Copyright

by

Indrajit Roy

2010
Protecting Sensitive Information from Untrusted Code

Committee:

Emmett Witchel, Supervisor

Michael D. Dahlin

David Mazières

Kathryn S. McKinley

Vitaly Shmatikov
Protecting Sensitive Information from Untrusted Code

by

Indrajit Roy, B.Tech.; M.S.C.S

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2010
To my family, whose hopes exceed these pages.
Acknowledgments

Both this thesis and my years as a graduate researcher have benefited exceedingly from the involvement of my adviser, Emmett Witchel. He is a stickler for details, always ready for discussions, and still responds to emails in the wee hours. During the period that I was adviser-less, and figuring out whether to work in computer science theory or systems, he generously sponsored my SPAA paper presentation. For all such things and more, that cannot be repaid, I remain humbly indebted.

Vitaly Shmatikov helped crystallize many ideas in this thesis. It has been a treat to watch him scratch out my paragraphs and substitute them with concise sentences. He is a skeptic whom I have had the hardest time to convince of my solutions. I thank Vitaly for the effort that he has invested and for his humorous matter-of-fact quips during the discussions.

Kathryn McKinley was always accessible for both technical and non-technical discussions. Her tips on searching for internships and, later, jobs have been quite effective. Thanks Kathryn for the advice and for sharing anecdotes about your graduate life.

Many thanks to Mike Dahlin and David Mazières for agreeing to be part of my thesis committee, and for providing helpful suggestions. David pointed out a synchronization based attack and a possible solution to mitigate the attack.

I was fortunate to have wonderful collaborators and co-authors like Don Porter, Mike Bond, Srinath Setty and Ann Kilzer. Don and Mike have enlightened
me on OS and JVM issues. Thanks guys, you have expedited my thesis.

Many thanks to the OSA group. The generally jolly, sometimes technical, lunch meetings satiated more than hunger. There were instances when I didn’t know what you guys were talking about. I hope someone did.

I thank all the members of the LASR group for listening to my talks and helping me improve them. In addition, Edmund Wong helped me absorb the way of the badminton racket, and I appreciate Eric Rozner’s comradeship during the Gradfest and the admission committee meetings. Prince Mahajan was subjected to re-reading drafts and re-listening my talks. I thank him for his patience, and the fun trips together.

Siddharth Chauhan, Prateek Jain, and Pushpraj Shukla are the reason that graduate school transitioned from a sedentary life to an enjoyable experience. Ramakrishna Reddy, even with no knowledge of my research, kept me motivated with his faith in me. I thank them all. Work was sometimes interspersed with online chats with some visible and other invisible (mode) friends. Thank you all. You have been the best distractions I could have hoped for.

I am thankful to Ms. N. Kapoor, who in high school, piqued my interest in computer science.

A final thanks to my parents, sister and grandmother—I hope you feel what I cannot express.

Indrajit Roy

The University of Texas at Austin

August 2010
Protecting Sensitive Information from Untrusted Code

Publication No. ____________________

Indrajit Roy, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Emmett Witchel

As computer systems support more aspects of modern life, from finance to health care, security is becoming increasingly important. However, building secure systems remains a challenge. Software continues to have security vulnerabilities due to reasons ranging from programmer errors to inadequate programming tools. Because of these vulnerabilities we need mechanisms that protect sensitive data even when the software is untrusted.

This dissertation shows that secure and practical frameworks can be built for protecting users’ data from untrusted applications in both desktop and cloud computing environment.

Laminar is a new framework that secures desktop applications by enforcing policies written as information flow rules. Information flow control, a form
of mandatory access control, enables programmers to write powerful, end-to-end security guarantees while reducing the amount of trusted code. Current programming abstractions and implementations of this model either compromise end-to-end security guarantees or require substantial modifications to applications, thus deterring adoption. Laminar addresses these shortcomings by exporting a single set of abstractions to control information flows through operating system resources and heap-allocated objects. Programmers express security policies by labeling data and represent access restrictions on code using a new abstraction called a security region. The Laminar programming model eases incremental deployment, limits dynamic security checks, and supports multithreaded programs that can access heterogeneously labeled data.

In large scale, distributed computations safeguarding information requires solutions beyond mandatory access control. An important challenge is to ensure that the computation, including its output, does not leak sensitive information about the inputs. For untrusted code, access control cannot guarantee that the output does not leak information. This dissertation proposes Airavat, a MapReduce-based system which augments mandatory access control with differential privacy to guarantee security and privacy for distributed computations. Data providers control the security policy for their sensitive data, including a mathematical bound on potential privacy violations. Users without security expertise can perform computations on the data; Airavat prevents information leakage beyond the data provider’s policy. Our prototype implementation of Airavat demonstrates that several data mining tasks can be performed in a privacy preserving fashion with modest performance overheads.
Contents

Acknowledgments v

Abstract vii

List of Tables xiii

List of Figures xv

Chapter 1 Introduction 1
  1.1 Constraining information flows ........................................... 2
  1.2 Regulating information content ........................................... 5
  1.3 Contributions ............................................................... 7
  1.4 Dissertation outline ....................................................... 7

Chapter 2 A Common Abstraction for Information Flow Control 9
  2.1 DIFC model ................................................................. 9
    2.1.1 DIFC abstractions .................................................. 10
    2.1.2 Information flow rules ............................................ 12
    2.1.3 Calendar example ................................................ 13
  2.2 Laminar design ............................................................ 15
    2.2.1 Enforcement mechanism .......................................... 16
    2.2.2 Programming model .............................................. 17
Chapter 3  Laminar Implementation and Evaluation  

3.1  JVM support  ......................................................... 31
3.2  Discussion  ............................................................ 36
  3.2.1  Objects  ............................................................ 36
  3.2.2  Locals  ............................................................ 37
  3.2.3  Statics  ............................................................ 42
  3.2.4  Labels  ............................................................. 42
3.3  OS support  ............................................................ 43
3.4  Limitations  .......................................................... 45
3.5  Laminar overhead  .................................................... 46
  3.5.1  JVM overhead  .................................................... 47
  3.5.2  OS overhead  ..................................................... 48
3.6  Application case studies  ............................................ 49
  3.6.1  GradeSheet  ....................................................... 50
  3.6.2  Battleship  ....................................................... 51
  3.6.3  Calendar  .......................................................... 52
  3.6.4  FreeCS chat server  .............................................. 53
  3.6.5  Handling covert channels  ..................................... 54
  3.6.6  Application insights  ............................................ 54
Chapter 4  Ensuring Privacy in Large Scale Computations  56

4.1  System overview ........................................  59
   4.1.1  Architecture of Airavat ............................  60
   4.1.2  Trusted computing base of Airavat ..................  62

4.2  Limitations of Airavat ..................................  62

4.3  MapReduce and MAC ....................................  63
   4.3.1  MapReduce ........................................  63
   4.3.2  SELinux .........................................  64

4.4  Differential privacy ...................................  65
   4.4.1  Function sensitivity ................................  67
   4.4.2  Privacy budget ...................................  68

Chapter 5  Airavat Enforcement Mechanisms  70

5.1  Enforcing differential privacy ..........................  70
   5.1.1  Range declarations and estimated sensitivity ......  72
   5.1.2  Range enforcement ..................................  73
   5.1.3  Mapper independence ................................  77
   5.1.4  Managing multiple outputs ..........................  77
   5.1.5  Trusted reducers and reducer composition ...........  79
   5.1.6  Enforcing $\delta$ ...................................  80
   5.1.7  Mapper composition ..................................  81
   5.1.8  Choosing privacy parameters ........................  82
   5.1.9  Computing with trusted mappers ......................  83

5.2  Enforcing mandatory access control ....................  83
   5.2.1  SELinux policy .....................................  83
   5.2.2  Timing channels ....................................  85
Chapter 6  Airavat Implementation and Evaluation  
6.1  HDFS modifications ................................. 87
6.2  Enforcing sensitivity ................................ 88
6.3  Ensuring mapper independence ...................... 88
6.4  Evaluation ............................................ 90
   6.4.1  Airavat overheads ............................... 90
   6.4.2  Queries on AOL dataset ......................... 92
   6.4.3  Covariance matrices .............................. 93
   6.4.4  Clustering Algorithm: k-Means .................. 95
   6.4.5  Classification algorithm: Naive Bayes .......... 95

Chapter 7  Related Work  
7.1  Mandatory access control ............................ 97
7.2  Information flow control ............................ 98
   7.2.1  Language-based DIFC ............................ 98
   7.2.2  OS-based IFC .................................. 98
   7.2.3  Integrating language and OS-based security .... 101
   7.2.4  Privilege separation ............................ 101
7.3  Covert channels .................................. 101
7.4  Differential privacy ................................. 102
7.5  Alternative definitions of privacy .................... 103
7.6  Quantifying information flow ....................... 105

Chapter 8  Conclusion  

Appendix A  Value of δ for the Threshold Reducer  

Bibliography  

Vita  

xii
List of Tables

3.1 Execution time in microseconds of several lmbench OS microbenchmarks, and overhead incurred by using Laminar. Lower is better. ............................................................. 48

3.2 Details of the various applications, including lines of code, the data that needs to be secured, the lines of code that had to be added to secure the application using Laminar and the fraction of time spent in security regions. ............................................. 50

3.3 The security sets associated with the principals and data objects in GradeSheet. $S$, $I$ and $C$ stand for security, integrity and capability sets. Student$(i)$ and TA$(j)$ refer to the $i^{th}$ student and $j^{th}$ teaching assistant, respectively. ............................................. 51

4.1 Parameters and components provided by different participants. Optional parameters are starred. ............................................................. 63

5.1 SELinux domains defined in Airavat and their usage. .................. 84

6.1 Details of the benchmarks, including the grouping of data, type of reducer used, number of MapReduce phases, and the accuracy metric. 90

6.2 Airavat performance details. ....................................................... 91
7.1 Issues for DIFC systems. Laminar offers better functionality than the combination of PL and OS solutions. . . . . . . . . . . . . . . . . . . . 100
List of Figures

2.1 Design of Laminar. Unlabeled objects have an implicit empty label. Trusted components are shaded. ........................................ 16

2.2 Laminar library API. LabelType denotes the secrecy or integrity label. CapType denotes the plus, minus or both capabilities for a given tag. The API also has wrapper functions (not shown) for the new system calls introduced in Laminar OS. ........................................ 18

2.3 Laminar system calls. The tag.t and capability.t types represent a single tag or capability, respectively. The struct label type represents a set of tags that compose a label, and the capList.t type is a list of capabilities. ........................................ 18

2.4 Example pseudocode to read and update a calendar. The thread (not shown) has the required capabilities $a^+, a^-, b^+$ and $i^+$ to initialize the security region. ........................................ 21

2.5 Pseudocode that shows how implicit flows are handled with secure/- catch. ........................................ 22

2.6 Pseudocode that leaks data via a termination channel. .............. 23

2.7 Pseudocode of a timing channel attack. With high probability the value of $L$ printed is the same as the secret $H$. .................. 24

2.8 Pseudocode that leaks data via synchronization. ...................... 25
3.1 Example code to read the marks of two students. The student1 and student2 objects are labeled. The object credentials contains the secrecy, integrity, and capabilities sets with which the security region is initialized. .................................................. 35
3.2 Example of allocating objects and using locals inside a security region 38
3.3 An example to illustrate how Laminar prevents information leaks through references. .................................................. 40
3.4 Laminar VM overhead on programs without security regions. ....... 47
3.5 Overhead of executing applications retrofitted with Laminar. ....... 49
3.6 Comparison of the execution time of GradeSheet and Calendar when security regions (SR) are used with and without a global lock (lower is better). .................................................. 54

4.1 High-level architecture of Airavat. .................................. 60

5.1 Simplified overview of range enforcers and noise generation. Trusted components are shaded. ................................. 75
5.2 Simplified pseudo-code demonstrating differential privacy enforcement. .. 76

6.1 Normalized execution time of benchmarks when running on Airavat, compared to execution on Hadoop. Lower is better................. 91
6.2 Effect of privacy parameter on the number of released AOL search queries .................................................. 93
6.3 Effect of privacy parameter on the accuracy in RMSE in kNN recommender system (lower is better) .................................. 94
6.4 Effect of privacy parameter on the accuracy in k-Means and Naive Bayes (higher is better). .................................................. 96

xvi
Chapter 1

Introduction

Computer systems should safeguard users’ data. Notwithstanding decades of research, security violations still occur. Many security violations occur because untrustworthy software is able to access sensitive data. For example, users may store sensitive information like credit card and social security numbers on the same desktops on which they run applications like web browsers or email clients. Such applications may have vulnerabilities that lead to security breaches. A widespread approach to protect data is to constrain software, which may have security vulnerabilities, and force it to conform to the security policies.

Current security policies and enforcement mechanisms are typically sprinkled throughout an application, making policies difficult to express, change, and audit. This scattered approach makes it hard to reason about the overall security guarantee provided by the application. For example, operating system security abstractions, such as file permissions and user identifiers, are too cumbersome to express a high level policy such as protecting a user’s financial data from an untrusted browser plug-in. Similarly, for cloud computing applications, where the input belongs to multiple distrusting parties, it is difficult to express security policies that would allow useful computations and still guarantee privacy to individual participants.
Even today, we lack frameworks that allow programmers to write security policies at the desired abstraction, and guarantee end-to-end policy enforcement.

This dissertation proposes two frameworks that dramatically simplify the programming task of securing desktop and cloud computing applications, and protecting sensitive data from untrusted software.

The Laminar framework enforces information flow control in order to build secure Java applications running on a single machine. The key insight in Laminar is to combine programming language and operating system techniques to constrain information flows through software components. For large scale distributed computations, the Airavat framework provides provable privacy guarantees to the data owners participating in the computation. Airavat achieves end-to-end security by merging access control restrictions with mechanisms to regulate the effect of the input data on the output of the computation. Programmers can use these two frameworks to create their own applications with modest, and sometimes even no security expertise.

1.1 Constraining information flows

Recent work has shown that the decentralized information flow control (DIFC) model can express and reason about complex security policies [47, 63, 79, 82]. The DIFC model provides security by associating labels with data and restricting the flow of information according to these labels. DIFC policies are based on how data flows through application components and more naturally match how developers and users think of security policies than traditional security mechanisms. For instance, traditional access control mechanisms are all-or-nothing; once an application has the right to read a file, it can do anything with that file’s data. In contrast, DIFC enforces more powerful policies, such as permitting an application to read a file, but disallowing the broadcast of the contents of that file over an unsecured
network channel. A DIFC implementation dynamically or statically enforces end-to-end, user-specified security policies by tracking information flow throughout the system.

As an example of the DIFC model, consider Alice and Bob who want to schedule a meeting while keeping their calendars mostly secret. Alice and Bob each place a secrecy label on their calendar file, and then only a thread with those secrecy labels can read it. Once a thread has a secrecy label, it has been tainted by that label, and can no longer write to an unlabeled output channel, such as standard output or the network. If the thread has the capability to declassify the information, it may remove the secrecy label and then write the data to an unlabeled output. In the calendar example, the thread obtains both Alice’s and Bob’s secrecy label to read their calendar files, but then the thread cannot remove the labels. When the thread is ready to output an acceptable meeting time, it must call a function that then declassifies the result. The application programmer needs to audit only the declassification function for correctness, and to ensure that the output contains no secret information.

DIFC provides two key advantages. First, information flow provides clear rules of legal propagation of data. In the calendar example, the label is tied to the data, and it restricts who may access the data. The secrecy labels ensure that a program that can read the data cannot leak it, whether accidentally or intentionally. Second, security policy decisions are localized and hence easier to audit. For example, the decision to declassify is localized to a small piece of code that can be audited. The result is a system where security policies are easier to express, maintain, and modify.

DIFC can be supported at architecture [78, 83], operating system [47, 79, 82], or language level [63, 64]. Each approach has strengths and limitations. Architecture-based solutions track labels of data in hardware and signal violations, but still require
trusted software to manage the labels. Operating system based DIFC systems have trouble enforcing data flow rules through program data structures because page mappings are an inefficient mechanism to control permissions for user-defined data structures. Language-based DIFC systems rely on extensive type system changes, wrap standard libraries, augment types throughout the entire program, and modify the program structure. Such intrusive changes result in programs being re-written, thus deterring adoption. Furthermore, programming language constructs and operating system security mechanisms are poorly integrated in current systems, complicating the expression and enforcement of security policies. Consider the policy that a user’s credit card number should not be broadcast on the network, whether the number originates in a file or a data structure. Files and data structures are currently governed by completely distinct security mechanisms, requiring developers to understand both mechanisms to enforce this policy. To address these shortcomings we propose Laminar which is a new framework that implements the decentralized information flow control model.

**Laminar overview.** The aim of Laminar is to provide a common security abstraction and labeling scheme for program objects and operating system resources, such as files and sockets. By combining the strengths of programming language and OS techniques, while minimizing their limitations, developers express comprehensive security policies which are then enforced by the Laminar virtual machine and operating system.

Laminar can be incrementally deployed—developers need only modify the security-sensitive portion of their programs. For example, in the calendar application, only the data structure for calendar entries is modified and marked as labeled by the programmer. The changes required are proportional to the amount of security-sensitive objects. In addition, trusted and untrusted threads, labeled and unlabeled files, sockets, data structures, and so on, coexist in the system.
Laminar introduces lexically scoped security regions. Security regions are used by the programmer to specify the security policy in form of information flows. The programmers enclose code within the security regions and give the code the privilege to access labeled data. Security regions also reduce the overhead of dynamic security checks. A typical security region might read a labeled configuration file and parse it into a labeled data structure. Since all code that manipulates the labeled configuration data structure resides in a security region, it is easier to identify and audit. Code unrelated to the data structure needs no modification.

Initial experience with Laminar shows that applications secured using this framework incur low overheads and application developers can express fine-grained security policies with modest changes to existing code.

1.2 Regulating information content

In many cases just regulating the flow of information is not sufficient to provide security guarantees while maintaining the usefulness of the application. In the cloud computing environment, consider a medical patient who is deciding whether to participate in a large health-care study. First, she may be concerned that a careless or malicious application operating on her data may expose it—for instance, by writing her data into a world-readable file which will then be indexed by a search engine. Second, she may be concerned that even if all computations are done correctly and securely, the result itself, e.g., aggregate health-care statistics computed as part of the study, may leak sensitive information about her personal medical record.

Information flow control, or any other form of mandatory access control, can solve the first concern of the participant. But it is not enough to achieve end-to-end privacy in environments where the input data may originate from multiple sources. The output of the computation may also leak sensitive information about the inputs. Since the output generally depends on all input sources, mandatory access control
mechanisms require that only someone who has access rights to all inputs should have access rights to the output. Enforcing this requirement would render the output unusable for most purposes. To be useful, the output of an aggregate computation must be “declassified,” but only when it is safe to do so, i.e., when it does not reveal too much information about any single input. Existing access control mechanisms simply delegate this declassification decision to the implementer. In the case of untrusted code, there is no guarantee that the output of the computation does not reveal sensitive information about the inputs.

To achieve meaningful privacy guarantees, this dissertation proposes the Airavat framework that combines access control rules with mechanisms that regulate the effect of the input on the output of the execution.

**Airavat overview.** Airavat is a system for distributed computations which provides end-to-end confidentiality, integrity, and privacy guarantees using a combination of mandatory access control and differential privacy. Airavat consists of the MapReduce framework, a distributed file system, a modified JVM, and SELinux. Its programming model is based on the popular MapReduce framework [22], and provides a familiar interface to developers. Differential privacy is a new methodology for ensuring that the output of aggregate computations does not violate the privacy of individual inputs [27]. It provides a mathematically rigorous basis for declassifying data in a mandatory access control system. To achieve privacy, differential privacy mechanisms add some random noise to the output of a computation, usually with only a minor impact on the computation’s accuracy.

Airavat enables secure, privacy-preserving, large-scale, distributed computations. With Airavat, programmers can write applications for a variety of scenarios including cloud-based computing services, genomic analysis, outsourced data mining, and clickstream-based advertising.
1.3 Contributions

This dissertation makes the following contributions:

• The design of Laminar, the first system with unified programming language and operating system mechanisms for enforcing DIFC. It introduces security regions, an intuitive primitive that eases deployment, security programming, implementation, and auditing.

• An implementation of Laminar that makes modest additions to Jikes RVM [10] (a Java virtual machine), and the Linux operating system. 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 ranging from 1% to 56%.

• The design of Airavat that provides end-to-end enforcement of the user’s privacy policies for large scale, distributed computations. Airavat brings out the synergy between access control and differential privacy: if a computation is differentially private, the access control restrictions can be safely relaxed.

• An implementation of the first secure and privacy preserving MapReduce framework. Four different data mining applications, written using Airavat, that are privacy preserving and have incur less than 33% performance overhead.

1.4 Dissertation outline

This dissertation is structured as follows. The next chapter describes the new information flow abstractions proposed in Laminar. Chapter 3 provides implementation details of Laminar and evaluates applications that use the framework. Chapter 4 describes the security challenges in the cloud computing environment. It gives a
brief overview of how Airavat solves these challenges, including a discussion on dif-
ferential privacy to aid understanding. Chapter 5 describes the mechanisms used
in Airavat to enforce differential privacy and mandatory access control for MapRe-
duce computations. Chapter 6 includes details of the implementation of Airavat,
and experiences of using the framework to write privacy preserving data mining
computations. Chapter 7 surveys related work. Chapter 8 summarizes this thesis.
Chapter 2

A Common Abstraction for Information Flow Control

Laminar introduces lexically scoped security regions as the unified abstraction for controlling information flows through application data structures and system resources. Programmers use this abstraction to write decentralized information flow control (DIFC) [63] policies for their applications which are then enforced by the Laminar runtime and operating system.

This chapter describes the DIFC model used by Laminar. It delves into the details of the Laminar design including the programming model, security regions and the division of work between the Java virtual machine and the operating system to enforce the security policy.

2.1 DIFC model

All DIFC systems need a mechanism to denote the sensitivity of information and the privileges of the participating users. This section describes the mechanisms used by Laminar and the DIFC rules that determine safe information flows.
In DIFC systems, the security policy is defined in terms of principals that read and write the data in the system. Examples of principals in DIFC systems are users [64], processes [47] and kernel threads [82]. Principals in Laminar are kernel threads.

### 2.1.1 DIFC abstractions

Standard DIFC abstractions include tags, labels, and capabilities. Tags are short, arbitrary tokens drawn from a large universe of possible values ($T$) [47]. A tag has no inherent meaning. A set of tags is called a label.

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

Principals assign labels to data objects. Data objects include program data structures (e.g., individual objects, arrays, lists, hash tables) and system resources (e.g., files and sockets). Previous OS-based systems limit principals to the granularity of a process [47, 79] or support threads by enforcing DIFC rules at the granularity of a page [82]. Our system is the first to expand principals to support threads as principals and enforce DIFC at object granularity.

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

A partial ordering of labels imposed by the subset relation forms a lattice [23]. At the bottom of the lattice are unlabeled resources, which have the empty label for security and integrity. An implicit empty label means that the program need not label every data structure, nor does the OS need to label every file in the file system. Allowing implicit empty labels makes Laminar easier to deploy incrementally.

A principal may change the label of a data object or principal if and only if it has the appropriate capabilities, which generalize ownership of tags [63]. A principal \( p \) has a capability set, \( C_p \), that defines whether it has the privilege to add or remove a tag. For each tag \( t \), let \( t^+ \) and \( t^- \) denote the capability to add and remove the tag \( t \). The capability \( t^+ \) allows a principal to \textit{classify} data with secrecy tag \( t \), while the \( t^- \) capability allows it to \textit{declassify} data. Classification raises data to a higher secrecy level; declassification lowers its secrecy level. Principals can add \( t \) to their secrecy label if they have the \( t^+ \) capability. If the principal adds \( t \), then we call it \textit{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 [50, 74].

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

Note that the capability set $C_p$ is defined on tags. A tag can be assigned to a secrecy or integrity label. In practice, a tag is rarely used for both purposes. $C_p^-$ is the set of tags which principal $p$ may declassify (drop endorsements), and $C_p^+$ is the set of tags that $p$ may classify (endorse). Principals and data objects have both a secrecy and integrity label; a data object with secrecy label $s$ and integrity label $i$ is written: $\{S(s), I(i)\}$. An empty label set is written: $\{(), ()\}$. The capability set of a principal that can add both $s$ and $i$ but can drop only $i$ is written: $\{C(s^+, i^+, i^-)\}$.

2.1.2 Information flow rules

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

**Secrecy rule.** Bell and LaPadula [12] introduced the simple security property and the *-property for secrecy. These properties enforce that no principal may read data at a higher level (no read up) or write data to a lower level (no write down). Expressed formally, information flow from $x$ to $y$ preserves secrecy if:

$$S_x \subseteq S_y$$

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

**Integrity rule.** The integrity rule constrains who can alter information [13] and restricts reads from lower integrity (no read down) and writes to higher integrity (no write up). 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 a label. Laminar requires that the principal must explicitly change its current labels. Zeldovich et al. show that automatic, or implicit, label changes can form a covert storage channel [82].

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

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

### 2.1.3 Calendar example

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


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

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

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

**Discussion.** In this example, Alice specifies a declassifier as a small code module that can be loaded into a larger server application, which can be completely ignorant of DIFC and requires no modifications to work with Alice’s DIFC-aware module. For previous DIFC systems, this example would be more cumbersome. OS-based DIFC systems would require the declassifier to run as a separate process. Language-based systems would require the entire application to be annotated for DIFC enforcement. By integrating OS and language techniques, Laminar substantially improves the state of the art in DIFC. We provide an in-depth comparison of Laminar with other DIFC systems in the related work (Chapter 7).

### 2.2 Laminar design

This section describes how Laminar enforces DIFC in an enhanced VM and OS. Figure 2.1 illustrates the Laminar architecture. The VM enforces DIFC rules within the application’s address space. The OS security module mediates accesses to system resources. Only the VM and the OS are trusted in our model.

For example, Alice may write a program in Java using the Laminar API to label her data. Alice’s program uses the same label namespace present in the file system: it can read data from a labeled file into a data structure with the same label. She compiles the code using a standard, untrusted, bytecode generator such as `javac`. The Laminar just-in-time (JIT) compiler and VM execute the bytecode, and the Laminar OS executes the Laminar VM. Laminar ensures that any accesses or modifications to labeled data follow the DIFC rules and occur in a security region,
Figure 2.1: Design of Laminar. Unlabeled objects have an implicit empty label. Trusted components are shaded.

2.2.1 Enforcement mechanism

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

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

### 2.2.2 Programming model

Laminar provides language extensions, a new security library, and new security-related system calls. The `secure` keyword is used to lexically scope a security region. Figure 2.2 depicts the library API, which includes tag creation, declassification, and label queries. The Laminar OS exports security system calls to the trusted VM for capability and label management, as shown in Figure 2.3. An untrusted application may directly use these system calls to manage its capabilities and labels.

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

Label getCurrentLabel(LabelType t)
Return the current secrecy or integrity label of the security region.

Tag createAndAddCapability()
Create a new tag and add both capabilities to the current principal.

void removeCapability(CapType c, Tag name, boolean global)
Drop the given capability from the current principal.
Setting the global flag drops a capability permanently, whereas not setting it drops the capability for the scope of a security region.

Object copyAndLabel(Object o, Label l)
Return a copy of the object o with new label l.

Figure 2.2: Laminar library API. LabelType denotes the secrecy or integrity label. CapType denotes the plus, minus or both capabilities for a given tag. The API also has wrapper functions (not shown) for the new system calls introduced in Laminar OS.

Laminar System Calls.

tag_t alloc_tag(capList_t &caps)
Return a new tag, add the plus and minus capabilities to the calling principal, and write the new capabilities into caps.

int set_task_label(tag_t l, int op, int type)
Set the type (secrecy or integrity) label of the current principal.

int drop_label_tcb(pid_t tid)
Drop the current temporary labels of the thread without capability checks. Can be called only by threads with the special integrity tag.

int drop_capabilities(capList_t *caps, int tmp)
Drop the given capabilities from the current principal. Tmp is a flag used to suspend a capability for a security region or during a fork().

int write_capability(capability_t cap, int fd)
Send a capability to another thread via a pipe.

int create_file_labeled(char* name, mode_t m, struct label *l)
Create a labeled file with the given labels.

int mkdir_labeled(char* name, mode_t m, struct label *l)
Create a labeled directory with the given labels.

Figure 2.3: Laminar system calls. The tag_t and capability_t types represent a single tag or capability, respectively. The struct label type represents a set of tags that compose a label, and the capList_t type is a list of capabilities.
2.2.3 Security regions

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

Intuitively, security regions are used by the programmer to specify the security policy for the enclosed code. Only code within a security region can access labeled data. Security regions demarcate the code regions that are security sensitive, easing the programmer's burden when adding security policies to existing programs. The programmer is required to wrap the pieces of code that touch labeled data in a security region, such as a routine that reads a sensitive file into a data structure. Usually only a small portion of code and data in a program is security sensitive, so security regions also simplify the task of auditing security-sensitive code. Security regions limit the effects of implicit information flows (Section 2.2.3), and make the DIFC implementation more efficient by reducing the amount of code that requires full DIFC checks.

Requiring threads to access labeled data within security regions limits the amount of work the VM and compiler must do to enforce DIFC, provided that a substantial portion of the execution time is spent operating on unlabeled data. Every time the program reads or writes labeled objects or OS resources within a security region, the system must check the information flow with respect to the current labels of the thread executing within the region. For example, an assignment $w = r$ inside a security region $R$ is safe if and only if the information flow from $r$ to the thread inside $R$ and from the thread to $w$ is legal. Note that the Laminar library API (Figure 2.2) does not include a routine for adding labels to a thread. In order to
add labels, a thread must start a security region.

Example

Figure 2.4 depicts code where the calendar server reads a file belonging to Alice, adds an event to the common calendar, and exports the common meeting schedule for Bob. As shown, the data structure cal has the secrecy tags \(a\) and \(b\). The thread entering the security region is initialized with the secrecy label \(S(a, b)\) and therefore it can read secret data guarded by these tags. Its integrity label \(i\) restricts it from reading data that is not tagged with \(i\). The thread has the capability \(C(a^-)\) to declassify tag \(a\). The assignment at line L1 is a valid information flow because the label of the thread executing at L1 is \(\{S(a, b), I(i)\}\), which is more restrictive than \(f\). (We adhere to the convention that \(a\) is the label of \(a\)).

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

Security region initialization

Laminar enforces certain rules when a thread enters a security region. Let \(S_R\), \(I_R\), and \(C_R\) be the secrecy, integrity, and capability sets of a security region, \(R\). Similarly, let \(S_P\), \(I_P\), and \(C_P\) be the sets associated with a kernel thread \(P\), that enters and then leaves \(R\). Laminar supports arbitrary nesting of security regions.
Calendar cal; // has labels {S(a,b), I(i)}
Output ret; // has labels {S(b), I(i)}
File f; // has labels {S(a), I(i)}

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

Figure 2.4: Example pseudocode to read and update a calendar. The thread (not shown) has the required capabilities a\(^+\), a\(^-\), b\(^+\) and i\(^+\) to initialize the security region.

P could, therefore, already be inside a security region when it enters R. When the thread enters the security region, the following rules hold:

\[
\begin{align*}
(S_R - S_P) & \subseteq C_P^+ \quad \text{and} \quad (S_P - S_R) \subseteq C_P^- \\
(I_R - I_P) & \subseteq C_P^+ \quad \text{and} \quad (I_P - I_R) \subseteq C_P^- \\
C_R & \subseteq C_P
\end{align*}
\] (2.1) (2.2) (2.3)

The first two rules ensure that the principal P can legally change its labels to those of the security region. The third rule states that the principal P can only retain a subset of its current capabilities when it enters a security region.

The above rules encapsulate the common sense understanding that a parent principal, P, has control over the labels and capabilities it passes to a security region, and that the system will not let the principal create a security region with security properties that the principal itself lacks. The rules also state that security regions nest in the natural way based on the labels and capabilities of the thread entering the nested region.
// H, x, and y has labels {S(h), I()}
// L has labels {S(), I()}
// Invariant: y == 2x
L = false;
secure ({S(h), I(), C()}) {
    x++;
    if (H) L = true;
    y = 2 * x;
    ...
} catch (...) {
    y = 2 * x;
}

Figure 2.5: Pseudocode that shows how implicit flows are handled with secure/catch.

Implicit information flows

A major benefit of security regions is that they limit the amount of analysis necessary to restrict implicit information flows. Implicit information flow leaks secret data through control flow decisions [24]. For example, the code in Figure 2.5 shows an implicit flow from the control variable H to the data variable L. By looking at the value of L, a thread can deduce the value of H. Since L is low secrecy and H is high secrecy, this implicit flow is a violation of DIFC rules. The system must therefore detect and prohibit this flow.

Laminar has a special construct to limit implicit flows—each security region has a required catch block as shown in Figure 2.5. The catch block executes with the same labels as the security region, and the capability set at the time of the exception. For instance, if H is true and the program attempts assignment to L, then Laminar raises an exception because the security region does not have the right to declassify H. The catch block gives the programmer a chance to restore program invariants before exiting the security region. No flow occurs between H and L because L is never assigned, regardless of whether H is true or false, and control flow continues
after the secure/catch blocks. The VM suppresses all exceptions inside a security region that are not explicitly caught, including exceptions within a catch block. The VM continues execution after the security region.

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

Laminar thus eliminates implicit flows by hiding the control flow of a security region from code outside of the security region. In Figure 2.5, code outside of the security region cannot distinguish an execution where $H$ is true from one where it is false. In contrast, DIFC systems that rely on static analysis prevent these flows by detecting them during compilation [63].

### 2.2.4 Covert channels

Apart from explicit and implicit flows an adversary may try to leak information covertly through timing and termination channels [48]. Both Laminar and static analysis DIFC systems assume that programs (or in Laminar’s case, security regions) terminate. Figure 2.6 shows an example of an implicit flow via a termination channel, that leaks secret information based on whether the application terminates.

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

Figure 2.6: Pseudocode that leaks data via a termination channel.
// H has labels {S(h), I()}
// L has labels {S(), I()}
L = 1;

/*Thread 1*/
secure ({S(h), I(), C()}) {
    if (H==1) Thread.sleep(5000);
} catch (...) {
} L=0;

/*Thread 2*/
secure ({S(h), I(), C()}) {
    Thread.sleep(1000);
} catch (...) {

} System.out.println(L);

Figure 2.7: Pseudocode of a timing channel attack. With high probability the value of L printed is the same as the secret H.

If control returns from this security region, then unprivileged code can learn that \( H \) is false. Similarly, a colluding application might learn that \( H \) is true if the application must be explicitly killed. OS-based DIFC systems eliminate termination channels by ensuring that no one with whom communication is prohibited is notified of thread termination.

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

In practice, covert channels have been low bandwidth and difficult to exploit. However, in a multithreaded environment, exploitation might be significantly easier. Figure 2.7 shows a multithreaded program that leaks information by inserting timing delays depending upon a secret value. In this example, Thread 1 sleeps for five seconds inside the security region if the secret value is 1, while Thread 2 always sleeps for one second. If the secret, \( H \), is 1 then Thread 1 would sleep for 5 seconds which, in most cases, is enough for Thread 2 to complete its execution of the security
Figure 2.8: Pseudocode that leaks data via synchronization.

region and print the value of L as 1. Thus, with high probability, information would be leaked because the value of L printed will be same as H.

Figure 2.8 shows another attack which deterministically leaks one bit of information after the security region has been executed. In this example, signal is a variable with empty labels. If the secret, H, is 1 then Thread 2 gets into a while loop until Thread 1 exits its security region and sets the value of signal to 1. Thus at the end of the security region the value of signal is the same as that of secret H.

The key idea used in these attacks is that the order in which the security regions exit can be exploited as a channel to leak information. In both the examples the order of execution is dependent on a secret value. In Figure 2.7 the order of execution is modified using a delay dependent on the secret value. In Figure 2.8 the order of execution is implicitly stored in the variable signal and communicated to another thread.

Language based DIFC systems like Jif are also prone to such attacks. Laminar mitigates some of these attacks and reduces the bandwidth of others. Laminar enforces that threads exits security regions in the order that they start the security region. By fixing the order of exit, Laminar renders the attack in Figure 2.7 useless. The attack in Figure 2.8 is still possible, but the leak occurs with half probability and
the threads deadlock otherwise. The bandwidth of the attack is, therefore, reduced substantially. We refer to this covert channel as a half-termination channel. Instead of fixing the order, another possible approach is to ensure that security regions execute with isolation semantics similar to those in transactional memory [41, 69, 71]. Isolation would render the attack in Figure 2.8 void because a thread inside a security region will not read concurrent updates to the signal variable.

2.2.5 VM-OS interface

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

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

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

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

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

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

The Laminar language API provides a method, `removeCapability`, that removes a thread’s capability in the VM, which then calls the `drop_capability` system call to notify the kernel. The `removeCapability` method takes an argument, `global`, that allows the user to specify whether the capability should be dropped only for the scope of the security region or permanently (i.e., globally). Globally dropped capabilities are not restored to the parent when the thread exits a security region. The VM uses the `set_task_label` system call to change the label of a thread at the beginning and end of a security region. This function has no user API; it is called solely by the VM at the entry and exit of security regions. Laminar does not allow security regions to change their labels, because the VM relies on labels staying the same throughout lexically scoped regions to prevent leaks through local
variables, as discussed in Section 3.1. To change labels in the middle of a security region, a thread may begin a nested security region.

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

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

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

2.2.6 Labeling data

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

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

2.2.7 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 memoized its result in a security region with one label, a later call with a different label may attempt to return the memoized value. Because the memoized result is secret, the attempt to return it will be prevented by the system. Such code must be modified to work in any DIFC system.

2.2.8 Trusted computing base

To implement Laminar, we added approximately 2,000 lines of code to Jikes RVM [5] a 1,000 line Linux security module, and 500 lines of modifications to the Linux kernel. This relatively small amount of code means that Laminar can be easily audited.

We rely on the standardization of the VM and the OS as the basis of Lam-
inar’s trust. In addition to trusting the base VM, Laminar requires that the VM correctly inserts the appropriate read and write instrumentations called barriers for all accesses and optimizes them correctly. Read and write barrier insertion is localized and standard in many VMs. In Linux, Laminar assumes that the kernel has the proper hooks to call into Linux security modules (LSM). Because many projects rely on LSMs, the Linux code base is under constant audit to make sure all necessary calls are made.
Chapter 3

Laminar Implementation and Evaluation

Laminar enforces DIFC policies with its modified JVM and the Linux operating system. The changes in the JVM provide the functionality to create labeled objects and perform information flow checks on the application data structures. In the operating system, our main addition is a linux security module based reference monitor that performs access checks on system resources.

This chapter describes our implementation of Laminar. It also discusses our experiences with retrofitting DIFC policies on four application case studies, and experiments to determine the overhead of security enforcement.

3.1 JVM support

We implement Laminar’s trusted VM in Jikes RVM 3.0.0 [5], a high-performance Java-in-Java virtual machine [10]. Our implementation of Laminar is publicly available on the Jikes RVM Research Archive.\footnote{http://www.jikesrvm.org/Research+Archive} As of August 2008, Jikes RVM’s perfor-
mance compared well with commercial VMs: the same average performance as Sun HotSpot 1.5; and 15–20% worse than Sun HotSpot 1.6, JRockit, and J9 1.9 [3]. All subsequent uses of JVM refer to the Laminar-enhanced version of Jikes RVM.

The JVM controls information flow by ensuring that all program accesses to labeled data occur in security regions. The JVM adds instrumentation called barriers at every object read and write. These barriers check at run time that accesses conform to the DIFC rules in Section 2.1.

**Starting a security region.** When a thread starts a security region, the JVM checks whether it has the capabilities to initialize the security region with the specified labels and capabilities, as described in Section 2.2.3. Thread capabilities are stored in the kernel. The JVM then caches a copy of the current capabilities of each thread to make the checks efficient inside the security region.

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

Because the lifetime of local variables is typically short, and tracking their labels would be expensive, our prototype restricts the programming model. Laminar statically (during JIT compilation) enforces the following restrictions on local variables: (1) a local variable written inside a security region may not later be read outside that security region if the region has secrecy labels, and (2) a local variable already written outside a security region may not be read inside the region if the region has integrity labels. Because a security region’s labels are dynamic, for simplicity our implementation requires both properties for every security region.

An exception to the above rule is a local reference. They can always be dereferenced inside the security regions and operations can be performed on the
dereferenced values. In addition, references that are declared as `final` can be assigned any non-null value inside the security region. Therefore, in the general case, read and write operations on the reference (such as the test `if(obj==null)`) are not allowed inside the security region.

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

The JVM restricts information flow to and from static variables. The Laminar prototype implementation prevents security regions with secrecy labels from writing static variables, and prevents regions with integrity labels from reading statics. Compiler-inserted barriers at each access inside a security region enforce these restrictions. A production implementation could support labeling statics with modest overhead because static accesses are relatively infrequent compared to field and array element accesses. The applications in Section 3.6 do not need labeled static variables.

**Supporting information flow for objects.** The JVM tracks information flow for objects that live in the heap. At allocation time, objects may be assigned secrecy and integrity labels that are immutable. By default, objects allocated inside security regions are assigned the labels of the region at the allocation point. The program may specify alternate labels, as long as they conform to DIFC rules. To change an object’s labels, our implementation provides an API call, `copyAndLabel`, that clones...
an object with specified labels. The label change must conform to the label change rule (Section 2.1). The JVM allocates labeled objects into a separate labeled object space in the heap, allowing instrumentation to quickly check whether an object is labeled. We modify the allocator to add two words to each object’s header, which point to secrecy and integrity labels.

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

The JVM’s compiler inserts read and write barrier (instrumentation) \cite{17} into application code to enforce DIFC rules. Inside security regions, the compiler inserts barriers at labeled object allocation (before the constructor call) to set the labels and check that they conform to DIFC rules. It inserts barriers at every read from and write to an object field or array element. Inside security regions, barriers load the accessed objects’ secrecy and integrity \texttt{Labels} and check that they conform to the current security region’s labels and capabilities. Outside security regions, read and write barriers check that the accessed objects are unlabeled (or equivalently, have the empty label). The compiler inserts barriers inside security regions at static accesses to verify that static reads (writes) occur only in regions without integrity (secrecy) labels.

The compiler inserts different barriers at an access depending on whether the access occurs inside or outside a security region. Choosing the right barrier at compile time can be difficult because a method may be called by code inside of and outside of a security region. In our prototype implementation, when a method first
// credentials = \{S(s_1, s_2), I(), C(s_1^-, s_2^-)\}
// credentialsNew = \{S(), I(), C(s_1^-, s_2^-)\}
// newLabel = \{S(), I()\}
[L1] secure(credentials){
[L2] int m1 = student1.marks;
[L3] int m2 = student2.marks;
[L4] MyObject obj = new MyObject(m1+m2);
[L5] secure(rewrite(credentialsNew)){
[L6] ret.val = Laminar.copyAndLabel(obj, newLabel);
   ...
}

Figure 3.1: Example code to read the marks of two students. The student1 and student2 objects are labeled. The object credentials contains the secrecy, integrity, and capabilities sets with which the security region is initialized.

executes and the compiler compiles it, the compiler checks whether the thread is in a security region and inserts barriers accordingly. Subsequent recompilation at higher optimization levels reuses this decision. This approach, which we call static barriers, fails if a method is called from both within and outside a security region. We also support a configuration where the compiler adds dynamic barriers that check whether the current thread is in a security region or not, and then executes the correct barrier. A production implementation would use cloning to compile two versions of each method executed from both contexts. The same approach is used in prior work on software transactional memory [67]. Static barriers add the same overhead that cloning would achieve.

Because object labels are immutable and security regions cannot change their labels, repeated barriers and checks on the same object are redundant. We implement an intraprocedural, flow-sensitive data-flow analysis that identifies redundant barriers and removes them. A read (or write) barrier is redundant if the object has been read (written), or if the object was allocated, along every incoming path. Although the optimization is intraprocedural, the compiler inlines small and hot methods, increasing the scope of redundancy elimination.
**Example.** The example illustrated in Figure 3.1 computes the sum of the marks obtained by two different students. The `student1` and `student2` objects are labeled and have different secrecy values associated with them. The object `credentials` contains a set of labels and capabilities. If the capabilities and labels inside `credentials` do not conform to the current thread’s capabilities, then the program terminates at L1. Once the security region starts, the thread’s current labels become those present in `credentials`. Lines L2 and L3 are reads of labeled objects that will result in an error if the flow from `student1.marks` or `student2.marks` to the thread in the security region is not allowed. In line L4, the JVM allocates object `obj` with the labels present in `credentials`. The programmer can write to the reference `obj` because it is used only inside the security region. In line L6, the thread attempts to change the labels of the object inside a nested security region. The JVM allows this change because the thread has the required capabilities. Note that the inner security region has an empty secrecy label.

### 3.2 Discussion

This section discusses in detail how labels are associated with language level primitives. We also present informal arguments for the correctness of language level semantics of the Laminar implementation.

#### 3.2.1 Objects

Objects always have labels. The default label is the empty label. Before any access to the fields contained in an object the JVM performs the DIFC checks on the labels of the object and the principal attempting to access it. Access is allowed only if the checks succeed.
**Allocation.** Objects allocated inside a security region get the labels of the security region. Therefore, to create an object with any arbitrary label $L$ it should be allocated inside a security region that has the label $L$. These newly created objects can be passed via references. The assignment to a reference occurs inside the security region. As explained in the next section, only references that have been declared `final` can be assigned inside a security region. In addition, the assignment should be to a non-null object. Laminar enforces these restrictions to prevent illegal information flows. Figure 3.2, line L2, shows how the reference $m$ is assigned a newly allocated object which has the labels \{$S(a), I(b)$\}.

**Objects as containers** The Laminar prototype does not support objects whose different fields have different labels. For example, consider an object pointed to by the reference `obj`. The object has two fields, x and y. To read or write to the values `obj.x` and `obj.y`, DIFC checks are performed on the same object, so both fields have the same labels. References inside an object, like `obj.r`, may point to objects that have labels different from those of `obj`. Each object acts as a security container for its fields and protects the fields from illegal access.

### 3.2.2 Locals

For efficiency reasons, Laminar does not associate labels with locals. To prevent information leaks through locals, we restrict how they can be used: (1) a local variable written inside a security region may not later be read outside that security region if the region has secrecy labels, and (2) a local variable already written outside a security region may not be read inside the region if the region has integrity labels. Thus, a local that is assigned a secret value inside a security region cannot leak the value as Laminar prevents it from been used outside the security region.

An exception to the above rules are local references. They can always be dereferenced inside the security regions and operations can be performed on the
public static void main(..){
[L1] final MyObj m;
    MyObj k;

    //allocate a labeled object
    //m is a final reference and can
    //be assigned inside the security region
    secure(S(a),I(b),C()){
    [L2]    m = new MyObj();
    }
[L3]    k = m;

    secure(S(a), I(b), C()){
    //n is a local reference and cannot
    //be used outside the security region
    [L4]    MyObj n=new MyObj(20);

    //dereference and update m
    [L5]    m.val = n.val+5;
    }
}

Figure 3.2: Example of allocating objects and using locals inside a security region
dereferenced values. In addition, references that are declared as final can be assigned any non-null value inside the security region.

Figure 3.2 shows an example of a local m that is used outside of a single security region. Since m is declared final in line L1, it can be assigned at line L2 inside the first security region. However, after the initial assignment we only use the dereference operation on m at line L5 inside the security region. The local n has the scope of the security region, so we can read/write or perform any operation on it since n is never used outside the security region.

**Security implication**  It is easy to see that by ruling out the use of the same local both inside and outside a security region, we prevent any direct information flows
from within a security region to outside. One may, however, store the value of the local and pass them around through objects. The object then acts as the security container for such locals. Illegal flows are prevented by performing security checks on the label of the object.

Next we argue that inside a security region, allowing any reference to be dereferenced, and assigning references declared as final does not violate the secrecy or integrity rules.

References have two channels through which information can flow. First, the actual content of the object that the reference points to is a channel. Consider the reference \texttt{obj.r}. If \texttt{r} points to a security sensitive object \texttt{m} then we need to protect the contents of \texttt{m} from unauthorized accesses. Here \texttt{m} is a labeled object. Hence any attempt to access or update the contents of \texttt{m}, would be checked for correct flow against the labels of \texttt{m}.

Second, a reference can be used to create a channel by modulating where it points. For example, the adversary may set the value of a reference \texttt{r} to null or non-null to leak a bit of information. To leak information by modulating a reference, the decision to modulate has to depend on labeled data. Since labeled data can be accessed only inside the security region it implies the reference has to be assigned inside a security region.

Laminar enforces that a reference can be assigned (written) inside a security region only if it has been declared as \texttt{final} and the assignment is to a non-null value. The keyword \texttt{final} ensures that the reference can be assigned only once and there can be no further modulation. The restriction on assigning a non-null value makes sure that any checks of the form \texttt{if(r==null)} will always return false if \texttt{r} has been written inside the security region. Thus the assignment of the local reference cannot be used to leak information.

In cases where the reference is contained inside an object, checks on the labels
Figure 3.3: An example to illustrate how Laminar prevents information leaks through references.
of the object prevent illegal flows. In Laminar, writes to objects inside a security region are allowed only when the label of the object is sufficient (according to the lattice) to protect any labeled information read in the security region. Therefore, the information contained in the reference, e.g. obj.r is set to null, is protected by the label of obj.

**Secrecy example.** Figure 3.3 illustrates certain ways in which one may attempt to leak information through a reference and how Laminar prevents such attempts. Let H be a reference to an object that holds secret data. One may try to implicitly leak information about H by declaring a reference m as null and assigning it a new object inside the security region (L3). However, the assignment at line L3 would be caught as a compiler error since m is declared as final and has already been assigned the value null.

Another possibility is to declare a reference n in line L2 and then, in line L4, assign it a non-null object inside a security region depending upon the content of H. Now n can be checked outside a security region (L5) to reveal information about the secret H. The compiler would flag the line L6 as an error reporting that n may not have been initialized. To counter this error n needs to be initialized on all paths that can lead to line L6. If n depends on a secret, it has to be initialized inside the security region on all valid paths that reach L6. Since Laminar enforces that references are assigned only non-null values inside a security region, the conditional at L6 will always be false.

Finally, one may attempt to store the value of H in a reference as in line L5, and then later print out the contents in line L7, which is outside the security region. Since printing the content involves dereference and accessing the contents of a labeled object, it would be prevented by the VM during runtime.
Integrity example. In Figure 3.2, line L3, \( k \) is a reference to a high integrity object. Since Laminar does not track labels of references, Laminar cannot prevent someone outside the security region from changing \( k \) such that it points to an object of low integrity. Such modifications to the reference is not a DIFC violation. Laminar guarantees that a high integrity security region will never read a low integrity value. If \( k \) is dereferenced inside a security region, the label on the object it points to will be a low integrity label. The VM will then raise an exception, thus disallowing the use of the low integrity object inside the security region. Note that Laminar does not allow \( k \) to be directly read inside the security region.

3.2.3 Statics

The Laminar prototype does not associate labels with static variables. However, every time a static is accessed the corresponding read or write barrier is called. These barriers prevent writing to statics inside a security region with non-empty secrecy label and reading statics inside security regions with non-empty integrity labels. It follows that in the general case when a security region has both secrecy and integrity labels, then statics cannot be used inside that region.

In the future, we could remove the restriction on statics by maintaining a hash table that tracks statics and their associated labels. Thus, every time the program accesses a static, a table lookup would determine if the flow is legal (similar to the rules for labeled objects). In addition the hash table could track label on static references as well. Hence, it would be legal to read or write static references inside a security region.

3.2.4 Labels

As explained in Section 3.1, Laminar encapsulates labels into immutable, opaque objects called \texttt{Labels}. Like any other object, the instances of \texttt{Labels} have secrecy
and integrity labels associated with them. If allocated inside a security region, the labels of the Labels objects have to conform to the usual rules that apply to any other object. The rules enforce that the label of the object will be such that the information flow from the security region to the object is legal. In general, the Labels objects have empty secrecy and high integrity labels, so that they can be used as parameters in all security regions.

3.3 OS support

We have implemented support for DIFC in Linux version 2.6.22.6 as a Linux Security Module (LSM) [80]. LSM provides hooks into the kernel customizing authorization rules. We also added a set of system calls to manage labels and capabilities (Figure 2.2). Some LSM-based systems, such as SELinux [53], manage access control settings through a custom filesystem similar to proc. This method is isomorphic to adding new system calls. The Laminar security module contains about 1,000 lines of code, and about 500 lines of modifications to the kernel to support the new system calls.

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

Files. Using LSM, Laminar intercepts inode and file accesses, which are used to perform operations on unopened files and file handles (including sockets and pipes), respectively. The Laminar security hooks perform a straightforward check of the
rules listed in Section 2.1.2. The label of an **inode** protects its contents and its metadata, except for the name and label, which are protected by the label of the parent directory.

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

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

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

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

Pipes. Laminar mediates inter-process communication (IPC) over pipes by labeling the inode associated with the pipe message buffer. A process may read or write to a pipe so long as its labels are compatible with the label of the pipe. Message delivery over a pipe in Laminar is unreliable. An error code due to an incorrect label or a full pipe buffer can leak information, so messages that cannot be delivered are silently dropped. Unreliable pipes are common in OS DIFC implementations [47, 79].

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

### 3.4 Limitations

While Laminar regulates explicit information flows and some 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 know whether a class has been loaded, and use this additional information to leak sensitive data. In multithreaded programs, attackers may collude and use synchronization based timing channels to leak information. Such channels can be mitigated by restricting the behavior of the scheduler [73]. Laminar assumes that code blocks enclosed inside security regions always terminate. Otherwise, as explained in Section 2.2.3, information can be leaked through termination channels.

Laminar does not regulate information flows that may occur through objects in JNI code. JNI code segments bypass the JVM reference monitor that tracks information flow at the language level. Thus, in applications that use JNI, Laminar can only provide information flow guarantees at the granularity of processes.

The current implementation of Laminar limits the use of static variables instead of associating labels with them (Section 3.1). Since statics are used infrequently, an improved implementation could track their labels without affecting the performance results. For efficiency reasons instead of tracking the labels of the references, the Laminar JVM tracks the labels of the objects that are pointed to by the references. This choice of implementation limits how references can be used inside a security region. For example, references cannot be read inside a security region that has integrity labels (Section 3.1). In the absence of efficient mechanisms, removing this limitation and tracking labels for references may increase the overheads of security enforcement.

### 3.5 Laminar overhead

This section reports the overhead of Laminar’s subsystems. The performance loss on Java benchmarks without security regions is 6% or 17%, depending on whether the dynamic compiler compiles separate versions of methods called both from inside and outside security regions. The Laminar OS incurs an overhead of less than 8% on
Figure 3.4: Laminar VM overhead on programs without security regions.

**lmbench** [61]. All experiments, including those in the next section, were conducted on a machine with a quad-core Intel Xeon 2.83 GHz processor. All experiments configure Jikes RVM to run on two cores. All results are normalized to values obtained on unmodified Linux 2.6.22 and Jikes RVM 3.0.0.

### 3.5.1 JVM overhead

Figure 3.4 shows the overhead of our Laminar-enabled JVM for the DaCapo benchmarks [16] and a fixed-workload version of SPECjbb2000 called *pseudojbb* [76]. Each experiment executes two iterations of the benchmark: the first includes compilation, and the second disables compilation and runs only the application. We report the running time of the second iteration. Because compilation decisions are nondeterministic, running times vary, so we execute 10 trials of each experiment and take the median.

The darker bar shows the overhead of dynamic barriers, which check dynamically if they are in a security region. Dynamic barriers add 17% overhead on average. The lighter bar is the overhead of using static barriers, 6% on average. As discussed in Section 3.1, a mature implementation of Laminar would use method cloning and eliminate all dynamic barriers. Because method cloning has comparable overheads to static barriers, code outside of a security region is expected to have an
We also measure compilation time and find that, on average, static barriers double it, and dynamic barriers triple it. However, compilation time is not our primary concern, especially for long-running programs. For these benchmarks, compilation time accounts for just 8-12% of running time on average, making barrier compilation’s effect on run time comparable to barriers’ effect on application execution time. The overhead is high in large part because we instruct the compiler to inline the barriers aggressively, which bloats the code and slows downstream optimizations. To lower compilation time without increasing run time substantially, an implementation could choose to inline less aggressively.

3.5.2 OS overhead

We use the lmbench [61] suite of benchmarks to measure the overheads imposed on unlabeled applications when running on Laminar OS. A selection of the results is presented in Table 3.1.

In general, the overhead of the Laminar OS modifications are less than 8%, which is similar to previously reported overheads for Linux security modules [80]. The only performance outlier is the “null I/O” benchmark, which has an overhead of 31%. This benchmark represents the worst case for Laminar in that the system

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

Table 3.1: Execution time in microseconds of several lmbench OS microbenchmarks, and overhead incurred by using Laminar. Lower is better.
call being measured does little work to amortize the cost of the label check. In comparison, Flume adds a factor of 4-35× to the latency of system calls relative to unmodified Linux [47].

3.6 Application case studies

This section describes four case study applications and how we retrofit them with DIFC security policies. Table 3.2 summarizes the details of the applications. Figure 3.5 shows the overhead of running the modified version with Laminar. The retrofitted applications implement more powerful security policies than their unmodified counterparts, and all modifications are at most 10% of the source.

The figure breaks down the overhead of Laminar into four parts. Start/end SR is the overhead of application modifications to support DIFC, including the starting and ending of security regions and other security operations, such as copy-AndLabel. The Alloc barriers configuration denotes the extra time for allocating labeled objects and assigning their label sets. Static barriers is the overhead from read and write barriers when the security context is known at compile time. Finally, Dynamic barriers is the extra overhead from barriers that check context at run time. We note that Gradesheet and Battleship run correctly with static barriers,
Table 3.2: Details of the various applications, including lines of code, the data that needs to be secured, the lines of code that had to be added to secure the application using Laminar and the fraction of time spent in security regions.

<table>
<thead>
<tr>
<th>Application</th>
<th>LOC</th>
<th>Protected Data</th>
<th>LOC Added</th>
<th>% time in SRs</th>
</tr>
</thead>
<tbody>
<tr>
<td>GradeSheet</td>
<td>900</td>
<td>Student grades</td>
<td>92 (10%)</td>
<td>6%</td>
</tr>
<tr>
<td>Battleship</td>
<td>1,700</td>
<td>Ship locations</td>
<td>95 (6%)</td>
<td>54%</td>
</tr>
<tr>
<td>Calendar</td>
<td>6,200</td>
<td>Schedules</td>
<td>290 (5%)</td>
<td>1%</td>
</tr>
<tr>
<td>FreeCS</td>
<td>22,000</td>
<td>Membership properties</td>
<td>1,200 (6%)</td>
<td>&lt;1%</td>
</tr>
</tbody>
</table>

but Calendar and FreeCS require dynamic barriers because some methods are called from both inside and outside security regions. Method cloning would obviate the need for dynamic barriers (Section 3.1).

In all our experiments, we disabled the GUI, as well as other I/O and network-related operations so that the Laminar overheads are not masked by them. Hence, the slowdown in deployed applications would be less than what is reported in our experiments. For comparison, Flume [47] 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.

3.6.1 GradeSheet

GradeSheet is a small program that manages the grades of students [14]. It uses three types of principals: professors, TAs and students. The main data structure is a two-dimensional object array GradeCell. The \((i, j)^{th}\) object of GradeCell stores the information about student \(i\) and her marks in project \(j\). A sample policy states that (1) the professor can read/write any cell, (2) the TA can read the marks of all students but only modify the ones related to the project that she graded, and (3) students can only view their own marks on any project.

Table 3.3 shows how this policy can be expressed by assigning labels and
capabilities to the data and the principals respectively. Specifically, we guard the 
\((i,j)^{th}\) entry in the GradeCell with the secrecy tag \(s_i\) and the integrity tag \(p_j\). 
Each student \(i\) has the capability to add or remove \(s_i\), so students can read their 
own marks in any project. Each TA \(j\) has the capability to add tags \(s_i\) and the 
integrity tag for the project that she graded \((p_j)\). This tag ensures that TAs can 
read the marks of all students, but the integrity constraint prevents them from 
modifying grades for projects that they did not grade.

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

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

### 3.6.2 Battleship

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

We started with JavaBattle, which is a 1,700-line Battleship program avail-

<table>
<thead>
<tr>
<th>Name</th>
<th>Security Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>GradeCell((i,j))</td>
<td>(S={s_i}, I={p_j})</td>
</tr>
<tr>
<td>Student((i))</td>
<td>(C={s_i^+, s_i^-})</td>
</tr>
<tr>
<td>TA((j))</td>
<td>(C=\bigcup_{i=1}^{n} s_i^+, p_j^+, p_j^-)</td>
</tr>
<tr>
<td>Professor</td>
<td>(C=\bigcup_{i=1}^{n} s_i^+, s_i^-, p_j^+, p_j^-)</td>
</tr>
</tbody>
</table>

Table 3.3: The security sets associated with the principals and data objects in GradeSheet. \(S, I\) and \(C\) stand for security, integrity and capability sets. Student\((i)\) and TA\((j)\) refer to the \(i^{th}\) student and \(j^{th}\) teaching assistant, respectively.
able on SourceForge. Each player $P_i$ allocates a tag $p_i$ and labels her board and the ships with it. The capability $p_i^-$ is not given to anyone else, ensuring that only the player can declassify the locations of her ships. In the original implementation, players directly inspect the coordinates of a shot to determine whether it hit or missed an opponent’s boat. Under Laminar, each player sends her guess to her opponent, who then updates his board inside a security region. The opponent then declassifies whether the guess was a hit or a miss and sends that information back to the first player. We had to add less than 100 lines of code to secure the program to run with Laminar.

In our experiments, the game is played between computers on a $15 \times 15$ grid without a GUI. Figure 3.5 shows that the secured version adds 56% overhead with static barriers. The overhead is high because the benchmark spends almost 54% of its time inside security regions. In a deployed Battleship, which would display the intermediate state of the board to the players, the overhead would be significantly less. In an experiment where we display the shot location after each move, the run time increases, and Laminar overhead drops to 1%.

3.6.3 Calendar

Like in the examples from earlier in the dissertation, we modified the k5nCal [6] multithreaded desktop calendar to label all data structures and .ics files that store a user’s calendar information with the user’s secrecy tag. All functions that access this data are wrapped inside security regions, including a scheduler that finds available meeting times for multiple users. In the original program, a user could view the calendar of other users, a feature we disabled.

Our experiments measure the time to schedule a meeting, which includes reading the labeled calendars of Bob and Alice, finding a common meeting date, and then writing the date to another labeled file that Alice can read. The scheduling
code is executed in a thread that has the capability to read data for both Alice and Bob, but can only declassify Bob’s data. The output file is protected by the label of Alice. Our experiment schedules 1,000 meetings. Figure 3.5 shows that the secured version of Calendar runs 14% slower than unmodified Calendar.

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

3.6.4 FreeCS chat server

FreeCS is an open-source chat server written in Java [4]. 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 commands based on the role of the user. All these policies are written in the form of if..then checks. These authorization checks are actually checks on the role of a user. For example, a user who is in the role of a VIP and has superuser power on a group can ban another user in the group.

We improve the security code in FreeCS by labeling sensitive data structures and accessing them inside security regions. We made most of our modifications in two classes—Group and User. With Laminar, we localized all security checks to these classes. The abstraction of a role maps naturally onto integrity labels. For example, we protect the banList data structure with two tags, one that corresponds to the notion of VIP and the other for the group’s superuser. Now, only users who have the add capability for these two tags can use the ban command. We also changed the authentication module to ensure that users are given the right capabilities when they log in. Our experiments measure the time to process requests
Figure 3.6: Comparison of the execution time of GradeSheet and Calendar when security regions (SR) are used with and without a global lock (lower is better).

from 4,000 users, each invoking three different commands. Laminar’s overhead is less than 1% (Figure 3.5).

3.6.5 Handling covert channels

As mentioned in Chapter 2.2.4, for multi-threaded programs, Laminar can mitigate covert channels that leak information through the execution of order of threads. Figure 3.6 shows the overhead of using a global lock to enforce that security regions exit in the order in which they were started. In GradeSheet the global lock increases the execution time by upto 6%. In Calendar there is almost no effect of the global lock as the overheads are dominated by work done outside the security regions. In the future, we may use compiler optimizations or transactional memory techniques to reduce this overhead.

3.6.6 Application insights

The four case studies reveal a pattern in the way applications are written. First, most applications have only a few key data structures that need to be secured,
like the array of student grades in GradeSheet or the playing boards in Battleship.
Second, the interface to access these data structures is quite narrow. For example, `InternalServer` in GradeSheet and `DataFile` in Calendar contain the functions used to access the important data. These observations support our hypothesis that only localized changes are needed to retrofit DIFC onto many types of applications. Third, most of the data structures require heterogeneous labeling — the single data structure `GradeCell` has different labels corresponding to different students. Heterogeneous labeling is impractical in OS-based systems [47, 79, 82], since they support a single label on the whole address space or require the programmer to map application data structures onto labeled pages. The Laminar VM easily solves this problem with fine-grain tracking of labels on the data structure, for example, individual array elements and objects in GradeSheet.
Chapter 4

Ensuring Privacy in Large Scale Computations

Cloud computing involves large-scale, distributed computations on data from multiple sources. For example, targeted advertisements can be created by mining a user’s clickstream, while health-care applications of the future may use an individual’s DNA sequence to tailor drugs and personalized medical treatments. Widespread adoption of cloud computing is possible only if it supports flexible computations while guaranteeing security and privacy for the input data. To balance the competing goals of a permissive programming model and the need to prevent information leaks, the untrusted code should be confined [48].

The security challenges posed by this new breed of applications require novel solutions. Consider a medical patient who is deciding whether to participate in a large health-care study. First, she may be concerned that a careless or malicious application operating on her data as part of the study may expose it—for instance, by writing it into a world-readable file which will then be indexed by a search engine. Second, she may be concerned that even if all computations are done correctly and securely, the result itself, e.g., aggregate health-care statistics computed as part of
the study, may leak sensitive information about her personal medical record.

Traditional approaches to data privacy are based on syntactic anonymization, \textit{i.e.}, removal of “personally identifiable information” such as names, addresses, and Social Security numbers. Unfortunately, anonymization does not provide meaningful privacy guarantees. High-visibility privacy fiascoes recently resulted from public releases of anonymized individual data, including AOL search logs [38] and the movie-rating records of Netflix subscribers [65]. The datasets in question were released to support legitimate data-mining and collaborative-filtering research, but naïve anonymization was easy to reverse in many cases. These events motivate a new approach to protecting data privacy.

One of the challenges of bringing security to cloud computing is that users and developers want to spend as little mental effort and system resources on security as possible. Completely novel APIs, even if secure, are unlikely to gain wide acceptance. Therefore, a key research question is how to design a practical system that (1) enables efficient distributed computations, (2) supports a familiar programming model, and (3) provides precise, rigorous privacy and security guarantees to data owners, even when the code performing the computation is untrusted. In the next few chapters we aim to answer this question for the cloud computing environment.

Mandatory access control (MAC) is a useful building block for securing distributed computations. Laminar and other MAC-based systems [44, 52, 57, 82] enforce a single access control policy for the entire system. This policy, which cannot be overridden by users, prevents information leakage via storage channels such as files, sockets, and program names.

Access control alone does not achieve end-to-end privacy in cloud computing environments, where the input data may originate from multiple sources. The output of the computation may leak sensitive information about the inputs. Since the output generally depends on all input sources, mandatory access control requires
that only someone who has access rights to all inputs should have access rights to the output; enforcing this requirement would render the output unusable for most purposes. To be useful, the output of an aggregate computation must be “declassified,” but only when it is safe to do so, i.e., when it does not reveal too much information about any single input. Existing access control mechanisms simply delegate this declassification decision to the implementor. In the case of untrusted code, there is no guarantee that the output of the computation does not reveal sensitive information about the inputs.

To address these challenges, we propose Airavat\(^1\), a system for distributed computations which provides end-to-end confidentiality, integrity, and privacy guarantees using a combination of mandatory access control and differential privacy. Airavat is based on the popular MapReduce framework, thus its interface and programming model are already familiar to developers. Differential privacy is a new methodology for ensuring that the output of aggregate computations does not violate the privacy of individual inputs [27]. It provides a mathematically rigorous basis for declassifying data in a mandatory access control system. Differential privacy mechanisms add some random noise to the output of a computation, usually with only a minor impact on the computation’s accuracy.

To prevent information leaks through system resources, Airavat runs on SELinux [57] and adds SELinux-like mandatory access control to the MapReduce distributed file system. To prevent leaks through the output of the computation, Airavat enforces differential privacy using modifications to a Java Virtual Machine and the MapReduce framework.

This chapter presents a broad overview of Airavat. It briefly discusses the MapReduce framework, SELinux and differential privacy which is background material for understanding the next two chapters.

\(^1\)The all-powerful king elephant in Indian mythology, known as the elephant of the clouds.
4.1 System overview

Airavat enables the execution of potentially untrusted data-mining and data-analysis code on sensitive data. Its objective is to accurately compute general or aggregate features of the input dataset without leaking information about specific data items.

As a motivating scenario, consider an online retailer, BigShop, which holds a large database of customer transactions. For now, assume that all records in the database have the form \( \langle \text{customer, order, date} \rangle \), with only one record per customer. A machine learning expert, Bob, pays BigShop to mine the data for certain transaction patterns. BigShop loads the data into the Hadoop framework and Bob writes the MapReduce code to analyze it.

Such computations are commonly used for targeted advertising and customer relationship management, but we will keep the example simple for clarity and assume that Bob wants to find the total number of orders placed on a particular date \( D \). He writes a mapper that looks at each record and emits the key/value pair \( \langle K, \text{order} \rangle \) if the date on the record is \( D \). Here, \( K \) is a string constant. The reducer simply sums up the values associated with each key \( K \) and outputs the result.

The main risk for BigShop in this scenario is the fact that Bob’s code is untrusted and can therefore be unintentionally buggy or even actively malicious. Because Bob’s mapper has direct access to BigShop’s proprietary transaction records, it can store parts of these data in a file which will be later accessed by Bob, or it can send them over the network. Such a leak would put BigShop at a commercial disadvantage and may also present a serious reputational risk if individual BigShop transactions were made public without the consent of the customer.

The output of the computation may also leak information. For example, Bob’s mapper may signal the presence (or absence) of a certain customer in the input dataset by manipulating the order count for a particular day: if the record of this customer is in the dataset, the mapper outputs an order count of 1 million;
otherwise, it outputs zero. Clearly, the result of the computation in this case violates
the privacy of the customer in question.

4.1.1 Architecture of Airavat

The three main entities in our model are (1) the data provider (BigShop, in our
motivating example), (2) the computation provider (Bob, sometimes referred to as
a user making a query), and (3) the computation framework (Airavat). We aim to
prevent malicious computation providers from violating the privacy policy of the
data provider(s) by leaking information about individual data items.

Computation providers write their code in the familiar MapReduce paradigm,
while data providers specify the parameters of their privacy policies. Airavat relieves
the data providers of the need to audit computation providers’ code for privacy
compliance.

Figure 4.1 gives an overview of the Airavat architecture. Airavat consists
of modifications to the MapReduce framework, the distributed file system, and
the Java virtual machine with SELinux as the underlying operating system. Airavat uses SELinux’s mandatory access control to ensure that untrusted code does not leak information via system resources, including network connections, pipes, or other storage channels such as names of running processes. To prevent information leakage through the output of the computation, Airavat relies on a differential privacy mechanism [27].

Data providers put access control labels on their data and upload them to Airavat. Airavat ensures that the result of a computation is labeled with the union of all input labels. A data provider, \( D \), can set the \textit{declassify flag} (DF in Table 4.1) to true if he wants Airavat to remove his label from the output when it is safe to do so. If the flag is set, Airavat removes \( D \)'s label if and only if the computation is differentially private. Data providers must also create a privacy policy by setting the value of several privacy parameters (explained in Section 4.4).

The computation provider must write his code in the Airavat programming model, which is close to standard MapReduce. The \textit{sensitivity} of the function being computed determines the amount of perturbation that will be applied to the output to ensure differential privacy (§ 4.4). Therefore, in Airavat the computation provider must supply an upper bound on the sensitivity of his computation by specifying the range of possible values that his mapper code may output. Airavat then ensures that the code never outputs values outside the declared range and perturbs those within the range so as to ensure privacy (§ 5.1.1). If malicious or incorrect code tries to output a value outside its declared range, the enforcement mechanism guarantees that no privacy breach will occur, but the results of the computation may no longer be accurate.

Apart from specifying the parameters mentioned above, neither the data provider, nor the computation provider needs to understand the intricacies of differential privacy and its enforcement.
4.1.2 Trusted computing base of Airavat

Airavat trusts the cloud provider and the cloud-computing infrastructure. It assumes that SELinux correctly implements MAC and relies on the MAC features added to the MapReduce distributed file system, as well as on a modified Java Virtual Machine to enforce certain properties of the untrusted mapper code (see Section 5.1.3). Airavat includes trusted implementations of several reducer functions.

We assume that the adversary is a malicious computation provider who has full control over the mapper code supplied to Airavat. The adversary may attempt to access the input, intermediate, and output files created by this code, or to reconstruct the values of individual inputs from the result of the computation.

4.2 Limitations of Airavat

Airavat cannot confine every computation performed by untrusted code. For example, a MapReduce computation may output key/value pairs. Keys are text strings that provide a storage channel for malicious mappers. In general, Airavat cannot guarantee privacy for computations that output keys produced by untrusted mappers. In many cases, privacy can be achieved by requiring the computation provider to declare the key in advance and then using Airavat to compute the corresponding value in a differentially private way.

MapReduce computations that necessarily output keys require trusted mappers. For example, printing the top $K$ items sold in a store involves printing item names. Because a malicious mapper can use a name to encode information about individual inputs, this computation requires trusted mappers. By contrast, the number of iPods sold can be calculated using an untrusted mapper because the key (“iPod” in this case) is known prior to the answer being released. (See Section 5.1
### 4.3 MapReduce and MAC

Table 4.1 lists the components of the system contributed by the data provider(s), computation provider(s), and Airavat. The following discussion explains entries in the table in the context of MapReduce computations, mandatory access control, or differential privacy. We place a bold label in the text to indicate that the discussion is about a particular row in the table.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data provider</td>
<td>Labeled data (DB)</td>
</tr>
<tr>
<td></td>
<td>Declassify flag (DF)</td>
</tr>
<tr>
<td></td>
<td>Privacy parameters ($\epsilon, \delta$)</td>
</tr>
<tr>
<td></td>
<td>Privacy budget ($P_B$)</td>
</tr>
<tr>
<td></td>
<td>Code to determine privacy groups* (PG)</td>
</tr>
<tr>
<td>Computation provider</td>
<td>Mapper range ($M_{\text{min}}, M_{\text{max}}$)</td>
</tr>
<tr>
<td>(user making a query)</td>
<td>Independent mapper code (Map)</td>
</tr>
<tr>
<td></td>
<td>Number of outputs (N)</td>
</tr>
<tr>
<td></td>
<td>Code to determine partitions* (PC)</td>
</tr>
<tr>
<td></td>
<td>Map of partition to output* (PM)</td>
</tr>
<tr>
<td></td>
<td>Max keys output for any privacy group* (n)</td>
</tr>
<tr>
<td>Airavat</td>
<td>Trusted reducer code (Red)</td>
</tr>
<tr>
<td></td>
<td>Modified MapReduce (MMR)</td>
</tr>
<tr>
<td></td>
<td>Modified distributed file system (MDFS)</td>
</tr>
<tr>
<td></td>
<td>SELinux policy (SE)</td>
</tr>
</tbody>
</table>

Table 4.1: Parameters and components provided by different participants. Optional parameters are starred.

for details.)
signed to a mapper which reads the data, performs some computation, and outputs a list of key/value pairs. In the next phase, reducers combine the values belonging to each distinct key according to some function and write the result into an output file. The framework ensures fault-tolerant execution of mappers and reducers while scheduling them in parallel on any machine (node) in the system. In MapReduce, combiners are an optional processing stage before the reduce phase. They are a performance optimization, so for simplicity, we defer them to future work. Airavat secures the execution of untrusted mappers (Map) using a MAC OS (SE), as well as modifications to the MapReduce framework (MMR) and distributed file system (MDFS).

4.3.2 SELinux

SELinux is a commercially available operating system that enforces mandatory access control. Our current implementation uses SELinux because it is a mature system that provides sufficient functionality to enforce Airavat’s security policies.

SELinux divides subjects and objects into groups called domains or types. The domain is part of the security attribute of system resources. A domain can be thought of as a sandbox which constrains the permissions of the process. For example, the system administrator may specify that a given domain can only access files belonging to certain domains. In SELinux, one can specify rules that govern transition from one domain to another. Generally, a transition occurs by executing a program declared as the entry point for a domain.

In SELinux, users are assigned roles. A role governs the permissions granted to the user by determining which domains he can access. For example, the system administrator role (sysadm_r) has permissions to access the ifconfig.t domain and can perform operations on the network interface. In SELinux, access decisions are declared in a policy file which is customized and configured by the system admin-
istrator. The Airavat-specific SELinux policy to enforce mandatory access control and declassification (SE, DF) is described in Section 5.2.

4.4 Differential privacy

The objective of Airavat is to enable large-scale computation on data items that originate from different sources and belong to different owners. The fundamental question of what it means for a computation to preserve the privacy of its inputs has been the subject of much research (see Chapter 7).

Airavat uses the recently developed framework of differential privacy [27, 28, 29, 30] to answer this question. Intuitively, a computation on a set of inputs is differentially private if, for any possible input item, the probability that the computation produces a given output does not depend much on whether this item is included in the input dataset or not. Formally, a computation \( \mathcal{F} \) satisfies \((\epsilon, \delta)\)-differential privacy [31] (where \( \epsilon \) and \( \delta \) are privacy parameters) if, for all datasets \( D \) and \( D' \) whose only difference is a single item which is present in \( D \) but not \( D' \), and for all outputs \( S \subseteq \text{Range}(\mathcal{F}) \),

\[
\Pr[\mathcal{F}(D) \in S] \leq \exp(\epsilon) \times \Pr[\mathcal{F}(D') \in S] + \delta
\]

Another intuitive way to understand this definition is as follows. Given the output of the computation, one cannot tell if any specific data item was used as part of the input because the probability of producing this output is the same as without this item. Not being able to tell whether the item was used at all in the computation precludes learning any useful information about it from the computation’s output alone.

The computation \( \mathcal{F} \) must be randomized to achieve privacy. Probability in the above definition is taken over the randomness of \( \mathcal{F} \). Deterministic computa-
tions are made privacy-preserving by adding random noise to their outputs. The privacy parameter $\epsilon$ controls the tradeoff between the accuracy of the output and the probability that it leaks information about any individual input.

The purpose of the $\delta$ parameter is to relax the multiplicative definition of privacy for certain kinds of computation. Consider TOPWORDS, which calculates the frequency of words in a corpus and outputs the top 10 words. Let $D$ and $D'$ be two large corpora; the only difference is that $D$ contains a single instance of the word “sesquipedalophobia,” while $D'$ does not. The probability that TOPWORDS outputs “sesquipedalophobia” is very small on input $D$ and zero on input $D'$. The multiplicative bound on the ratio between these probabilities required by differential privacy cannot be achieved (since one of the probabilities is zero), but the absolute difference is very small. The purpose of $\delta$ in the definition is to allow a small absolute difference in probabilities. In many of the computations we considered, this situation does not arise and $\delta$ can be safely set to 0.

In Chapter 7, we discuss why differential privacy is the “right” concept of privacy for cloud computing. The most important feature of differential privacy is that it does not make any assumptions about the adversary. When satisfied, it holds regardless of the auxiliary or prior knowledge that the adversary may possess. Furthermore, differential privacy is composable: a composition of two differentially private computations is also differentially private although $\epsilon$ and $\delta$ may increase.

There are many mechanisms for achieving differential privacy [11, 30, 33, 60]. We will use the mechanism that adds Laplacian noise to the output of a computation $f : D \rightarrow R^k$:

$$f(x) + \text{Lap}(\Delta f/\epsilon)^k$$

where $\text{Lap}(\Delta f/\epsilon)$ is a symmetric exponential distribution with standard deviation $\sqrt{2\Delta f/\epsilon}$.

**Privacy groups.** To provide privacy guarantees which are meaningful to users, it
is sometimes important to consider input datasets that differ not just on a single record, but on a group of records (PG). For example, when searching for a string within a set of documents, each input might be a line from a document, but the privacy guarantee should apply to whole documents. Differential privacy extends to privacy groups via composability: the effect of $n$ input items on the output is at most $n$ times the effect of a single item.

### 4.4.1 Function sensitivity

A function’s sensitivity measures the maximum change in the function’s output when any single item is removed from or added to its input dataset. Intuitively, the more sensitive a function, the more information it leaks about the presence or absence of a particular input. Therefore, more sensitive functions require the addition of more random noise to their output to achieve differential privacy.

Formally, the sensitivity of a function $f : D \rightarrow R^k$ is

$$\Delta(f) = \max_{D,D'} \| f(D) - f(D') \|_1$$

for any $D, D'$ that are identical except for a single element, which is present in $D$, but not in $D'$. In this paper, we will be primarily interested in functions that produce a single output, i.e., $k = 1$.

Many common functions have low sensitivity. For example, a function that counts the number of elements satisfying a certain predicate has sensitivity 1 (because the count can change by at most 1 when any single element is removed from the dataset). The sensitivity of a function that sums up integers from a bounded range is the maximum value in that range. Malicious functions that aim to leak information about an individual input or signal its presence in the input dataset are likely to be sensitive because their output must necessarily differentiate between the datasets in which this input is present and those in which it is not present.
In general, estimating the sensitivity of arbitrary untrusted code is difficult. Therefore, we require the computation provider to furnish the range of possible outputs for his mappers and use this range to derive estimated sensitivity. Estimated sensitivity is then used to add sufficient random noise to the output and guarantee privacy regardless of what the untrusted code does. If the code is malicious and attempts to output values outside its declared range, the enforcement mechanism will choose a value within the range. The computation still guarantees privacy, but the results may no longer be accurate (§ 5.1.1).

**Sensitivity of SUM.** Consider a use of SUM that takes as input 100 integers and returns their sum. If we know in advance that the inputs are all 0 or 1, then the sensitivity of SUM is low because the result varies at most by 1 depending on the presence of any given input. Only a little noise needs to be added to the sum to achieve privacy.

In general, the sensitivity of SUM is determined by the largest possible input. In this example, if one input could be as big as 1,000 and the rest are all 0 or 1, the probability of outputting any given sum should be almost the same with or without 1,000. Even if all actual inputs are 0 or 1, a lot of noise must be added to the output of SUM in order to hide whether 1,000 was among the inputs.

Differential privacy works best for low-sensitivity computations, where the maximum influence any given input can have on the output of the computation is low.

### 4.4.2 Privacy budget

Data providers may want an absolute privacy guarantee that holds regardless of the number and nature of computations carried out on the data. Unfortunately, an absolute privacy guarantee cannot be achieved for meaningful definitions of privacy.
A fundamental result by Dinur and Nissim [26] shows that the entire dataset can be decoded with a linear (in the size of the dataset) number of queries. This is a serious, but inevitable, limitation. Existing privacy mechanisms which are not based on differential privacy either severely limit the utility of the data, or are only secure against very restricted adversaries (see [29] and Chapter 7).

The composability of differential privacy and the need to restrict the number of queries naturally give rise to the concept of a “privacy budget” (\(P_B\)) [33, 58]. Each differentially private computation with a privacy parameter of \(\epsilon\) results in subtracting \(\epsilon\) from this budget. Once the privacy budget is exhausted, results can no longer be automatically declassified. The need to pre-specify a limit on how much computation can be done over a given dataset constrains some usage scenarios. We emphasize that there are no definitions of privacy that are robust, composable, and achievable in practice without such a limit.

After the privacy budget has been exhausted, Airavat still provides useful functionality. While the output can no longer be automatically declassified without risking a privacy violation, Airavat still enforces access control restrictions on the untrusted code and associates proper access control labels with the output. In this case, outputs are no longer public and privacy protection is based solely on mandatory access control.
Chapter 5

Airavat Enforcement

Mechanisms

This chapter explains the mechanisms used by Airavat to guarantee differential privacy for computations involving untrusted mappers. It also describes how Airavat confines MapReduce computations, preventing information leaks via system resources by using mandatory access control mechanisms. We use BigShop from Chapter 4.1 as our running example.

5.1 Enforcing differential privacy

Airavat supports both trusted and untrusted mappers. Because reducers are responsible for enforcing privacy, they must be trusted. The computation provider selects a reducer from a small set included in the system.

The outputs of mappers and reducers are lists of key/value pairs. An untrusted, potentially malicious mapper may try to leak information about an individual input by encoding it in (1) the values it outputs, (2) the keys it outputs, (3) the order in which it outputs key/value pairs, or (4) relationships between output
values of different keys.

MapReduce keys are arbitrary strings. Airavat cannot determine whether a key encodes sensitive information. The mere presence of a particular key in the output may signal information about an individual input. Therefore, Airavat never outputs any keys produced by untrusted mappers. Instead, the computation provider submits a key or list of keys as part of the query and Airavat returns (noisy) values associated with these keys. As explained below, Airavat always returns a value for every key in the query, even if none of the mappers produced this key. This prevents untrusted mappers from signaling information by adding or removing keys from their output.

For example, Airavat can be used to compute the noisy answer to the query “What is the total number of iPods and pens sold today?” (see the example in Section 5.1.4) because the two keys iPod and pen are declared as part of the computation. The query “List all items and their sales” is not allowed in Airavat, unless the mapper trusted. The reason is that a malicious mapper can leak information by encoding it in item names.

Trusted Airavat reducers always sort keys prior to outputting them. Therefore, a malicious mapper cannot use key order as a channel to leak information about a particular input record.

A malicious mapper may attempt to encode information by emitting a certain combination of values associated with different keys. As explained below, trusted reducers use the declared output range of mappers to add sufficient noise to ensure differential privacy for the outputs. In particular, a combination $C$ of output values across multiple keys does not leak information about any given input record $r$ because the probability of Airavat producing $C$ is approximately the same with or without $r$ in the input dataset.
5.1.1 Range declarations and estimated sensitivity

Airavat reducers enforce differential privacy by adding exponentially distributed noise to the output of the computation. The sensitivity of the computation determines the amount of noise: the noise must be sufficient to mask the maximum influence that any single input record can have on the output (§ 4.4.1).

In the case of untrusted mappers, the function(s) they compute and their sensitivity are unknown. To help Airavat estimate sensitivity, we require the computation provider to declare the range of output values \((M_{\text{min}}, M_{\text{max}})\) that his mapper can produce. Airavat combines this range with the sensitivity of the function implemented by the trusted reducer (\textsc{Red}) into estimated sensitivity. For example, estimated sensitivity of the \textsc{Sum} reducer is \(\max(|M_{\text{max}}|, |M_{\text{min}}|)\), because any single input can change the output by at most this amount.

The declared mapper range can be greater or smaller than the true global sensitivity of the function computed by the mapper. While global sensitivity measures the output difference between any two inputs that differ in at most one element (§ 4.4.1), the mapper range captures the difference between any two inputs. That said, the computation provider may assume that all inputs for the current computation lie in a certain subset of the function’s domain, so the declared range may be lower than the global sensitivity. In our clustering case study (§ 6.4.4), such an assumption allows us to obtain accurate results even though global sensitivity of clustering is very high (on “bad” input datasets, a single point can significantly change the output of the clustering algorithms).

The random noise added by Airavat to the output of MapReduce computations is a function of the data provider’s privacy parameter \(\epsilon\) and the estimated sensitivity. For example, Airavat’s \textsc{Sum} reducer adds noise from the Laplace distribution, \(\text{Lap}(\frac{b}{\epsilon})\), where \(b = \max(|M_{\text{max}}|, |M_{\text{min}}|)\).
**Example.** In the BigShop example, Bob writes his own mapper and uses the \texttt{SUM} reducer to compute the total number of orders placed on date \(D\). Assuming that a customer can order at most 25 items on any single day, Bob declares his mapper range as \((0, 25)\). The estimated sensitivity is 25 because the presence or absence of a record can affect the order total by at most 25.

**Privacy groups.** In the BigShop example, we may want to provide privacy guarantees at the level of customers rather than records (a single customer may have multiple records). Airavat supports privacy groups (§4.4), which are collections of records that are jointly present or absent in the dataset. The data provider supplies a program (PG) that takes a record as input and emits the corresponding group identifier, \(\texttt{gid}\). Airavat attaches these identifiers to key/value pairs to track the dispersal of information from each input privacy group through intermediate keys to the output. The mapper range declared by the computation provider is interpreted at the group level. For example, suppose that each BigShop record represents a purchase, a customer can make at most 10 purchases a day, and each purchase contains at most 25 orders. If all orders of a single customer are viewed as a privacy group, then the mapper range is \((0, 250)\).

### 5.1.2 Range enforcement

To prevent malicious mappers from leaking information about inputs through their output values, Airavat associates a \textit{range enforcer} with each mapper. The range enforcer checks that the value in each key/value pair output by the mapper lies within its declared range. This check guarantees that the actual sensitivity of the computation performed by the mapper does not exceed the estimated sensitivity, which is based on the declared range. If a malicious mapper outputs a value outside the range, the enforcer replaces it with a value inside the range. In the latter case, differential privacy holds, but the computation may no longer produce accurate or
meaningful results.

Range enforcement in Airavat prioritizes privacy over accuracy. If the computation provider declares the range incorrectly, the computation remains differentially private. However, the results are not meaningful and the provider gets no feedback about the problem, because any such feedback would be an information leak. The lack of feedback may seem unsatisfying, but other systems that tightly regulate information flow make similar tradeoffs. For example, MAC systems Flume and Asbestos make pipes (used for interprocess communication) unreliable and do not give the user any feedback about their failure because such feedback would leak information [47, 79].

Providing a mapper range is simple for some computations. For example, Netflix movie ratings (§6.4.3) are always between 1 and 5. When computing the word count of a set of documents, however, estimating the mapper range is more difficult. If each document is at most $N$ words, and the document is a privacy group, then the $0 - N$ range will guarantee privacy of individual documents. Depending on the number of documents, such a large range may result in adding excessive noise. For some domains, it might not be possible to obtain a reasonable estimate of the mapper’s range. Airavat gives accurate results only when the computation provider understands the sensitivity of his computation.

In the BigShop example, the range enforcer ensures that in every $(K, V)$ pair output by the mapper, $0 \leq V \leq 25$. Suppose a malicious mapper attempts to leak information by outputting 1,000 when Alice’s record is in the input dataset and 0 otherwise. Because 1,000 is outside the declared range, the range enforcer will replace it with, say, 12.5. The difference between 0 and 12.5 is less than the estimated sensitivity. Therefore, enough noise will be added so that one cannot tell, by looking at the output, whether this output was obtained by adding noise to 12.5 or 0. The noisy output thus does not reveal whether Alice’s record was present in
the input dataset or not.

**Distributed range enforcement.** A single MapReduce operation may execute mappers on many different machines. These mappers may process input elements with the same key or privacy group. Airavat associates a range enforcer with each mapper and merges their states at the end of the map phase. After merging, Airavat ensures that the values corresponding to each key or privacy group are within the declared range (see Figure 5.1).

**Example: “noisy sum.”** Figure 5.2 illustrates differential privacy enforcement with an untrusted mapper and the \( \text{SUM} \) reducer. This “noisy sum” primitive was shown by Blum *et al.* [18] to be sufficient for privacy-preserving computation of all algorithms in the statistical query model [45], including \( k \)-Means, Naive Bayes, principal component analysis, and linear classifiers such as perceptrons (for a slightly different definition of privacy).

Each input record is its own privacy group. The computation provider supplies the implementation of the actual mapper function \texttt{Map}, which converts every
// Inputs and definitions
Data owner: DB, \(\epsilon, \delta = 0, \ P_B\),
Computation provider: Map, \(M_{\text{min}}, M_{\text{max}}, N\)
Airavat: \(\text{SUM (trusted reducer, Red)}\)
\[b = \max(|M_{\text{max}}|, |M_{\text{min}}|)\]
\[\mu = \frac{(M_{\text{max}} - M_{\text{min}})}{2}\]

// Map phase
if(\(P_B - \epsilon \times N < 0\)){
    print ‘Privacy limit exceeded’;
    TERMINATE
}
\(P_B = P_B - \epsilon \times N\)
For(Record r in DB){
    \((k_0, v_0), \ldots, (k_n, v_n) = \text{Map}(r)\)
    For(i: 1 to n){
        if(\(v_i < M_{\text{min}}\) or \(v_i > M_{\text{max}}\)) {
            \(v_i = \mu\)
        }
    }
    emit\((k_0, v_0)\ldots(k_n, v_n)\)
}

// Reduce phase
count = N
Reduce(Key k, List val){
    if(--count \leq 0) { Skip }
    V = \text{SUM}(val)
    \text{print } V + \text{Lap}\left(\frac{b}{\epsilon}\right)
}
for(i: count to 0) {
    \text{print } \text{Lap}\left(\frac{b}{\epsilon}\right)
}

Figure 5.2: Simplified pseudo-code demonstrating differential privacy enforcement.
input record into a list of key/value pairs.

5.1.3 Mapper independence

Airavat forces all invocations of a mapper in a given MapReduce computation to be independent. Only a single input record is allowed to affect the key/value pairs output by the mapper. The mapper may not store the key/value pair(s) produced from an input record and use them later, when computing the key/value pair for another record. Without this restriction, estimated sensitivity used in privacy enforcement may be lower than the actual sensitivity of the mapper, resulting in a potential privacy violation. Mappers can only create additional keys for the same input record, they cannot merge information contained in different input records. We ensure mapper independence by modifying the JVM (§ 6.3).

Each mapper is permitted by the Airavat JVM to initialize itself once by overriding the configure function, called when the mapper is instantiated. To ensure independence, during initialization the mapper may not read any files written in this MapReduce computation.

5.1.4 Managing multiple outputs

A MapReduce computation may output more than one key/value pair (e.g., Figure 5.2). The computation provider must specify the number of output keys (N) beforehand; otherwise, the number of outputs can become a channel through which a malicious mapper can leak information about the inputs. If a computation produces more (fewer) than the declared number of outputs, then Airavat removes (creates) outputs to match the declared value.

Range restrictions are enforced separately for each (privacy group, key) pair. Therefore, random noise is independently added to all values associated with the final output keys. Recall that Airavat never outputs a key produced by an untrusted
mapper. Instead, the computation provider must specify a key or list of keys as part of the query, and Airavat will return the noisy values associated with each key in the query. For such queries, $N$ can be calculated automatically.

In general, each output represents a separate release of information about the same input. Therefore, Airavat must subtract more than one $\epsilon$ from the privacy budget (see Figure 5.2). If different outputs are based on disjoint parts of the input, then smaller deductions from the privacy budget are needed (see below).

**Example.** Consider the BigShop example, where each record includes the customer’s name, a list of products, and the number of items bought for each product (e.g., [Joe, iPod, 1, pen, 10]). The privacy group is the customer, and each customer may have multiple records. Bob wants to compute the total number of iPods and pens sold. Bob must specify that he expects two outputs. If he specifies the keys for these outputs as part of the query (e.g., “iPod” and “pen”), then the keys will be printed. Otherwise, only the values will be printed. Airavat subtracts $2\epsilon$ from the privacy budget for this query.

Bob’s mapper, after reading a record, outputs the product name and the number of sold items (e.g., ⟨iPod, 1⟩, ⟨pen, 10⟩—note that more than one key/value pair is output for each record). Bob also declares the mapper range for each key, e.g., (0, 5) for the number of iPods bought and (0, 25) for the number of pens. Airavat range enforcers automatically group the values by customer name and enforce the declared range for each item count. The final reducer adds random noise to the total item counts.

**Computing on disjoint partitions of the input.** When different outputs depend on disjoint parts of the input, the MapReduce computation can be decomposed into independent, parallel computations on independent datasets, and smaller deductions from the privacy budget are sufficient to ensure differential privacy. To help
Airavat track and enforce the partitioning of the input, the computation provider must (1) supply the code (PC) that assigns input records to disjoint partitions, and (2) specify which of the declared outputs will be based on which partition (PM).

The PC code is executed as part of the initial mapper. For each key/value pair generated by a mapper, Airavat constructs records of the form \((key, value, gid, pid)\), where \(gid\) is the privacy group identifier and \(pid\) is the partition identifier.

The computation provider declares which partition produces which of the \(N\) final outputs (PM). Airavat uses PM to compute \(p\), the maximum number of final outputs that depend on any single partition. If PC and PM are not provided, Airavat sets \(p\) to equal \(N\). Airavat charges \(\epsilon \times \min(N, p)\) from the privacy budget. For example, a computation provider may partition the BigShop data into two cities Austin and Seattle which act as the partition identifiers. He then specifies that the MapReduce computation will have 8 outputs, the first five of which are calculated from the Austin partition and the next three from the Seattle partition. In this example, \(N = 8, p = 5\), and to run the computation, Airavat will subtract \(5\epsilon\) from the privacy budget. In Figure 5.2, \(\epsilon \times N\) is charged to the privacy budget because the \(N\) outputs depend on the entire input, not on disjoint partitions.

Airavat enforces the partitioning declared by the computation provider. Trusted Airavat reducers use partition identifiers to ensure that only key/value pairs that have the correct \(pid\) are combined to generate the output. Airavat uses PM for computations on disjoint partitions in the same way as it uses \(N\) for unpartitioned data. If the number of outputs for a partition is less (more) than what is specified by PM, Airavat adds (deletes) outputs.

5.1.5 Trusted reducers and reducer composition

Trusted reducers such as SUM and COUNT are executed directly on the output of the mappers. The computation provider can combine these reducers with any untrusted
mapper, and Airavat will ensure differential privacy for the reducer’s output. For example, to calculate the total number of products sold by BigShop, the mapper will be responsible for the parsing logic and manipulation of the data. COUNT is a special case of SUM where the output range is \{0, 1\}. MEAN is computed by calculating the SUM and dividing it by the COUNT.

Reducers can be composed sequentially. THRESHOLD, K-COUNT, and K-SUM reducers are most useful when applied to the output of another reducer. THRESHOLD prints the outputs whose value is more than C, where C is a parameter. K-COUNT counts the number of records, and K-SUM sums the values associated with each record. For example, to count the number of distinct words occurring in a document, one can first write a MapReduce computation to group the words and then apply K-COUNT to calculate the number of groups. The sensitivity of K-COUNT is equal to the maximum number of distinct keys that a mapper can output after processing any input record.

5.1.6 Enforcing $\delta$

Privacy guarantees associated with the THRESHOLD reducer may have non-zero $\delta$. Intuitively, $\delta$ bounds the probability that the values generated from a given record will exceed the threshold and appear in the final output. Assuming that the mapper outputs at most $n$ keys after processing a single record and the threshold value is $C$,

$$\delta \leq \frac{n}{2} \exp \left( \epsilon \cdot (1 - \frac{C}{\Delta f}) \right)$$

The proof appears in the appendix. When the computation provider uses the THRESHOLD reducer, Airavat first calculates the value of $\delta$. If it is less than the bound specified by the data provider, then computation can proceed; otherwise it is aborted.
5.1.7 Mapper composition

Multiple mappers \( \{M_1, \ldots, M_j\} \) can be chained one after another, followed by a final reducer \( R_j \). Each mapper after the initial mapper propagates the partition identifier (\( \text{pid} \)) and privacy group (\( \text{gid} \)) values from the input record to output key/value pairs. Airavat enforces the declared range for the output of the final mapper \( M_j \). Noise is added only once by the final reducer \( R_j \).

To reduce the charge to the privacy budget, the computation provider can specify the maximum number of keys \( n \) that any mapper can output after reading records from a single privacy group. If provided, Airavat will enforce that maximum. If \( n \) is not provided, Airavat sets \( n \) equal to \( N \). If a mapper generates more than \( n \) key/value pairs, Airavat will only pass \( n \) randomly selected pairs to the next mapper.

When charging the total cost of a composed computation to the privacy budget, Airavat uses \( \epsilon \times \min(N, p, n^j) \) where \( j \) is the number of composed mappers, \( p \) is the maximum number of outputs from any partition (§5.1.4), and \( N \) is the total number of output keys. If the computation provider supplies the optional arguments, then \( N > p > n^j \) results in a more economical use of the privacy budget.

**MapReduce composition not supported.** Airavat supports composition of mappers and composition of reducers, but not general composition of MapReduce computations (i.e., reducer followed by another mapper). For many reducers, the output of a MapReduce depends on the inputs in a complex way that Airavat cannot easily represent, making sensitivity calculations difficult.

In the future, we plan to investigate MapReduce composition for reducers that do not combine information associated with different keys (e.g., those corresponding to a “select” statement).
5.1.8 Choosing privacy parameters

Providers of sensitive data must supply privacy parameters $\epsilon$ and $\delta$, as well as the privacy budget $P_B$, in order for their data to be used in an Airavat computation. These parameters are part of the differential privacy model. They control the trade-off between accuracy and privacy. It is not possible to give a generic recommendation for setting their values because they are highly dependent on the type of the data, the purpose of the computation, privacy threats that the data provider is concerned about, etc.

As $\epsilon$ increases, the amount of noise added to the output decreases. Therefore, the output becomes more accurate, but there is a higher chance that it reveals the presence of a record in the input dataset. In many cases, the accuracy required determines the minimum $\epsilon$-privacy that can be guaranteed. For example, in Section 6.4.5 we classify documents in a privacy-preserving fashion. Our experiments show that to achieve 95% accuracy in classification, we need to set $\epsilon$ greater than 0.6.

Intuitively, $\delta$ bounds the probability of producing an output which can occur only as a result of a particular input (see Section 4.4). Clearly, such an output immediately reveals the presence of the input in question. In many computations—for example, statistical computations where each input datapoint is a single number—$\delta$ should be set to 0. In our AOL experiment (§6.4.2), which outputs the search queries that occur more than a threshold number of times, $\delta$ is set to a value close to the number of unique users. This value of $\delta$ bounds the probability that a single user’s privacy is breached due to the release of his search query.

The privacy budget ($P_B$) is finite. If data providers specify a single privacy budget for all computation providers, then one provider can exhaust more than its fair share. Data providers could specify privacy budgets for each computation provider to ensure fairness. Managing privacy budgets is an administrative issue.
inherent to all differential privacy mechanisms and orthogonal to the design of Aira-
vat.

5.1.9 Computing with trusted mappers

While basic differential privacy only applies to computations that produce numeric
outputs, it can be generalized to discrete domains (e.g., discrete categories or strings)
using the exponential mechanism of McSherry and Talwar [60]. In general, this
requires both mappers and reducers to be trusted, because keys are an essential
part of the system’s output.

5.2 Enforcing mandatory access control

Airavat uses SELinux to execute untrusted code in a sandbox-like environment and
to ensure that local and HDFS files are safeguarded from malicious users. While
decentralized information flow control (DIFC) [72, 79, 82] would provide far greater
flexibility for access control policies within Airavat, only prototype DIFC operating
systems exist. By contrast, SELinux is a broadly deployed, mature system.

5.2.1 SELinux policy

Airavat’s SELinux policy creates two domains, one trusted and the other untrusted.
The trusted components of Airavat, such as the MapReduce framework and DFS,
execute inside the trusted domain. These processes can read and write trusted
files and connect to the network. Untrusted components, such as the user-provided
mapper, execute in the untrusted domain and have very limited permissions.

Table 5.1 shows the different domains and how they are used. The AiravatT.t
type is a trusted domain used by the MapReduce framework and the distributed
file system. Airavat labels executables that launch the framework and file system
with the AiravatT_exec.t type so the process executes in the trusted domain. This
trusted domain reads and writes only trusted files (labeled with airavatT.rw_t). No other domain is allowed to read or write these files. For example, the distributed file system stores blocks of data in the underlying file system and labels files containing those blocks with airavatT.rw_t.

In certain cases Airavat requires the trusted domain to create configuration files that can later be read by untrusted processes for initialization. Airavat uses the airavatT.notsec_t domain to label configuration files which do not contain any secrets but whose integrity is guaranteed. Since MapReduce requires network communication for transferring data, our policy allows network access by the trusted domain.

Only privileged users may enter the trusted domain. To implement this restriction, Airavat creates a trusted SELinux user called airavat_user. Only airavat_user can execute files labeled with airavatT.exec_t and transition to the trusted domain. The system administrator maps a Linux user to airavat_user.

Table 5.1: SELinux domains defined in Airavat and their usage.
The untrusted domain, `airavatU.t`, has very few privileges. A process in the untrusted domain cannot connect to the network, nor read or write files. There are two exceptions to this rule. First, the untrusted domain can read configuration files of the type `airavatT.notsec.t`. Second, it can communicate with the trusted domain using pipes. All communication with the mapper happens via these pipes which are established by the trusted framework. A process can enter the untrusted domain by executing a file of the type `airavatU_exec.t`. In our implementation, the framework transitions to the untrusted domain by executing the JVM that runs the mapper code.

Each data provider labels its input files (DB) with a domain specific to that provider. Only the trusted `airavatT.t` domain can read files from all providers. The output of a computation is stored in a file labeled with the trusted domain `airavatT.rw.t`. Data providers may set their declassify flag if they agree to declassify the result when Airavat guarantees differential privacy. If all data providers agree to declassify, then the trusted domain label is removed from the result when differential privacy holds. If only a subset of the data providers agree to declassify, then the result is labeled by a new domain, restricted to entities that have permission from all providers who chose to retain their label. Since creating domains in SELinux is a cumbersome process, our current prototype only supports full declassification. DIFC makes this ad hoc sharing among domains easy.

5.2.2 Timing channels

A malicious mapper may leak data using timing channels. MapReduce is a batch-oriented programming style where most programs do not rely on time. The bandwidth of covert timing channels is reduced by making clocks noisy and low-resolution [43]. Airavat currently denies untrusted mappers access to the high-resolution processor cycle counter (TSC), which is accessed via Java APIs. A recent timing attack re-
quires the high-definition processor counter to create a channel with 0.2 bits per second capacity [70]. Without the TSC, the data rate drops three orders of magnitude.

We are working to eliminate all obvious time-based APIs from the Airavat JVM for untrusted mappers, including `System.currentTimeMillis`. We assume an environment like Amazon’s elastic MapReduce, where the only interface to the system is the MapReduce programming interface and untrusted mappers are the only untrusted code on the system. Untrusted mappers cannot create files, so they cannot use file metadata to measure time. Airavat eliminates the API through which programs are notified about garbage collection (GC), so untrusted code has only indirect evidence about GC through the execution of finalizers, weak, soft, and phantom references (no Java native interface calls are allowed). Channels related to GC are inherently noisy and are controlled by trusted software whose implementation can be changed if it is found to leak too much timing information.

Airavat does not block timing channels caused by infinite loops (non-termination). Such channels have low bandwidth, leaking one bit per execution. Cloud providers send their users billing information (including execution time) which may be exploited as a timing channel. Quantizing billing units (e.g., billing in multiples of $10) and aggregating billing over long time periods (e.g., monthly) greatly reduce the data rate of this channel. A computer system cannot completely close all time-based channels, but a batch-oriented system like MapReduce where mappers may not access the network can decrease the utility of timing channels for the attacker to a point where another attack vector would appear preferable.
Chapter 6

Airavat Implementation and Evaluation

The Airavat implementation includes modifications to the Hadoop MapReduce framework and Hadoop file system (HDFS), a custom JVM for running user-supplied mappers, trusted reducers, and an SELinux policy file. In our prototype, we modified 2,000 lines of code in the MapReduce framework, 3,000 lines in HDFS, and 500 lines of code in the JVM. The SELinux policy is approximately 450 lines that include the type enforcement rules and interface declarations. This chapter describes the changes to the HDFS, implementation details of the range enforcers, and JVM modifications. It also describes experiments that (1) depict the tradeoff between accuracy of the computation and privacy guarantee, and (2) shows that Airavat incurs modest overheads while providing the guarantee of differential privacy.

6.1 HDFS modifications

An HDFS cluster consists of a single NameNode server that manages the file system namespace and a number of DataNode servers that store file contents. HDFS cur-
rently supports file and directory permissions that are similar to the discretionary access control of the POSIX model. Airavat modifies HDFS to support MAC labels, by placing them in the file inode structure. Inodes are stored in the NameNode server. Any request for a file operation by a client is validated against the inode label. In the DataNodes, Airavat adds the HDFS label of the file to the block information structure.

6.2 Enforcing sensitivity

As described in Section 5.1.1, each mapper has an associated range enforcer. The range enforcer determines the group for each input record and tags the output produced by the mapper with the gid. In the degenerate case when each input belongs to a group of its own, each output by the mapper is given a unique identifier as its gid. The range enforcer also determines and tags the outputs with the partition identifier, pid. The default is to tag each record as belonging to the same partition.

During the reduce phase, each reducer fetches the sorted key/value pairs produced by the mappers. The reducer then uses the gid tag to group together the output values. Any value that falls outside the range declared by the computation provider ($M_{min}$ ... $M_{max}$) is replaced by a value inside the range. Such a substitution (if it happens) prioritizes privacy over accuracy (§ 5.1.1). The reducer also enforces that only key/value pairs with the correct pid are combined to generate the final output.

6.3 Ensuring mapper independence

To add the proper amount of noise to ensure differential privacy, the result of the mapper on each input record must not depend on any other input record (§ 5.1.3). A mapper is stateful if it writes a value to storage during an invocation and then
uses this value in a later invocation. Airavat ensures that mapper invocations are not stateful by executing them in an untrusted domain that cannot write to files or the network. The MAC OS enforces the limitation that mappers cannot write to system resources.

For memory objects, Airavat adds access checks to two types of data: *objects*, which reside on the heap, and *statics*, which reside in the global pool. Airavat modifies the Java virtual machine to enforce these checks. Our prototype uses Jikes RVM 3.0.0 [5], a Java-in-Java research virtual machine.

Airavat prevents mappers from writing static variables. This restriction is enforced dynamically by using write barriers that are inserted whenever a static is accessed. Airavat modifies the object allocator to add a word to each object header. This word points to a 64-bit number called the **invocation number** (ivn). The Airavat JVM inserts read and write barriers for all objects. Before each write, the ivn of the object is updated to the current invocation number (which is maintained by the trusted framework). Before a read, the JVM checks if the object’s ivn is less than the current invocation number. If so, then the mapper is assumed to be stateful and the JVM throws an exception. After this exception, the current map invocation is re-executed and the final output of the MapReduce operation is not differentially private and must be protected using MAC (without declassification).

Jikes RVM is not mature enough to run code as large and complex as the Hadoop framework. We therefore use Hadoop’s streaming feature to ensure that mappers run on Jikes and that most of the framework executes on Sun’s JVM. The streaming utility forks a trusted Jikes process that loads the mapper using reflection. The Jikes process then executes the map function for each input provided by the streaming utility. The streaming utility communicates with the Jikes process using pipes. This communication is secured by SELinux.
Table 6.1: Details of the benchmarks, including the grouping of data, type of reducer used, number of MapReduce phases, and the accuracy metric.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Privacy grouping</th>
<th>Reducer primitive</th>
<th>#MapReduce computations</th>
<th>Accuracy metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOL queries</td>
<td>Users</td>
<td>THRESHOLD, SUM</td>
<td>Multiple</td>
<td>% Queries released</td>
</tr>
<tr>
<td>kNN recommender</td>
<td>Individual rating</td>
<td>COUNT, SUM</td>
<td>Multiple</td>
<td>RMSE</td>
</tr>
<tr>
<td>k-Means</td>
<td>Individual points</td>
<td>COUNT, SUM</td>
<td>Multiple, till convergence</td>
<td>Intra-cluster variance</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>Individual articles</td>
<td>SUM</td>
<td>Multiple</td>
<td>Misclassification rate</td>
</tr>
</tbody>
</table>

6.4 Evaluation

This section empirically makes the case that Airavat can be used to efficiently compute a wide variety of algorithms in a privacy-preserving manner with acceptable accuracy loss. Table 6.1 provides an overview of the case studies. Our experiments show that computations in Airavat incur approximately 32% overhead compared to those running on unmodified Hadoop and Linux. In all experiments except the one with the AOL queries, the mappers are untrusted. The AOL experiment outputs keys, so we trust the mapper not to encode information in the key.

6.4.1 Airavat overheads

We ran all experiments on Amazon’s EC2 service on a cluster of 100 machines. We use the large EC2 instances, each with two cores of 1.0–1.2 GHz Opteron or Xeon, 7.5 GB memory, 850 GB hard disk, and running SELinux-enabled Fedora 8. The numbers reported are the average of 5 runs, and the variance is less than 8%. K-Means and Naive Bayes use the public implementations from Apache Mahout [1].

Figure 6.1 breaks down the execution time for each benchmark. The values are normalized to the execution time of the applications running on unmodified
Figure 6.1: Normalized execution time of benchmarks when running on Airavat, compared to execution on Hadoop. Lower is better.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>JVM overhead</th>
<th>Total overhead</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOL</td>
<td>36.3%</td>
<td>23.9%</td>
<td>228 ±3</td>
</tr>
<tr>
<td>Cov. Matrix</td>
<td>43.2%</td>
<td>19.6%</td>
<td>1080 ±6</td>
</tr>
<tr>
<td>k-Means</td>
<td>28.5%</td>
<td>29.4%</td>
<td>154 ±7</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>37.4%</td>
<td>32.3%</td>
<td>94 ±2</td>
</tr>
</tbody>
</table>

Table 6.2: Airavat performance details.

Hadoop and unmodified Linux. The graph depicts the percentage of the total time spent in different phases, such as map, sort, and reduce. The category Copy represents the phase where the output data from the mappers is copied by the reducer. Note that the copy phase generally overlaps with the map phase. The benchmarks show that Airavat slows down the computation by less than 33%.

Table 6.2 measures the performance overhead of enforcing differential privacy. The JVM instrumentation, to ensure mapper independence, adds up to 44% overhead in the map phase.
6.4.2 Queries on AOL dataset

Recently, Korolova et al. showed how to release search queries while preserving privacy [46]. They first find the frequency of each query and then output the noisy count of those that exceed a certain threshold. Intuitively, the threshold suppresses uncommon, low-frequency queries, since such queries are likely to breach privacy.

We demonstrate how Airavat can perform similar computations on the AOL dataset, while ensuring differential privacy. Airavat does not output non-numeric values if the mapper is untrusted because non-numeric values can leak information (§5.1). The outputs of this experiment are search queries (which are non-numeric) and their frequencies, so we assume that the mapper is trusted. We use SUM and THRESHOLD as reducers to generate the frequency of distinct queries and then output those that exceed the threshold. The privacy group is the user, and $M$ is the maximum number of search queries made by any single user. The mapper range is $(0, M)$. We vary $M$ in our experiments.

Our experiments use the AOL data for the first week of April 2006 (253K queries). Since we use the threshold function, Airavat needs a non-zero $\delta$ as input. We chose $\delta = 10^{-5}$ based on the number of unique users for this week, 24,861. Fixing the value of $\epsilon$ and $\delta$ also determines the minimum threshold to ensure privacy. The exact threshold value can be calculated from the formula in section 5.1.5: $C = M(1 - \frac{\ln(\frac{24}{\delta M})}{\epsilon})$.

It is possible that a single user may perform an uncommon search multiple times (e.g., if he searches for his name or address). Releasing such search queries can compromise the user’s privacy. The probability of such a release can be reduced by increasing $M$ and/or setting a low value of $\delta$. A large value of $M$ implies that the release threshold $C$ is also large, thus reducing the chance that an uncommon query will be released.

In our experiments, we show the effect of different parameters on the number
of queries that get published. First, we vary $M$, the maximum number of search queries that belong to any one user. Figure 6.2 shows that as we increase the value of $M$, the threshold value also increases, resulting in a smaller number of distinct queries being released. Second, we vary the privacy parameter $\epsilon$. As we increase $\epsilon$, i.e., decrease the privacy restrictions, more queries can be released. Note that fewer than 1% of total unique queries (109K) are released. The reason is that most queries are issued very few times and hence cannot be released without jeopardizing the privacy of users who issued them.

### 6.4.3 Covariance matrices

Covariance matrices find use in many machine-learning computations. For example, McSherry and Mironov recently showed how to build a recommender system that preserves individual privacy [59]. The main idea is to construct a covariance matrix in a privacy-preserving fashion and then use a recommender algorithm such as k-nearest neighbor (kNN) on the matrix.
We picked 1,000 movies from the Netflix prize dataset and generated a covariance matrix using Airavat. The computation protects the privacy of any individual Netflix user. We cannot calculate the complete matrix in one computation using the Airavat primitives. Instead, we fill the matrix cell by individual cell. The disadvantage of this approach is that the privacy budget is expended very quickly. For example, if the matrix has $M^2$ cells, then we subtract $\epsilon M^2$ from the privacy budget (equivalently, we achieve $\epsilon M^2$-differential privacy).

Because each movie rating is between 1 and 5 and an entry of the covariance matrix is a product of two such ratings, the mapper range is $(0, 25)$. Figure 6.3 plots the root mean squared error (RMSE) of the kNN algorithm when executed on the covariance matrix generated by Airavat. The x-axis corresponds to the privacy guarantee for the complete covariance matrix. Our results show that with the guarantee of 5-differential privacy, the RMSE of kNN is approximately 0.97. For comparison, Netflix’s own algorithm, called Cinematch, has a RMSE of 0.95 when applied on the complete Netflix dataset.
6.4.4 Clustering Algorithm: k-Means

The k-Means algorithm clusters input vectors into $k$ partitions. The partitioning aims to minimize intra-cluster variances. We use Lloyd’s iterative heuristic to compute k-Means. The algorithm proceeds in two steps [18]. In the first step, the cardinality of each cluster is calculated. In the second step, all points in the new cluster are added up and then divided by the cardinality derived in the previous step, producing new cluster centers. The input dataset consists of 600 examples of control charts [7]. Control charts are used to assess whether a process is functioning properly. Machine learning techniques are often applied to such charts to detect anomaly patterns.

Figure 6.4 plots the accuracy of the k-Means algorithm as we change the privacy parameter $\epsilon$. We assume that each point belongs to a different user whose privacy must be guaranteed. The mapper range of the computation that calculates the cluster size is $(0,1)$. The mapper range for calculating the actual cluster centers is bounded by the maximum value of any coordinate over all points, which is 36 for the current dataset. We measure the accuracy of the algorithm by computing the intra-cluster variance. With $\epsilon > 0.5$, the accuracy of the clustering algorithm exceeds 90%.

6.4.5 Classification algorithm: Naive Bayes

Naive Bayes is a simple probabilistic classifier that applies the Bayes Theorem with assumptions of strong independence. During the training phase, the algorithm is given a set of feature vectors and the class labels to which they belong. The algorithm creates a model, which is then used in the classification phase to classify previously unseen vectors.

Figure 6.4 plots the accuracy against the privacy parameter $\epsilon$. We used
the 20newsgroup dataset,¹ which consists of different articles represented by words that appear in them. We train the classifier on one partition of the dataset and test it on another. The value of $\epsilon$ affects the noise which is added to the model in the training phase. We measure the accuracy of the classifier by looking at the number of misclassified articles. An article contributes at most 1,000 to a category of words, so the range for mapper outputs is $(0, 1000)$. Our results show that, for this particular dataset, we require $\epsilon > 0.6$ to achieve 95% accuracy.

---

¹http://people.csail.mit.edu/jrennie/20Newsgroups/
Chapter 7

Related Work

This chapter briefly surveys the different security models and systems that are relevant to this dissertation. It also contrasts the contributions of Laminar and Airavat to the previous systems.

7.1 Mandatory access control

Mandatory access control (MAC) assigns security attributes to system resources and uses these attributes to constrain the interaction of subjects (e.g., processes) with objects (e.g., files). In contrast to discretionary access control (e.g., UNIX permissions), MAC systems (1) check permissions on every operation and transitively enforce access restrictions (e.g., processes that access secret data cannot write non-secret files) and (2) enforce access rules specified by the system administrator at all times, without user override. MAC systems include mainstream implementations such as SELinux [57] and AppArmor [2] which appear in Linux distributions, as well as research prototypes [72, 79, 82] which implement the decentralized information flow control.
7.2 Information flow control

Information flow control (IFC) stemmed from research in multi-level security for defense projects [25]. The Bell and LaPadula [12], and Biba [13] models are some of the earliest models that describe rules of interaction between different security classes. Denning proposed information flow rules based on the lattice structure derived from security classes [23]. In the original military IFC systems [44], an administrator must allocate all labels and approve all declassification requests. Modern mandatory access control systems, like SELinux, also limit declassification and require a static collection of labels and principals. In contrast, decentralized information flow control (DIFC) systems allow individual applications to allocate labels and declassify data for their labels, providing a richer model for implementing security policies [63].

7.2.1 Language-based DIFC

Language-based DIFC systems [62, 64, 75] augment the type system to include secrecy and integrity constraints enforced by the bytecode generator. These systems label program data structures and objects at a fine granularity, but require programming an intrusive type system or an entirely new language. These language based systems trust the whole operating system and provide no guarantees against security violations on system resources, like files and sockets.

7.2.2 OS-based IFC

Asbestos [79] and HiStar [82] are new operating systems that provide DIFC properties. Flume [47] is a user-level reference monitor that provides DIFC guarantees without making extensive changes to the underlying operating system.

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

Laminar provides DIFC guarantees at the granularity of lexically scoped code blocks and data structures with modest changes to the VM. It also adds a security module to a standard operating system, as opposed to Asbestos and HiStar, which completely rewrite the OS. Most of Laminar’s OS DIFC enforcement occurs in a security module whose architecture is already present within Linux (Linux security modules [80] (LSM)). The Laminar OS does not need Flume’s endpoint abstraction to enforce security during operations on file descriptors (e.g., writes to a file or pipe), as the kernel-level reference monitor can check the information flow for each operation on a file descriptor.

Laminar adopts the label structure and the label/capability distinction derived from JIF and used by Flume. Capabilities in DIFC systems are formally defined in the next section. They are distinct from the capabilities used in capability-based operating systems. Capability-based operating systems, like EROS [74], use pointers with access control information to combine system and language mechanisms for stronger security. However, capability systems cannot enforce DIFC rules, and programs must be completely rewritten to work with the capability programming model.

Table 7.1 summarizes the taxonomy of design issues common to DIFC systems. Laminar combines the strengths of PL- and OS-based systems. Laminar
<table>
<thead>
<tr>
<th>Issue</th>
<th>PL solution [64, 75]</th>
<th>OS solution [47, 79, 82]</th>
<th>Laminar solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main subsystems modified</td>
<td>Compiler and type system</td>
<td>(1) Complete OS [79, 82] (2) User-level reference monitor and kernel module [47]</td>
<td>VM and kernel module</td>
</tr>
<tr>
<td>Trusted computing base</td>
<td>Compiler, VM, and OS</td>
<td>OS</td>
<td>VM and OS</td>
</tr>
<tr>
<td>Securing individual application data structures</td>
<td>Whole-program static analysis</td>
<td>Either not supported or inefficient because of page table mechanisms</td>
<td>Dynamic analysis and VM enforcement using read/write barriers</td>
</tr>
<tr>
<td>Securing files and OS resources</td>
<td>Not handled</td>
<td>(1) Modify entire OS or (2) User-level reference monitor with kernel module support</td>
<td>Kernel module</td>
</tr>
<tr>
<td>Termination, timing, probabilistic channels</td>
<td>Not handled</td>
<td>HiStar [82] and Flume [47] handle termination channels by suppressing termination notification</td>
<td>Not handled</td>
</tr>
<tr>
<td>Implicit information flow</td>
<td>Static analysis</td>
<td>Not applicable. Information flow tracked at granularity of thread [82] or address space [47]</td>
<td>Dynamic analysis using lexically scoped security regions</td>
</tr>
<tr>
<td>Deployment issues</td>
<td>Code must use new language or type system</td>
<td>Excludes multithreaded applications whose threads have heterogeneous security needs</td>
<td>Incrementally deployable</td>
</tr>
</tbody>
</table>

Table 7.1: Issues for DIFC systems. Laminar offers better functionality than the combination of PL and OS solutions.

handles implicit flows and provides fine-grained information flow control just like PL systems but without resorting to whole-program static analysis. Like OS-based systems, Laminar can enforce security policies on system resources. Laminar makes it easier to deploy and use information flow control systems by introducing the intuitive notion of lexically scoped code blocks called security regions.
7.2.3 Integrating language and OS-based security

Hicks et al. observe that security-typed languages can be used to ensure that OS security policies are not violated by trusted system applications, such as logrotate [42]. Their framework, called SIESTA, extends Jif to enforce SELinux [53] 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.

7.2.4 Privilege separation

Wedge [15] and Privtrans [19] allow programmers to manually partition their applications into parts with different privilege levels. The privilege separation reduces the amount of code that needs to be trusted. DIFC systems are more powerful because they can enforce more expressive policies where code can read sensitive data, without having the privilege to disclose the data. Similar to Laminar, Wedge can support different tags for memory regions. However its enforcement is at the thread level, which incurs high overheads. For example, the stthreads in Wedge are $8 \times$ slower than pthreads. Each sthread in Wedge can enforce only one policy in the address space. In contrast, security regions can enforce different policies in the same address space.

7.3 Covert 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 [78]. They prove that a correct dynamic DIFC system will over-approximate information flow, rejecting
some programs that do not contain actual information flow violations. Laminar is a dynamic DIFC system and its security regions explicitly over-approximate information flow.

DIFC systems attempt to eliminate covert channels, which may be used to leak information, but do not eliminate timing channels [48] or probabilistic channels [73].

7.4 Differential privacy

Differential privacy guarantees are somewhat similar to robust or secure statistical estimation, which provides statistical computations with low sensitivity to any single input (e.g., see [37, 39, 40]). While robust estimators do not by themselves guarantee privacy, they can serve as the basis for differentially private estimators [32].

In its current version, Airavat requires computation providers to provide an upper bound on the sensitivity of their code by declaring the range of its possible outputs in advance. An alternative is to have the enforcement system estimate local, input-specific sensitivity of the function computed by the code—either by re-running it on perturbed inputs, or by sampling from the input space [68]. Local sensitivity measures how much the output of the function varies on neighboring inputs from a subset of the function’s domain. It often requires less noise to be added to the output in order to achieve the same differential privacy guarantee.

PINQ. Privacy Integrated Queries (PINQ) is a declarative system for computing on sensitive data [58] which ensures differential privacy for the outputs of the computation. Airavat mappers are Java bytecode, with restrictions on the programming model enforced at runtime. Mapper independence is an example of a restriction enforced by the language runtime which is absent from PINQ. PINQ provides a restricted programming language with a small number of trusted, primitive data
operations in the LINQ framework. PINQ employs a request/reply model, which avoids adding noise to the intermediate results of the computation by keeping them on a trusted data server or an abstraction of a trusted data server provided by a distributed system.

Airavat’s privacy enforcement mechanisms provide end-to-end guarantees, while PINQ provides language-level guarantees. Airavat’s enforcement mechanisms include all software in the MapReduce framework, including language runtimes, the distributed file system, and the operating system. Enforcing privacy throughout the software stack allows Airavat computations to be securely distributed across multiple nodes, achieving the scalability that is the hallmark of the MapReduce framework. While the PINQ API can be supported in a similar setting (e.g., DryadLINQ), PINQ’s security would then depend on the security of Microsoft’s common language runtime (CLR), the Cosmos distributed file system, the Dryad framework, and the operating system. Securing the levels below the language layer would require the same security guarantees as provided by Airavat.

### 7.5 Alternative definitions of privacy

Differential privacy is a relative notion: it assures the owner of any individual data item that the same privacy violations, if any, will occur whether this item is included in the aggregate computation or not. Therefore, no additional privacy risk arises from participating in the computation. While this may seem like a relatively weak guarantee, stronger properties cannot be achieved without making unjustified assumptions about the adversary [27, 28]. Superficially plausible but unachievable definitions include “the adversary does not learn anything about the data that he did not know before” [21] and “the adversary’s posterior distribution of possible data values after observing the result of the computation is close to his prior distribution.”
Secure multi-party computation [36] ensures that a distributed protocol leaks no more information about the inputs than is revealed by the output of the computation. The goal is to keep the intermediate steps of the computation secret. This technique is not appropriate in our setting, where the goal is to ensure that the output itself does not leak too much information about the inputs.

While differential privacy mechanisms often employ output perturbation (adding random noise to the result of a computation), several approaches to privacy-preserving data mining add random noise to inputs instead. Privacy guarantees are usually average-case and do not imply anything about the privacy of individual inputs. For example, the algorithm of Agrawal and Srikant [9] fails to hide individual inputs [8]. In turn, Evfimievski et al. show that the definitions of [8] are too weak to provide individual privacy [34].

$k$-anonymity focuses on non-interactive releases of relational data and requires that every record in the released dataset be syntactically indistinguishable from at least $k - 1$ other records on the so-called quasi-identifying attributes, such as ZIP code and date of birth [20, 77]. $k$-anonymity is achieved by syntactic generalization and suppression of these attributes (e.g., [49]). $k$-anonymity does not provide meaningful privacy guarantees. It fundamentally assumes that the adversary’s knowledge is limited to the quasi-identifying attributes and thus fails to provide any protection against adversaries who have additional information [54, 55]. It does not hide whether a particular individual is in the dataset [66], nor the sensitive attributes associated with any individual [51, 54]. Multiple releases of the same dataset or mere knowledge of the $k$-anonymization algorithm may completely break the protection [35, 84]. Variants, such as $l$-diversity [54] and $m$-invariance [81], suffer from many of the same flaws.
7.6 Quantifying information flow

McCamant et al. show that program analysis techniques can be used to estimate how much information is leaked by a program [56]. Privacy in MapReduce computations, however, is difficult if not impossible to express as a quantitative information flow problem. The flow bound cannot be set at 0 bits because the output depends on every single input. But even a 1-bit leakage may be sufficient to reveal, for example, whether a given person’s record was present in the input dataset or not, violating privacy. By contrast, differential privacy guarantees that the information revealed by the computation cannot be specific to any given input.
Chapter 8

Conclusion

Protecting sensitive data from untrusted code is an important security challenge. Successfully solving this challenge will have the widespread effect of decreasing security breaches, increasing customer confidence in software products, and spurring the adoption of the multi-billion dollar business of cloud computing services. Existing systems either compromise on the security guarantees or are difficult to use and program, thus deterring adoption.

This dissertation is a step towards building practical and secure systems. It has shown how programmers can use the proposed new security abstractions and mechanisms to confine untrusted code while guaranteeing end-to-end security.

For desktop applications, we show how Laminar allows programmers to use decentralized information flow control to restrict the application behavior and reason about security properties. Using Laminar a programmer can structure his application into components with different access privileges. This segregation safeguards sensitive data from security breaches due to programmer bugs and malicious code. Previous DIFC systems have either used only programming language abstractions or OS abstractions. Laminar instead enforces DIFC rules for Java programs using an extended JVM and OS. By unifying programming language 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. It provides a natural programming model to retrofit powerful and auditable security policies onto existing, complex, multithreaded programs.

In the cloud computing environment, Airavat is the first system that integrates mandatory access control with differential privacy, enabling many privacy-preserving MapReduce computations without the need to audit untrusted code. Airavat has two distinct advantages. First, it gives the data providers the precise guarantee of differential privacy even when the computation performed is untrusted. Second, the computation provider need not be a security expert to use this framework. She writes the large scale computation in the familiar MapReduce paradigm and adheres to certain simple programming model restrictions. This dissertation has demonstrated the practicality of Airavat by evaluating it on a variety of large scale data mining computations.
Appendix A

Value of $\delta$ for the Threshold Reducer

Consider the threshold reducer which outputs values that are greater than a constant value $C$. In Airavat, threshold may be applied to the output of another reducer, e.g. $\text{THRESHOLD}(\text{SUM} \geq C)$. Unlike other reducers discussed in this dissertation, threshold guarantees $(\epsilon, \delta)$-differential privacy where $\delta$ may be non-zero. This appendix derives the relation of $\delta$ to the other known parameters used in the MapReduce model.

Our derivation is based on the proof techniques used by Korolova et al. [46]. In this appendix we include only the main steps. Interested readers should refer to Section 5.2 of [46] for better understanding.

Let $A$ be the algorithm that implements the threshold function. Let $\hat{D} \subseteq \text{Range}(A)$ be any arbitrary output set, and $D_1$ and $D_2$ be any two input sets that differ in one element. To guarantee differential privacy we need to prove the following:

$$Pr[A(D_1) \in \hat{D}] \leq e^{\epsilon} Pr[A(D_2) \in \hat{D}] + \delta$$  \hspace{1cm} (A.1)
In this appendix we will only consider the case where $D_1$ has an input $R$ that is not present in $D_2$. In the converse case setting $\delta$ to zero suffices. Let $\Delta f$ be the sensitivity of the aggregate computation on which the threshold reducer is applied, i.e. the maximum effect that any input $R$ can have on the value associated with any output key (before threshold is applied). Let $K = \langle k_1...k_n \rangle$ be the $n$ keys that are generated by the mapper when it processes $R$. We can partition $K$ into two sets $\langle K_s, K_t \rangle$ s.t. $K_t$ are the keys that are exclusively generated by $R$ and hence would never appear if $R$ is absent. The remaining keys are $K_s$.

Let $\hat{D}^+$ and $\hat{D}^-$ be the two subsets of the output set $\hat{D}$. $\hat{D}^+$ contains output sets that can be obtained only from $D_1$ (e.g. has some element from $K_t$), while $\hat{D}^-$ are all those sets that can be obtained from both $D_1$ and $D_2$. Let $b$ be the scale factor of the Laplacian distribution.

To prove equation A.1 we can need to calculate the following ratio:

$$\frac{Pr[A(D_1) \in \hat{D}]}{Pr[A(D_2) \in \hat{D}]} = \frac{Pr[A(D_1) \in \hat{D}^-] + Pr[A(D_1) \in \hat{D}^+]}{Pr[A(D_2) \in \hat{D}^-]} = \frac{Pr[A(D_1) \in \hat{D}^-]}{Pr[A(D_2) \in \hat{D}^-]} + \frac{Pr[A(D_1) \in \hat{D}^+]}{Pr[A(D_2) \in \hat{D}^-]}$$

(A.2)
**Ratio 1.** To calculate the first ratio let us consider the case when there is only a single key $k \in K_s$.

$$\frac{Pr[A(D_1) \in \hat{D}^-]}{Pr[A(D_2) \in \hat{D}^-]} = \frac{Pr[A(D_2 \cup k) \in \hat{D}^-]}{Pr[A(D_2) \in \hat{D}^-]}$$

$$\leq \max \left( \frac{Pr[A(D_1) \text{ releases } k]}{Pr[A(D_2) \text{ releases } k]}, \frac{Pr[A(D_1) \text{ suppresses } k]}{Pr[A(D_2) \text{ suppresses } k]} \right)$$

$$\leq \max \left( \frac{Pr[A(D_2) + \Delta f + \text{Lap}(b) > C]}{Pr[A(D_2) + \text{Lap}(b) > C]}, \frac{Pr[A(D_1) + \Delta f + \text{Lap}(b) < C]}{Pr[A(D_2) + \text{Lap}(b) < C]} \right)$$

$$\leq \max \left( \exp\left(\frac{\Delta f}{b}\right), 1 \right)$$

$$\leq \exp\left(\frac{\Delta f}{b}\right)$$

Next we consider the case when $k \in K_t$, i.e. $k$ is a key that is never output from $D_2$ since the record $R$ is missing in $D_2$.

$$\frac{Pr[A(D_1) \in \hat{D}^-]}{Pr[A(D_2) \in \hat{D}^-]} = \frac{Pr[A(D_2 \cup k) \in \hat{D}^-]}{Pr[A(D_2) \in \hat{D}^-]}$$

$$\leq \max \left( \frac{Pr[A(D_1) \text{ suppresses } k]}{Pr[A(D_2) \text{ suppresses } k]} \right)$$

$$\leq \frac{Pr[A(D_1) + \Delta f + \text{Lap}(b) < C]}{1}$$

$$\leq 1 - 0.5 \exp\left(\frac{\Delta f - C}{b}\right)$$

Using the above values we can generalize the results to the case when there is more than one key:
\[
\frac{\Pr[A(D_1) \in \hat{D}^-]}{\Pr[A(D_2) \in \hat{D}^-]} = \frac{\Pr[A(D_2 \cup K_s) \in \hat{D}^-]}{\Pr[A(D_2) \in \hat{D}^-]} \cdot \frac{\Pr[A(D_2 \cup K_s \cup K_t) \in \hat{D}^-]}{\Pr[A(D_2 \cup K_s) \in \hat{D}^-]} \\
\leq \prod_{k \in K_s} \exp\left(\frac{\Delta f}{b}\right) \cdot \prod_{k \in K_t} \max(\exp\left(\frac{\Delta f}{b}\right), 1 - 0.5 \exp\left(\frac{\Delta f - C}{b}\right)) \\
\leq \exp\left(\frac{n \cdot \Delta f}{b}\right) \quad \text{(A.3)}
\]

**Ratio 2.** Next we calculate the probability that an output from \(D_1\) belongs to \(\hat{D}^+\).

\[
\Pr[A(D_1) \in \hat{D}^+] \leq \sum_{k \in K_t} \Pr[k \text{ is released}] \\
= \sum_{k \in K_t} \Pr[\Delta f + \text{Lap}(b) > C] \\
= \sum_{k \in K_t} \Pr[\text{Lap}(b) > C - \Delta f] \\
= \frac{1}{2} \sum_{k \in K_t} \exp\left(\frac{\Delta f - C}{b}\right) \\
\leq \frac{n}{2} \exp\left(\frac{\Delta f - C}{b}\right) \quad \text{(A.4)}
\]

**Value of \(\delta\).** Substituting the values calculated in equation A.3 and A.4, and setting \(b = \frac{\Delta f}{\epsilon}\), we get:

\[
\frac{\Pr[A(D_1) \in \hat{D}]}{\Pr[A(D_2) \in \hat{D}]} = \frac{\Pr[A(D_1) \in \hat{D}^-]}{\Pr[A(D_2) \in \hat{D}^-]} + \frac{\Pr[A(D_1) \in \hat{D}^+]}{\Pr[A(D_2) \in \hat{D}^-]} \\
\leq \exp\left(\frac{n \cdot \Delta f}{b}\right) + \frac{n}{2} \exp\left(\frac{\Delta f - C}{b}\right) \cdot \frac{1}{\Pr[A(D_2) \in \hat{D}^-]} \\
\leq \exp(n \cdot \epsilon) + \frac{n}{2} \exp(\epsilon \cdot \left(1 - \frac{C}{\Delta f}\right)) \cdot \frac{1}{\Pr[A(D_2) \in \hat{D}^-]}
\]

Therefore, from the definition of differential privacy (eq. A.1), \(\delta \leq \frac{n}{2} \exp(\frac{\Delta f - C}{b})\).
Bibliography


[71] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing
hardware transactional memory in an operating system. In SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pages 87–102, New York, NY, USA, 2007. ACM.


Vita

Indrajit Roy was born in India to Kamal and Leena Roy. In 2005 he graduated from the Indian Institute of Technology Kanpur with a B.Tech. degree in computer science and engineering. The same year he enrolled in the doctoral program in the Department of Computer Science at the University of Texas at Austin.

Permanent Address: A/33 Shanti Nagar, Cantt.
Kanpur, U.P., India 208004

This dissertation was typeset with $\LaTeX\,2\varepsilon$\footnote{\LaTeX\,2\varepsilon\! is an extension of \LaTeX. \LaTeX\! is a collection of macros for \TeX. \TeX\! is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.} by the author.