method to have a `catch` block, as shown in Figure 4. The `catch` block executes with the same labels and capabilities as the security method. A security method may explicitly catch specific exception types (e.g., an arithmetic exception caused by a potential divide by zero in Figure 4) and use the ellipsis syntax to catch all other exceptions (equivalent to a `catch` block that catches any `Throwable`). The VM suppresses other types of exceptions inside a security method that are not explicitly caught inside the security method, including exceptions within the catch block. Thus, exceptions cannot escape a security method. The VM continues execution after the security method.

A major benefit of security methods is that they limit the amount of analysis necessary to restrict implicit information flows. Figure 6 includes an attempt to create an implicit flow. This security method code tries to leak the value of secret variable `o.H`, which it may not declassify, by deliberately creating an exception when it attempts an illegal explicit flow to the variable `L`. A thread might attempt to register an exception handler outside of the security method that would learn the value of `o.H` based on whether an exception occurred. This attack will not work because the VM will suppress any exceptions from leaving a security method.

To prevent information leaks, recall that security methods also cannot return a value unless they have an empty secrecy label (Section 5.2). Thus, security exceptions inside a secret security method cannot be reflected in the return value. A security method that has an empty secrecy label but a nonempty integrity label may return a value. If such a nonsecret security method incurs a security exception, the return value will either be set by the `catch` block or will be the default value for the return type.

Alternatively, a VM prototype could permit security methods to be simple blocks, as we proposed initially, called *security regions* [Roy et al. 2009]. Security regions must exit via fall-through control flow. Security regions cannot use `break`, `return`, or `continue` to

```
// Object o has labels ⟨S(h), I()⟩,
//   contains boolean H.
secure (Label S = S(h),
        Label I = I())
void termination (Object o) {
   if (o.H) while (true) { }
} catch (...) { }
```

Fig. 7. Leaking data via a termination channel. Runtime values of labels and capability sets are inlined for clarity.

```
// Object o has labels ⟨S(h), I()⟩,
//   contains boolean H.
secure (Label S = S(h),
        Label I = I())
void simpleTiming (Object o) {
   if (o.H) Thread.sleep (5000) { }
} catch (...) { }

void untrustedCode() {
   // Label S = S(h), I = I()
   long start = System.currentTimeMillis();
   simpleTiming (S, I) (o);
   long end = System.currentTimeMillis();

   if (end - start > 4000) {
      System.out.println(``o.H is true'');
   } else {
      System.out.println(``o.H is false'');
   }
}
```

Fig. 8. Leaking data via a single-threaded timing channel. Runtime values of labels and capability sets are inlined for clarity.

exit, except in the trivial case where the control flow will continue at the statement that immediately follows the security region.

Laminar thus eliminates implicit flows by hiding the control flow of a security method from code outside of the security method. In Figure 4, code outside the security method cannot distinguish an execution where `o.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 [Myers and Liskov 1997]. To prevent implicit flows, dynamic DIFC systems generally either restrict the programming model, which we have done, or adopt a hybrid of static and dynamic analysis [Chandra and Franz 2007; Nair et al. 2008; Venkatakrishnan et al. 2006].

6.5. Timing and Termination Channels

In addition to explicit and implicit flows, an adversary may try to leak information covertly through timing and termination channels [Lampson 1973]. A *timing channel* attempts to leak information based on how long a piece of code executes. A *termination channel* is a special timing channel that leaks information by executing in an infinite loop depending on a secret value. We do not eliminate all timing and termination channels for multithreaded programs, but we discuss potential solutions that minimize their bandwidth.

Termination Channels. Figure 7 shows an example of a termination channel that attempts to leak secret information based on whether the application terminates. If control returns from this security method, then unprivileged code can learn that `o.H` is false. Similarly, a colluding application might learn that `o.H` is true if the application appears to hang.

No general-purpose DIFC system can ensure termination of a program (or, in Laminar's case, a security method). The primary goal in dealing with termination channels is preventing a deterministic or high-bandwidth channel. OS-based systems can suppress termination notification [Efstathopoulos 2008; Krohn et al. 2007; Zeldovich et al. 2006] and thereby eliminate termination channels. Even this approach arguably

```
// Object o has labels ⟨S(h), I()⟩
//   and contains boolean H
// L has labels ⟨S(), I()⟩
   static boolean L = true;
```

```
/* Thread 1 */

secure (Label S = S(h),
        Label I = I())
void timing1 (o) {
   if (o.H==true)
     Thread.sleep(5000);
} catch (...) { }
L=false;
```

```
/* Thread 2 */

secure (Label S = S(h),
        Label I = I())
void timing2 () {
   Thread.sleep(1000);
} catch (...) { }
System.out.println(L);
```

Fig. 9. A timing channel attack that with high probability prints L with the same value as the secret H. Runtime values of labels and capability sets are inlined for clarity.

creates some disruption in the CPU scheduling that might create a channel that is noisy and thus difficult to exploit.

In a language-level system like Laminar, untrusted code placed after a security method can detect whether a security method has terminated. A number of solutions have been explored in the literature, surveyed by Kashyap et al. [2011]. One option is to use static analysis to identify the labels of all variables used to make control flow decisions and only permit the code to execute if it can declassify these labels [Chandra and Franz 2007; Liu et al. 2009] or to restrict the programming model to forbid using a secret value as a conditional variable [Volpano and Smith 1999]. Another option is to partition and schedule the code based on labels [Kashyap et al. 2011]. A final option is to bound the maximum execution time of sensitive code [Askarov et al. 2010; Tiwari et al. 2009a] and return control to the untrusted code even if the sensitive code has not completed.

For Laminar, the most attractive approach to termination channels is simply bounding the execution time of a security method. If the maximum execution time (perhaps specified by the programmer) is exceeded, a security exception would be generated. Control would be transferred to the `catch` block, permitting the secure code to clean up (again for a bounded period) and then return to the unlabeled thread. This approach would prevent security methods from leaking data based on termination by artificially forcing all security methods to terminate.

Timing Channels. Similarly, a timing channel can leak information based on the execution time of a security method (or other privileged code in a different DIFC system). Figure 8 shows a timing channel that artificially delays execution based on the value of secret variable `o.H`. This sort of channel can be created even with a single thread by recording the time before and after execution.

In practice, these timing and termination channels have been low bandwidth and are difficult to exploit—especially in single-threaded applications. However, multithreaded applications are more vulnerable to these exploits because more threads can synchronize the order in which they execute a security method, which is then visible outside the security methods. Figure 9 illustrates a timing channel where threads artificially delay the length of security method execution based on the value of secret variable `o.H`. Even though neither security method explicitly leaks anything or fails to terminate, the execution time orders the execution of updates to the static variable `L`, thus leaking the value of `H` with high probability, but somewhat slowly.

Figure 10 shows a more subtle attack that efficiently and deterministically leaks one bit of information per security method execution. In this example, `signal` is a variable

```
// Object o has labels ⟨S(h), I()⟩
//   and has boolean field H
// signal has labels ⟨S(), I()⟩
static int signal = -1;
```

/*Thread 1*/

```
secure (Label S = S(h),
        Label I = I())
void timing3 (o) {
   while (o.H==false && signal!=0);
} catch (...) { }
if(signal==-1) signal=1;
System.out.println(signal);
```

/*Thread 2*/

```
secure (Label S = S(h),
        Label I = I())
void timing4 (o) {
   while (o.H==true && signal!=1);
} catch (...) { }
if(signal==-1) signal=0;
System.out.println(signal);
```

Fig. 10. Leaking information via synchronization and timing. Runtime values of labels and capability sets are inlined for clarity.

with empty labels. If the secret, `H`, is 1 then Thread 2 gets into a `while` loop until Thread 1 exits its security method and sets the value of `signal` to 1. Thus, at the end of the security method, the value of `signal` is the same as that of secret H. A variant of this attack is also possible with only one thread in a security method and a second thread sleeping and then writing to `signal`.

We observe that the attacker in Figure 10 increases the bandwidth of timing channels by leveraging a data race on a signal variable. It is likely that the bandwidth of some timing attacks in a dynamic DIFC system like Laminar could be reduced if the program were known to be Data Race Free (DRF). Relying on programmers to write DRF programs is straightforward but will fail to prevent attacks if programmers make mistakes. Guaranteeing DRF through language design and type checking would prohibit data races but requires programmer effort [Boyapati et al. 2002]. Alternatively, the memory model could be strengthened so that synchronization-free regions appear to execute atomically [Ouyang et al. 2013]. We note that even DRF programs can still include timing channels, such as the one in Figure 8. Previous work has demonstrated how a language-based DIFC system can reduce or eliminate timing channels in multi-threaded programs by requiring data race freedom and that all traces of accesses to public or low-secrecy variables are not influenced by secret inputs [Huisman et al. 2006; Zdancewic and Myers 2003]. In general, locks for variables that are accessed across multiple labels must be acquired in code with either the lowest secrecy and highest integrity. Since correct lock acquisition complicates the programming model, we leave further investigation to future work.

Recent work [Askarov et al. 2010; Askarov and Myers 2012; Zhang et al. 2011] provides an alternative promising approach to mitigating timing channels in language-based systems by (1) predicting the expected runtime of a security-sensitive method, (2) ensuring that every instance runs at least this long by delaying the return, and (3) if the prediction is exceeded, increasing the prediction for future instances. This predictive mitigation strategy substantially limits the ability of an attacker to create a timing-based implicit flow.

A variant of timing-based mitigation could be adopted by Laminar, in which programmers specify the execution time of a security method, plus some epsilon for imprecision in the runtime system. Fixing execution time would address both timing and termination channels, and we expect that this would be robust to synchronization-based timing attacks. We leave a formal treatment of this approach in the presence of concurrency to future work.

```
// Object o has secrecy label S(a)
//  and has a field boolean H
void thread() {
  String path;
  CapSet current = getCurrentCapabilities();
  Label L = createAndAddCapability();

  // Label S = S(a), I = I()

  leakH (S, I) (o);
  myMkdir (L, I) (path);

  File theDir = new File(path);
  if (theDir.exists())
        report("H was false");
  else
        report("H was true");
}

secure (Label S = S(a), Label I = I())
void leakH (Object o, Label L) {
  if (o.H) removeCapability(PLUS, L); // Runtime error in Laminar
}

secure (Label S = S(l), Label I = I())
void myMkdir (String name) {
  // thread capabilities would include l+ if o.H == false,
  //   affecting whether mkdir_labeled succeeds
  mkdir_labeled(name, S, I);
}
```

Fig. 11. An attempt to use thread capabilities as a storage channel. Runtime values of labels and capability sets are inlined for clarity. Based on the value of H, the thread tries to permanently drop a capability. Laminar prevents this leak by ensuring programs only make permanent capability changes outside of a security method.

6.6. Capability Management

Laminar adds a set of capabilities to each thread that persist across security methods. A critical concern is to ensure that the capability set not be used to create a storage channel to leak information. To avoid this, we treat a thread's capability set as a nonsecret, trusted variable, and any tag creation or deletion is an explicit, mediated information flow. Because we trust the JVM to manipulate the capability set correctly, the capability set's integrity label is treated as $\top$ inside of a security method, and we permit threads to read the capability set outside of a security method.

Figure 11 illustrates how such an attack might be attempted otherwise. The attacker thread initially creates a disposable capability for tag `l`, in `Label L`. Inside one security method, the thread drops `l` based on the value of secret `o.H` and later tries to use the capability (implicitly) in another security method to create a labeled directory. The thread does possess the $C(l^-)$ to declassify any data protected by `l`, which should create a public output if it is successful. In this attack, the value of `o.H` determines whether the thread drops the $C(l^-)$ capability, which determines whether the thread can create a directory with an irrelevant tag in its label. The untrusted code can see whether the directory exists and learn the value of `o.H`. In this example, the thread is essentially using the thread's capability set as a storage channel to leak a secret value.

```
// The thread takes in the reference to these caps as inputCap;
// 0 is alice, 1 is bob.
server_thread(Capability inputCap[2])

  // The thread has capabilities C(a^{+,-}, b^{+,-})
  AliceCap = inputCap[0];
  BobCap = inputCap[0]; // Bug: Really AliceCap

  /* Schedule the Meeting Mtg, with labels Label S = S(A,B), Label I = I() */

  BobDeclassify( S, I, BobCap)(Mtg);
  AliceDeclassify( S, I, AliceCap)(Mtg);


secure (Label S = S(A,B), Label I = I(), Capability C = BobCap)
final void BobDeclassify() {
   // Bobcap is really alice cap.
   // Copy Alice's calendar to Bob with labels ⟨S(B), I()⟩
}
```

Fig. 12. A potential "confused deputy" when managing capabilities in an untrusted thread. Runtime values of labels and capability sets are inlined for clarity.

To prevent such a leak in Laminar, threads may only drop a capability either (1) outside of a security method or (2) in a security method with an empty secrecy label. Essentially, dropping a capability is a write to the thread's capability set, which has an empty secrecy label. Thus, this operation must be treated as an explicit write and mediated appropriately.

Capabilities and Confused Deputies. One problem with threads that dynamically assign capabilities to security methods is that a bug in the untrusted thread code can inadvertently give a security method an inappropriate capability. Figure 12 shows a "confused deputy" [Hardy 1988] variant of the calendar example. The server thread accidentally gives Bob's declassifier Alice's declassification capability. Perhaps realizing the mistake, Bob copies Alice's entire calendar into his calendar—a legal information flow.

Dynamic capability management was a design decision made in the interest of programmability. Unfortunately, as it stands, this choice increases the auditing burden on the security method developer. Not only must Alice audit her own security methods, she must audit the capability management code of threads that hold her declassification capability. Capabilities are Alice's primary credentials in Laminar, so it is not surprising that capability management code requires a security audit. In some cases, it might be possible to trade auditing capability management code for auditing all security methods that a thread may call. However, that set might be difficult to determine statically, and it might include dynamically loaded methods and methods written by other users.

To mitigate some of the risks of accidentally passing capabilities to the wrong security method, especially in the presence of inheritance of standard methods, capabilities must be explicitly passed to endorsing and declassifying security methods. Moreover, security methods receiving capabilities must be marked as `final`, thus disabling inheritance. For security methods that manipulate labeled data without label changes, no capabilities need be passed to the method. When a security method is not explicitly passed capabilities, the thread's capability set will be temporarily assigned to the empty set for the duration of the security method.

When a security method calls a function, its capabilities are not passed to this function unless the function is a nested security method that is explicitly passed capabilities as arguments. This restriction reduces the risk of unexpected information flows through a third-party library.

An alternative design could allow Alice to map her capability to a hash of specific security methods, either in addition to or instead of thread capabilities. Such a mapping of capabilities to a security method alleviates the need to audit any code outside of the security method. We leave development of such a mechanism for future work.

7. LIMITATIONS

Although Laminar regulates explicit information flows and hides control flow within a security method to prevent implicit flows, it is prone to attacks that exploit covert channels. For example, in the case of dynamic class loading, a user can query the VM to determine if a class has been loaded and use this additional information to leak sensitive data. In multithreaded programs, attackers may collude and use timing channels to leak information. We propose to mitigate these timing channels by fixing the execution time of a security method (Section 6.5). Such channels could also be mitigated by restricting the behavior of the scheduler [Sabelfeld and Myers 2003]. Laminar assumes that code blocks enclosed inside security methods always terminate. Otherwise, as explained in Section 6.4, information can leak through termination channels.

The Laminar prototype trusts Java Native Interface (JNI) code that is included as part of the JVM. It does not track information flow through JNI code and does not allow third-party JNI modules. A production JVM could track the information flow through correct JNI code because the JNI specification requires C and Java to use separate heaps. The required copying of input and output data could serve as a natural point at which to check labels. We note that, as an optimization, many JVM implementations do give the C code pointers into the Java heap. This optimization must be disabled. A deeper concern is protecting against untrusted, user-provided JNI code. Because C is not memory safe, a malicious JNI module could guess or otherwise discover the location of JVM-internal bookkeeping. Protecting the JVM from untrusted JNI code would require a sandboxing technique, such as running the JNI in a separate address space, and is beyond the scope of our work.

In general, the implementation could handle Java reflection calls by intercepting them and handling them like normal calls for the purposes of Laminar's security checks. The implementation could similarly handle calls to `sun.misc.Unsafe` methods, which perform raw memory accesses, by instrumenting the methods to perform Laminar's checks. However, the prototype currently ignores these cases.

There is, however, a specific concern with combining reflection, multithreading, and file descriptors to create a covert channel. For instance, one could conditionally create a secret file inside of a security method, which influences the assigned file descriptor to file or socket creation outside of a security method, leading to a covert channel. This risk is only introduced when threads with different labels share a file descriptor table. The current Laminar prototype blocks this attack by relying on the fact that file descriptor values are hidden from the application in the `FileInputStream` and `FileOutputStream` classes without reflection or `sun.misc.Unsafe`. Thus, care would need to be taken in allowing an application to directly interact with the file descriptor table.

The current implementation of Laminar does not allow application developers to read object labels, which may be useful for debugging. It is possible that some degree of visibility into object labels could be given to developers without creating new covert channels, but we leave this issue to future work.

The current implementation of Laminar treats static variables as unlabeled instead of associating labels with them (Section 5.3). Since most programs use statics infrequently, an improved implementation could track their labels without affecting the performance results.

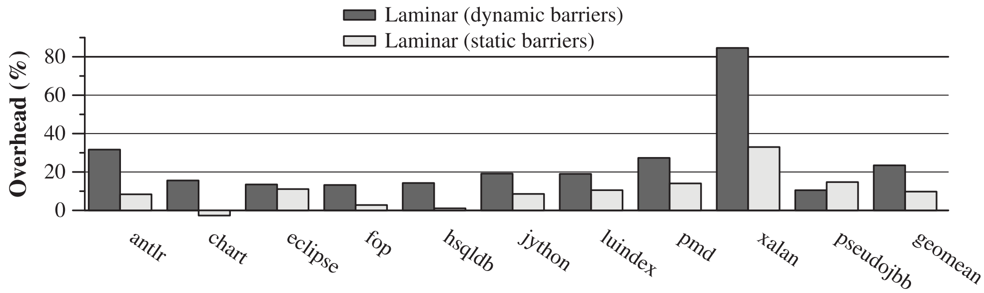

Fig. 13. Laminar VM overhead on programs without security methods.

As described in Section 3.8, the Laminar design does not trust `javac` to implement any DIFC enforcement but does trust `javac` to correctly compile the Java source to bytecode. Our prototype does not include a bytecode verifier, which could detect and reject invalid bytecode. Thus, the prototype Laminar VM trusts that bytecodes conform to the specification. A production Laminar VM implementation would include a bytecode validator.

The Laminar design requires security code to be written as methods, in order to simplify the enforcement of information flow rules on local variables (Section 5.2). With additional static analysis, it is possible that security code could be arbitrary, lexically scoped regions, as originally proposed [Roy et al. 2009]. After experimenting with a number of variations on the design, our experience is that restricting security code to methods strikes the best balance among programmability, security, and efficiency.

Finally, several restrictions on the programming model are not currently checked in the Laminar VM runtime system but would be implemented in a production system. Rather, we require programmers to adhere to these restrictions and manually enforce them in our application studies. Specifically, the Laminar JVM prototype does not currently enforce input and output restrictions to security methods, enforce restrictions on labeled allocation, or restrict that security methods that accept capabilities are declared `final`. The VM could easily enforce all of these rules at runtime, and the JIT compiler could use static analysis to enforce some of them as well.

8. LAMINAR PERFORMANCE

This section reports the performance overheads incurred by adding Laminar to Jikes RVM and Linux. We conducted these experiments on a quad-core Intel Xeon 2.83GHz processor with 4GB of RAM. We configure Jikes RVM to run on four cores. The VM's heap is configured with a maximum size of 1,024MB. All results are normalized to values obtained on unmodified Linux 2.6.22.6 and Jikes RVM 3.0.0. We measured Laminar's overhead on standard Java benchmarks without security methods to be less than 10% using static barriers specific to code outside security methods. We measured Laminar OS overhead on `lmbench`, a standard OS benchmark, to be less than 8% on average.

8.1. JVM Overhead

Figure 13 shows the overhead of Jikes RVM with the Laminar enhancements on the DaCapo Java benchmarks [Blackburn et al. 2006], version 2006-10-MR2, and a fixed-workload version of SPECjbb2000 called `pseudojbb` [Standard Performance Evaluation Corporation 2001]. Because compilation decisions are nondeterministic, running times vary, so we execute 25 trials of each experiment and take the mean.

Table IV. Execution Time in Microseconds of Several lmbench OS Microbenchmarks on Linux with Laminar

Benchmark	Linux	Linux w/ 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

These bars represent two sets of runs, one with dynamic barriers and one with only static barriers. The darker bar shows the overhead of dynamic barriers, which check dynamically if they are in a security method as well as performing the secrecy and integrity checks as appropriate. Dynamic barriers add 23% overhead on average. The lighter bar is the overhead of using static barriers, which only do the appropriate per-object DIFC checks. This overhead is 9.7% on average. As discussed in Section 5.1, a mature implementation of Laminar would use method cloning to eliminate dynamic barriers. Because method cloning has comparable overheads to static barriers, code outside of a security method is expected to have an average overhead of 9.7%. This result is consistent with Blackburn and Hosking's measurements of barriers [Blackburn and Hosking 2004].

8.2. OS Overhead

We use the standard `lmbench` [McVoy and Staelin 1996] system call microbenchmark suite to measure the overheads imposed on unlabeled applications when running on Laminar OS. A selection of the results is presented in Table IV.

In general, the overhead of the Laminar OS modifications are less than 8%, which is similar to previously reported overheads for Linux security modules [Wright et al. 2002]. The only performance outlier is the "null I/O" benchmark, which has an overhead of 31%. This benchmark represents the worst case for Laminar because the system call does very little work to amortize the cost of the label check. As a comparison, Flume adds a factor of 4-35× to the latency of system calls relative to unmodified Linux [Krohn et al. 2007].

9. APPLICATION CASE STUDIES

This section describes four case studies (GradeSheet, Battleship, Calendar, and FreeCS) and how we retrofitted these applications with DIFC security policies. GradeSheet implements a database with security policies for entering and reading grades by professors, TAs, and students. Battleship is a two-player game that keeps secrets about ship locations. Calendar manages multiple users calendars and arranges meeting times, similar to our running example. FreeCS is a chat server that implements security policies on group memberships and invitations. For each benchmark, we describe in more detail its functionality, modifying and retrofitting its security policies, and its performance.

Table V summarizes application details. All of the retrofitted applications implement more powerful security policies than their unmodified counterparts, yet all modifications add at most 10% to the source code. We list the lines of code statically within a security method, which is under 7% of the total lines of code. This count does not include code in methods called by a security method (e.g., library methods) that do not implement security policies.

Table V. Application Characteristics

Application	LOC	Protected Data	Added LOC (%)		SM LOC (%)		% time in SMs
GradeSheet	900	Student grades	92	(10%)	62	(6.9%)	<1%
Battleship	1,700	Ship locations	95	(6%)	57	(3.4%)	18%
Calendar	6,200	Schedules	290	(5%)	189	(3.0%)	1%
FreeCS	22,000	Membership properties	1,200	(6%)	80	(0.4%)	<1%

Lines of code (LOC), security sensitive data, Laminar specific LOC we added, LOC inside a Security Method (SM) statically (excluding code called by an SM), and percent time in security methods.

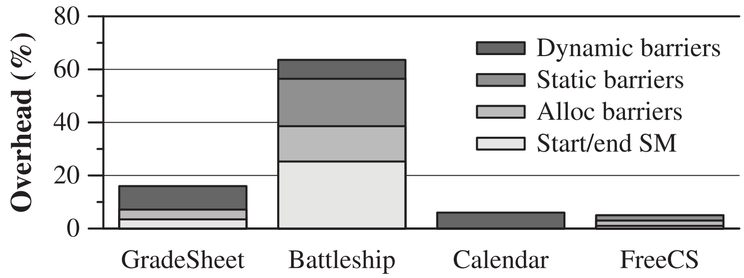

Fig. 14. Overhead of executing applications retrofitted with Laminar.

Figure 14 breaks down into four parts the overheads added by securing them using Laminar. *Start/end SM* is the overhead of application modifications to support DIFC, including starting and ending security methods and security operations, such as `copyAndLabel`. The *Alloc barriers* configuration denotes the time overhead for allocating labeled objects and assigning their label sets. The *Static barriers* configuration is the overhead from read and write barriers when the security context is known at compile time. Finally, the *Dynamic barriers* configuration is the overhead from barriers that check context at runtime. 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 methods. As discussed in Section 5.1, method cloning would obviate the need for dynamic barriers, and we thus expect that in practice overhead will match the overhead of *Static barriers*.

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. In particular, when we wait for the Battleship game to draw the GUI between scripted moves in the test cases, the measured Laminar overhead drops to 1%. For comparison, Flume [Krohn et al. 2007] 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.

9.1. GradeSheet

GradeSheet is a small program that manages the grades of students [Birgisson et al. 2008]. GradeSheet has three types of end users: 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 grades on project j. A sample policy states that (1) the professor can read/write any cell, (2) the TA can read the grades of all students but only modify the ones related to the project that she graded, and (3) students can only view their own grades for all projects.

Table VI shows how to express this policy by assigning labels and capabilities to the data and the threads working on behalf of each type of user, respectively. Specifically,

Table VI. GradeSheet Security Sets for Objects and Threads Serving End Users, where *S* Is Secrecy, *I* Is Integrity, and *C* Is Capability

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

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 capabilities to add or remove s_i, so students can read their own grades 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 grades 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 grades in a project, which leaks information about the grades of other students. Using Laminar, we specified that only the professor is allowed to calculate the average and declassify it.

Our experiments measure the time taken by the server to process a mix of queries by the TA. Overall, the queries are 72% writes and 28% reads, including reads of student ID and average grade, and reads and writes of student grades. The Laminar-enabled version has a 7% slowdown compared to the unmodified version.

9.2. 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 modified `JavaBattle`,[5] 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 method. The opponent then declassifies whether the guess was a hit or a miss and sends that information back to the first player. We added fewer 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 14 shows that the secured version adds 56% overhead with static barriers. The overhead is high because the benchmark spends a substantial portion of its time of its time (18%) inside security methods. In a deployed Battleship, which would display the intermediate state of the board to the players, the overhead is significantly less. In an experiment where we displayed the shot location after each move, the runtime of the application increases significantly, and Laminar overhead drops to 1%.

9.3. Calendar

We modified `k5nCal`,[6] a multithreaded desktop calendar that provides a graphical interface and allows users to subscribe to multiple external iCalendar-based calendars. It

[5]http://sourceforge.net/projects/javabattle/.
[6]http://k5ndesktopcal.sourceforge.net.

has different threads for rendering the GUI, importing calendar files, and periodically fetching updates from remote calendars. Our modifications provide similar functionality as in the examples from earlier in this article. We label all data structures and `.ics` files that store a user's calendar information with the user's secrecy tag. We wrap all functions that access private calendar data inside security methods, 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 executes in a thread that has the capabilities to read data for both Alice and Bob, but can only declassify Bob's data. The output file is protected by Alice's secrecy tag. Our experiment schedules 1,000 meetings. Figure 14 shows that the secured version of Calendar runs 6% slower than unmodified Calendar.

We note that a substantial portion of the time in the calendar application is spent on internal thread creation and management, and even more time would be spent rendering a GUI if we had not disabled this feature. For comparison, we lifted the scheduling code out of the rest of the application and wrote a microbenchmark that scheduled appointments in a tight loop on a single thread. In this case, the percentage of time in security methods increased to 71%, and the total overhead was 77%. In practice, we expect things like user interaction and thread management to dominate execution time, thus minimizing the impact of security methods.

9.4. FreeCS Chat Server

`FreeCS`[7] is an open-source chat server written in Java. 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 methods. We made most of our modifications in two classes—`Group` and `User`. We localized all security checks by adding security methods to these classes. The abstraction of a role maps naturally onto integrity labels. For example, we protected 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 modified the authentication module to assign each user either the VIP capability, superuser capability, or no enhanced capability when she logs in. The authentication module is trusted to manage the VIP and superuser capabilities. Our experiments measure the time to process requests from 4,000 users, each invoking three different commands. Laminar's overhead is 5% (Figure 14).

9.5. 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`

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

in GradeSheet and `DataFile` in Calendar contain the functions used to access the important data. These observations support our hypothesis that Laminar requires only localized and modest changes to add DIFC security to many types of applications. Third, most of the data structures require heterogeneous labeling—the single array data structure `GradeCell` has different labels corresponding to different students. Heterogeneous labeling is impractical in OS-based systems [Krohn et al. 2007; Vandebogart et al. 2007; Zeldovich et al. 2006] because 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-grained tracking of labels on the data structure, for example, individual array elements and objects in GradeSheet.

An open question is the how this approach will scale to larger applications. Our experience with Laminar is that the performance overheads are primarily determined by the amount of code that executes inside a security method and that developer effort is a function of how many declassification or endorsement points the code requires, rather than the amount of data the program secures. The case studies presented here had natural and simple endorsement and declassification points, which were close to the actual uses of labeled data—minimizing overheads and effort. For larger applications, this trend may continue. However, it is possible that larger applications may instead require a larger portion of code in security methods to manipulate labeled data or that more substantial refactoring may be required to minimize the code that must execute in a security method. We leave larger application case studies for future work.

10. 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 VM 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. Table VII summarizes the taxonomy of design issues common to DIFC systems, ranging from the trusted code base, security guarantees, resource granularity, to threats, all addressed in more detail here.

From IFC to DIFC. Information Flow Control (IFC) stemmed from research in multilevel security for defense projects [Department of Defense 1985]. In the original military IFC systems, an administrator must allocate all labels and approve all declassification requests [Karger et al. 1991]. Modern Mandatory Access Control (MAC) systems, like security-enhanced Linux (SELinux), also limit declassification and require a static collection of labels and principals. DIFC systems provide a richer model for implementing security policies in which applications allocate labels and assign them to data and declassify [Myers and Liskov 1997].

Static DIFC analysis. Many language-based DIFC systems augment the type system to include secrecy and integrity constraints enforced by the bytecode generator [Myers 1999; Myers et al. 2001; Simonet and Rocquencourt 2003]. These systems label program data structures and objects at a fine granularity but require programming an intrusive type system or in an entirely new language. These language-based systems trust the whole OS and provide no guarantees against security violations on system resources, such as files and sockets.

Hybrid DIFC enforcement. A key strength of static analysis is that static analysis tends to be the most robust language-level defense against implicit channels (Section 6.4). Purely dynamic systems generally cannot effectively regulate implicit flows. As a result, a number of primarily dynamic JVM systems have augmented

Table VII. High-level Approaches to DIFC Implementation

Issue	PL solutions	OS solutions	PL & OS solution
	[Arden et al. 2012; Chandra and Franz 2007; Liu et al. 2009; Myers et al. 2001; Simonet and Rocquencourt 2003]	Asbestos [Efstathopoulos 2008; Vandebogart et al. 2007], HiStar [Zeldovich et al. 2006], Flume [Krohn et al. 2007]	**Laminar**
Modified	Compiler & type system ([Arden et al. 2012; Liu et al. 2009; Myers et al. 2001; Simonet and Rocquencourt 2003]) or JVM and bytecode compiler ([Chandra and Franz 2007])	(1) Complete OS or (2) User-level reference monitor & kernel module	VM and kernel module
Trusted	Compiler, VM, & OS	OS	VM & OS
Fine-grained information flow tracking?			
	Interprocedural static analysis or JVM instrumentation	Either not supported or inefficient because of page table mechanisms	Dynamic VM enforcement via inserted read/write barriers
Secure files & OS resources?			
	Can label file handles in the application and add dynamic checks to system calls, but limited visibility into OS to validate these assumptions.	(1) Modify entire OS or (2) User-level reference monitor & kernel	Kernel
Implicit flows?	Static analysis, combined with dynamic checks in some cases [Arden et al. 2012; Chandra and Franz 2007; Liu et al. 2009].	Not applicable—tracks information flow at thread or address space granularity	*Security method* design restricts visibility into control flow from outside the security method.
Termination, probabilistic, and timing channels?			
	Predictive Mitigation [Askarov et al. 2010; Askarov and Myers 2012; Zhang et al. 2011]	HiStar, Flume, & Asbestos suppress termination notification	Not handled

Laminar combines aspects of PL and OS solutions, and innovates in dynamic flow tracking.

dynamic enforcement of explicit flows with static analysis for implicit flows (thus called a Hybrid DIFC system).

Chandra and Franz develop a version of the JVM that enforces information flow control policies on unmodified Java programs [Chandra and Franz 2007]. Like Laminar, this JVM combines static analysis on Java bytecode with dynamic analysis. Security policies are expressed externally—such as restricting how sensitive data may exit the program. This system relies on whole-program, side-effect analysis to restrict implicit flows by labeling the program counter. Moreover, this system does not address threads and allows implicit flows through uncaught exceptions. Finally, the dynamic analysis in this system is relatively expensive, 23–159%, whereas Laminar's reported application overheads are 5–56%. Laminar's security methods instead strike a balance that minimizes programmer effort but substantially limits the scope and overhead of static and dynamic analysis.

Trishul adopts a similar design as Chandra and Franz, but better handles implicit flows through caught exceptions via static analysis [Nair et al. 2008]. Trishul does not handle uncaught runtime exceptions, such as divide by zero. Trishul relies on a conservative global program counter secrecy label when static analysis cannot prevent an implicit flow, such as when referencing certain object reference fields. This abstraction is prone to “label creep,” and programmers must manually remove labels according to application security policies. The performance overhead of Trishul varies and tends to be highest on object manipulation and lowest for system calls. For a prime number benchmark, the overhead is 167% [Nair 2009]. A key contribution of Laminar is a highly optimized JVM design, as well as a judicious and programmable abstraction selection that keeps overheads low.

The Laminar VM prevents implicit flows instead by restricting how control can return from a security method—a property that can be checked dynamically. Arguably, some restrictions could be relaxed with additional static analysis. Although Laminar does not rely on static analysis for safety, it does employ some analysis during JIT compilation to optimize security checks (Section 5.1) and could be considered a hybrid DIFC system.

OS IFC. Asbestos [Vandebogart et al. 2007] and HiStar [Zeldovich et al. 2006] are new OSs that provide DIFC properties. Flume [Krohn et al. 2007] is a user-level reference monitor that provides DIFC guarantees without making extensive changes to the underlying OS. These 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 enforces information flow at page granularity and supports a form of multithreading by forcing 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 methods 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 (LSMs) [Wright et al. 2002]). 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) because 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 distinct from *capability-based* operating systems, such as EROS [Shapiro et al. 1999]. These systems use pointers with access control information to combine system and language mechanisms for stronger security but use a centralized IFC model, rather than the richer DIFC model. Thus, capability systems cannot enforce DIFC rules, and programs must be completely rewritten to work with the capability programming model.

Integrating language and OS security. Hicks et al. observe that security-typed languages can ensure that OS security policies are not violated by trusted system applications, such as `logrotate` [Hicks et al. 2007]. Their framework, called SIESTA, extends Jif to enforce SELinux [Loscocco and Smalley 2001] 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.

Implicit information flows. Implicit information flows can leak secret data based on program control flow, as when a conditional statement is based on the value of a secret variable. DIFC systems based on static analysis can identify when labeled variables can influence control flow and use this information to label the program counter of the function [Chandra and Franz 2007; Liu et al. 2009]. In other words, a function with a secret program counter label may not be called by a function with an unlabeled program counter.

Venkatakrishnan et al. develop a framework that statically transforms program code in a simple procedural language into a form that can detect implicit flows at runtime [Venkatakrishnan et al. 2006]. Their model essentially adds explicit assignments to a program counter variable in the code at all conditional statements and procedure calls and catches illegal flows at runtime. This model is applied in the context of IFC and noninterference and has not been extended to DIFC or concurrency.

Le Guernic proposes an automaton-based information flow model and type system that uses static analysis to identify potential implicit flows [Guernic 2007]. Unlike other systems, this model also identifies synchronization events in threaded systems and imposes additional restrictions at runtime around conditional statements. These restrictions include requiring that locks be acquired before any conditional is evaluated based on a secret variable and executing statements within certain conditionals atomically. Unlike many IFC systems, this design avoids termination channels on failures by suppressing individual lines of code that might cause an implicit flow. Laminar adopts a similar approach to securing concurrency by limiting the possible interleavings of security methods.

Shroff proposes a dynamic monitor and type system that can prevent implicit flows either with the help of a static type analysis, which can be overly conservative in some cases, or by learning the implicit flows in repeated executions [Shroff et al. 2007]. In the dynamic-only mode, the system records the explicit flows within all taken branches. In subsequent executions, if a different branch is taken, the recorded flows of previous executions are used to identify potential implicit flows. In dynamic mode, this system can permit some number of leaks before converging on a tight approximation of secure rules.

Fabric and Mobile Fabric add additional checks, both static and dynamic, to prevent additional implicit flows in distributed and federated systems, respectively [Arden et al. 2012; Liu et al. 2009]. For example, loading a class from a remote server may indicate that a secret code took a certain execution path. The Fabric systems add additional labels and checks to prevent these flows.

Laminar restricts implicit information flows by restricting how exceptional control flow returns from a security method. OS DIFC systems generally do not need to address implicit flows because DIFC is enforced at process granularity, which hides control flow within the process by design.

Asymmetric behavior for secrecy and integrity. In general, DIFC systems treat secrecy and integrity as duals. As a result, the bottom of the label lattice, or least-restricted data, is public and trusted. In Laminar, most application code and data are untrusted and have an empty label. We believe this choice is appropriate for a threat model where an adversary may have contributed code to the application, and any given policy concern applies to a small subset of the code. As a result, however, the measures taken to ensure secrecy and integrity are different. For instance, removing capabilities and creating security container objects must execute outside of a security method to ensure that the operation is public and does not leak secret information. In contrast, security methods trusted with an integrity tag must generally sanitize public data and endorse this input. In the worst cases, malformed public data can make the system unavailable.

Termination, timing, and probabilistic channels. Implicit flows can be combined with termination, storage, and other features to create more powerful 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 [Vachharajani et al. 2004]. They prove that a correct dynamic DIFC system will overapproximate information flow, rejecting some programs that do not contain actual information flow violations. Russo and Sabelfeld similarly prove that a purely dynamic DIFC system will reject programs that a static analysis would not under a flow-sensitive analysis (i.e., when variables can change labels over the course of the computation) [Russo and Sabelfeld 2010]. Russo and Sabelfeld argue that these deficiencies can be recovered in a hybrid model, where some measure of static and dynamic analysis are combined. Laminar is a hybrid DIFC system but relies on dynamic checks and restricting the programming model to mitigate covert channels, and thus its security methods explicitly overapproximate information flow.

Recent work by Zhang, Askarov, and Myers developed a predictive mitigation strategy for timing channels [Askarov et al. 2010; Askarov and Myers 2012; Zhang et al. 2011]. Predictive mitigation essentially ensures that all instances of a sensitive method run for the same length of time. If a method runs longer than expected, all future instances run for the new maximum length. This strategy has been developed in static analysis systems but could be extended to dynamic DIFC systems such as Laminar.

In general, DIFC systems attempt to eliminate covert channels, which may be used to leak information, but do not eliminate timing channels [Lampson 1973] or probabilistic channels [Sabelfeld and Myers 2003]. DIFC systems can eliminate some implicit flows, as discussed in Section 6.4.

Formalizing information flow properties. Prior work has formally defined safety properties for information flow systems, primarily in the context of a type system. The most restrictive is *noninterference* [Goguen and Meseguer 1982], in which the output of a low-security computation cannot be influenced by the values of high-security computation. This definition precludes declassification and endorsement. In the case of our calendar example, a calendar application that enforced noninterference could not output a mutually agreeable meeting time. *Observational determinism* is a generalization of noninterference to concurrent programs [Huisman et al. 2006; Zdancewic and Myers 2003]. Observational determinism generally requires the program to be DRF, as well as requiring equivalent traces of possible accesses to nonsecret data.

An alternative safety condition is *robustness* [Chong and Myers 2005, 2006; Myers et al. 2004; Zdancewic and Myers 2001]. Within the lattice of labels in the decentralized label model, a robust system enforces boundaries on the ability of a principal to influence data or read data. In other words, a robust system would not allow a principal to expand its ability to read data based on the parts of the system it can influence. This model incorporates declassification, endorsement, multiple and mutually distrusting principals, and principals that can contribute and execute code.

Noninterference, observational determinism, and robustness have been applied primarily to DIFC-type systems. We leave adapting a property such as observational determinism to a dynamic DIFC system for future work.

In summary, Laminar combines the strengths of PL and OS DIFC systems. Laminar handles implicit flows and enforces the same fine-grained information flow control policies as performed by prior PL DIFC systems but without static analysis of all program components. Laminar enforces the same DIFC security policies on system resources as the OS DIFC systems enforce. Laminar, however, makes it easier to deploy and use information flow control systems by introducing security methods that encapsulate security code and are intuitive to program.

11. 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.

Although abstractions such as the security method minimize the refactoring burden on the programmer who wishes to adopt DIFC, the implementation mechanisms, such as dynamic policy enforcement and allowing a thread to execute methods with different labels, introduce additional opportunities for covert channels. To prevent some covert channels, the current Laminar implementation imposes a number of modest restrictions that we would like to relax in future work, such as limiting the use of static variables and forbidding file relabeling. This future work should be driven by a formal model of security methods that facilitates careful reasoning about security properties, especially about covert channels that arise due to concurrency.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, David Mazières, Eddie Kohler, Maxwell Krohn, and Nickolai Zeldovich for their feedback on earlier drafts of this document. Both David and the TOPLAS reviewers identified timing and termination attacks that led to improvements in the Laminar model.

REFERENCES

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, S. F. 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. 2000. The jalapeño virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238.

O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. 2012. Sharing mobile code securely with information flow control. In *Proceedings of the IEEE Symposium on Security and Privacy*. 191–205.

A. Askarov, D. Zhang, and A. C. Myers. 2010. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. 297–307.

D. E. Bell and L. J. LaPadula. 1973. *Secure Computer Systems: Mathematical Foundations*. Technical Report MTR-2547, Vol. 1. MITRE Corp., Bedford, MA.

K. J. Biba. 1977. *Integrity Considerations for Secure Computer Systems*. Technical Report ESD-TR-76-372. USAF Electronic Systems Division, Bedford, MA.

A. Birgisson, M. Dhawan, Úlfar Erlingsson, V. Ganapathy, and L. Iftode. 2008. Enforcing authorization policies using transactional memory introspection. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*.

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. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPLA'06)*. 169–190.

S. M. Blackburn and A. L. Hosking. 2004. Barriers: Friend or foe? In *Proceedings of the ACM International Symposium on Memory Management*. DOI:http://dx.doi.org/10.1145/1029873.1029891

C. Boyapati, R. Lee, and M. Rinard. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'02)*. 211–230. DOI:http://dx.doi.org/10.1145/582419.582440

D. Chandra and M. Franz. 2007. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*. 463–475.

S. Chong and A. C. Myers. 2005. Language-based information erasure. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'05)*. 241–254.

S. Chong and A. C. Myers. 2006. Decentralized robustness. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW'06)*. 242–256.

D. E. Denning. 1976. A lattice model of secure information flow. *Communications of the ACM* 19, 5 (May 1976), 236–243.

D. E. Denning and P. J. Denning. 1977. Certification of programs for secure information flow. *Communications of the ACM* 20, 7 (July 1977), 504–513.

Department of Defense. 1985. *Department of Defense Trusted Computer System Evaluation Criteria* (DOD 5200.28-STD (The Orange Book) ed.). Department of Defense.

P. Efstathopoulos. 2008. *Policy Management and Decentralized Debugging in the Asbestos Operating System*. Ph.D. Dissertation. University of California, Los Angeles.

J. A. Goguen and J. Meseguer. 1982. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy (SSP'82)*. 11–20.

G. Le Guernic. 2007. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the Computer Security Foundations Symposium (CSF'07)*. 218–232.

N. Hardy. 1988. The confused deputy: (Or why capabilities might have been invented). *SIGOPS Operation Systems Review* 22, 4 (Oct. 1988), 36–38.

B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. 2007. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX'07)*.

M. Huisman, P. Worah, and K. Sunesen. 2006. A temporal logic characterisation of observational determinism. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW'06)*.

P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. 1991. A retrospective on the VAX VMM security kernel. *IEEE Transactions in Software Engineering* 17, 11 (1991).

V. Kashyap, B. Wiedermann, and B. Hardekopf. 2011. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP'11)*. 413–428.

M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. 2007. Information flow control for standard OS abstractions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'07)*.

B. W. Lampson. 1973. A note on the confinement problem. *Communications of the ACM* 16, 10 (1973), 613–615.

H. M. Levy. 1984. *Capability-Based Computer Systems*. Digital Press, Bedford, MA.

J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. 2009. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'09)*. 321–334.

P. Loscocco and S. Smalley. 2001. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX'01)*.

J. Manson, W. Pugh, and S. V. Adve. 2005. The Java Memory Model. Retrieved from http://dl.dropbox.com/u/1011627/journal.pdf.

L. McVoy and C. Staelin. 1996. LMbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference (USENIX'96)*.

A. C. Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*. ACM, New York, NY, 228–241.

A. C. Myers and B. Liskov. 1997. A decentralized model for information flow control. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'97)*. 129–142.

A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. 2001. Jif: Java Information Flow. Retrieved from http://www.cs.cornell.edu/jif.

A. C. Myers, A. Sabelfeld, and S. Zdancewic. 2004. Enforcing robust declassification. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*. 172–186.

S. K. Nair. 2009. *Remote Policy Enforcement Using Java Virtual Machine*. Ph.D. Dissertation. Vrije University, Amsterdam.

S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. 2008. A virtual machine based information flow control system for policy enforcement. *Electronic Notes on Theoretical Computer Science* 197, 1 (Feb. 2008), 3–16.

Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. 2008. Design and implementation of transactional constructs for C/C++. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*. 195–212. DOI:http://dx.doi.org/10.1145/1449955.1449780

J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. 2013. ...and region serializability for all. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism (HotPar'13)*.

W. Pugh. 2005. May 12th Description of Final Fields. Retrieved from http://www.cs.umd.edu/~pugh/java/memoryModel/may-12.pdf.

I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. 2009. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*.

A. Russo and A. Sabelfeld. 2010. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF'10)*. 186–199.

A. Sabelfeld and A. C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21 (2003).

J. S. Shapiro, J. M. Smith, and D. J. Farber. 1999. EROS: A fast capability system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99)*.

P. Shroff, S. Smith, and M. Thober. 2007. Dynamic dependency monitoring to secure information flow. In *Proceedings of the Computer Security Foundations Symposium (CSF'07)*. 203–217.

V. Simonet and I. Rocquencourt. 2003. Flow Caml in a nutshell. In *Proceedings of the 1st APPSEM-II Workshop*. 152–165.

Standard Performance Evaluation Corporation. 2001. *SPECjbb2000 Documentation* (release 1.01 ed.). Standard Performance Evaluation Corporation.

M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood. 2009a. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 493–504.

M. Tiwari, H. M. G. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. 2009b. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. 109–120.

N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. 2004. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'04)*.

S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. 2007. Labels and event processes in the Asbestos operating system. *ACM Transactions on Computer Systems* 25, 4 (2007), 11.

V. N. Venkatakrishnan, W. Xu, D.l C. DuVarney, and R. Sekar. 2006. Provably correct runtime enforcement of non-interference properties. In *Proceedings of the 2006 International Conference on Information and Communications Security (ICICS'06)*. 332–351.

D. Volpano and G. Smith. 1999. Probabilistic noninterference in a concurrent language. *Journal of Computer Security* 7, 2–3 (Nov. 1999), 231–253.

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

S. Zdancewic and A. C. Myers. 2003. Observational determinism for concurrent program security. In *Proceedings of the IEEE Computer Security Foundations Workshop (CSFW'03)*. 29–43.

Steve Zdancewic and Andrew C. Myers. 2001. Robust declassification. In *CSFW*. IEEE Computer Society, Washington, DC, 15–23.

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

N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. 2008. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design & Implementation (NSDI'08)*.

N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. 2008. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2011. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'11)*. 563–574.

D. Zhang, A. Askarov, and A. C. Myers. 2012. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*.

Received March 2011; revised February 2014; accepted June 2014

QUERIES

Q1: AU: Please include full author addresses and emails.
Q2: AU: Is a term or word missing "we introduce security methods, an intuitive..."?
Q3: AU: Should this phrase be changed to "the lowest secrecy or highest integrity" as it follows "either"?
Q4: AU: For Boyapati et al. (and all refs) please supply the publisher and the page range.