BUILDING DISTRIBUTED SYSTEMS WITH INFORMATION FLOW CONTROL

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Krishnaprasad Vikram

August 2015

BUILDING DISTRIBUTED SYSTEMS WITH

INFORMATION FLOW CONTROL

Krishnaprasad Vikram, Ph.D.

Cornell University 2015

Computing technology has made recording and copying information cheap and convenient, resulting in numerous security problems: from accidental copying leading to confidentiality breaches to rapid proliferation of spam, worms and other malicious code. At the same time, distributed information systems provide value through efficient information dissemination. This thesis investigates techniques that address the challenge of building distributed systems while providing the assurance of security.

This thesis first focuses on web information systems based on the client-server communication paradigm. Servlet Information Flow (SIF) is a novel software framework for building high-assurance web applications. Security concerns are expressed as end-to-end confidentiality and integrity policies within the application code. Expressive policies allow users and application providers to protect information from one another. Together, the compiler and the runtime apply information flow analysis to prevent flow of confidential information to clients and flow of low-integrity information from clients, thereby moving the trust out of the application and into the framework. This increased assurance is obtained with modest enforcement overhead.

Where SIF enables servers to quickly and securely disseminate data to numerous clients, Swift is a new approach for building web applications that allows moving code, in addition to data, to clients. Moving code to the client makes the applications more responsive for the clients, since not every user request needs a round trip to the server. While more efficient, this mechanism

introduces security complications since the client can manipulate code running on it and influence, or gain illegal access to, sensitive server-side data. Swift allows the programmer to write the entire application code as a single sequential Java-like program with security policy annotations. The compiler automatically partitions the program between the client and server so as to respect all security policies while generating efficient client-server communication protocols.

Finally, this thesis identifies a general problem for distributed systems: *read channels*, which leak information via the pattern of data fetch requests to an untrusted host. We first discuss a type systems approach based on attaching an *access label* to each reference to a remote object. We show how the type system can prevent read channels by statically discovering their presence in a distributed program. We also discuss the expressiveness limitations of the type system approach. To address these limitations, we present a program transformation technique based on abstract interpretation to automatically eliminate read channels in any given program. We evaluate the performance of this technique on some benchmark programs.

BIOGRAPHICAL SKETCH

Very early in his life, Vikram was brought to the steel city of Jamshedpur, India where he spent much of his childhood amongst the abundant greenery and the ambitious enthusiasm of fellow denizens. Here's where he got interested in engineering, science, science fiction and then computer science as if it were a natural progression of tastes. He got his first computer at the age of fourteen. Annoyed and awed at the same time by idea of computer viruses, he decided he would play with them one day. He went to study computers at IIT Kanpur where he was fascinated by natural language processing and artificial intelligence at first but soon regarded the bottleneck issues to lie in the lower layers of the computing system itself. Inspired by subsequent projects in compilers, security and networks he found himself enrolled for graduate study at Cornell University right after, where these topics were studied in greater depth.

To my family.

ACKNOWLEDGMENTS

I would like to thank Andrew Myers, my thesis advisor, for his steady support, inspiration and encouragement through all phases of graduate school. Andrew encourages his students to do quality research, while still giving them the freedom and flexibility to choose their paths, and have fun doing so. From him, I've learned a lot (although I have more to learn) about both the value and the process of doing research, especially the focus on practical solutions to important problems. Other members in Andrew's Applied Programming Languages group — Stephen Chong, Michael Clarkson, Nate Nystrom, Lantian Zheng, Xin Qi, Jed Liu, Michael George, Xin Zheng, Aslan Askarov, Owen Arden and Danfeng Zhang — were all instrumental in maintaining a collegial atmosphere, by striving to do good research themselves. Special thanks to the senior members of the group, who were a great source of guidance and wisdom. In particular, Steve Chong was my first research mentor, and his help with the Jif compiler as well as doing research in general, was invaluable. Jed Liu and Michael George were co-hackers on the Fabric project, and I will never forget the long days (and many nights!) spent in the systems lab together. Thanks also to Saikat Guha, Bernard Wong, Alan Shieh and many other veterans of the systems lab for making it a fun place to work.

I was lucky to have had the chance to intern at IBM Watson as well as Microsoft Research Redmond. Michael Steiner, Charanjit Jutla, Pankaj Rohatgi, Suresh Chari and J R Rao at IBM ensured that we had an intellectually engaging atmosphere, and provided me with useful mentorship and guidance. Ben Livshits, Trishul Chilimbi and Sumit Gulwani at Microsoft were also accomplished researchers from whom I have learned a lot.

The professors and students in Upson Hall were always friendly and helpful. I rarely felt that I was in a foreign country, this far away from home. Thanks also to the various czars who made Cornell CS a welcoming place. Thanks to the PLDG and Systems Lunch organizers, including faculty members Andrew, Gün, Robbert, Paul, Fred, Ken, Dexter, Keshav and Radu for helping us be in touch with the latest research. Fred and Dexter were incredibly supportive and helpful as my committee members. David Bindel was kind enough to act as a proxy committee member on very short notice. As officemates, Prakash Linga and Biswanath Panda were also a source of inspiration. Ashwin Machanavajjhala and Ranveer Chandra also provided valuable mentorship.

Thanks to the folks in 32-G908 at MIT as well as Barbara Liskov who hosted us for a year. It was a uniquely rewarding experience blending in with another research group that had similar interests to ours.

Various housemates and others have also provided me with an intellectually rich experience. Ranajay and Sanjay were always ready for an engaging debate or a night out together, or often both! Yogi, Shriram, Ashutosh, Ashivni, Preetha, Amit, Vidhyashankar, Vivek and many others at Cornell also provided much needed spirited company. At MIT, Siddarth, Mayank, Aparna, Nitin, Vivek and Jagdeep made sure that I was never lonely in a completely new town. Shyam and Chaitanya visited and kept in touch while they worked on their dissertations, helping me keep up the tempo. Tudor, Daria, Thanh, Chun-Nam, Anton and Muthu were great company during early years of grad school. Thanks to Tudor, who introduced me to skiing, helping make Ithaca winters a little more bearable and to Amar, who introduced me to squash. Thanks also to Andrew Myers and Kavita Bala for organizing holiday dinners at their house, and for treating us like family.

I would also like to thank faculty members Amitabha Mukerjee, Achla Raina, Deepak Gupta and Dheeraj Sanghi at IIT Kanpur, who introduced me to the world of research even while I was an undergraduate.

My family has been very supportive of my research plans, even though this put me far away from them. My parents and my sister, especially, have given me the space and the freedom to pursue my goals in life and have sacrificed a lot for my sake and I am very grateful to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

We increasingly rely on computers for managing various aspects of our lives from personal productivity, office work, gaming and social networking to healthcare, shopping and finance. In recent years, many of these applications and systems have been distributed across multiple hosts, crossing *geographical* and *trust* boundaries. Thus, the problem of constructing *secure distributed* systems is crucial.

Constructing and maintaining these distributed systems is a complicated process; reasoning about and enforcing their security is even harder. To make matters worse, programmers as humans are error-prone. Unfortunately, current security enforcement techniques and development methods are seriously inadequate in addressing these complications. This thesis offers novel methods and techniques that are an improvement over current methods for building secure distributed systems.

In the context of computer science, security means both confidentiality and integrity, although our focus is largely on confidentiality. Currently popular techniques for reasoning about and enforcing system security — access control, cryptography, firewalls and antivirus software — suffer from inadequacies that keep the system vulnerable. For example, access control lists can prevent unauthorized processes from reading a file, but cannot prevent a process from using the data illegitimately once access has been granted to it. Similarly, encryption can secure the communication channel between two endpoints but cannot prevent misuse of information by the receiver endpoint process once it decrypts the data. Firewalls and antivirus tools prevent passage of malicious data and exe-

cution of malicious code respectively, based on heuristics and patterns of previously known malicious behavior; they offer limited protection against new attacks. This work uses the technique of *information flow control*, which provides strong, end-to-end enforcement of security, in contrast to the brittle security protection of the currently popular techniques. Most previous work has applied information flow control to enforce the security of centralized computer and operating systems. This thesis extends and adapts this work to enforce the security of distributed systems. This is elaborated in Sections 1.1 and 1.2.

Current development methods for distributed systems fall short on various fronts: (a) they are either too low-level or do not offer language integration of network messages and (b) they allow developers to fix security bugs only as vulnerabilities are exposed, thus encouraging development without concern for security. There is an urgent need to fix these issues by developing higher level languages and mathematical tools to provide security assurance *prior* to deployment, as practised in other, more mature, engineering disciplines. The state of distributed system development is arguably the same as that of program development for single machines in the 1950s before FORTRAN and COBOL compilers were built. This dissertation demonstrates how security-typed languages can form the basis for a sound development methodology for building distributed systems *securely* and *conveniently*, as detailed in Section 1.3.

1.1 Information Flow Control Overview

The idea of information flow security arose from studies of security of early operating systems for military, government and commercial organizations [8, 21].

Information flow methods were introduced to address the *confinement problem* [38] that arises in these systems. In contrast to the popular tools described earlier, information flow mechanisms track and control the flow of information through the entire system in an end-to-end [74] fashion. Information flow mechanisms ensure that the adversary is not only disallowed access to secret data, but also prevented from learning anything about that data through inference. This results in strong enforcement of security.

Information flow control is of direct practical relevance to systems that execute some untrusted code on a trusted platform. These could be browsers that download and run Java applets, smartphones and tablet PCs that download, install and run applications, regular desktop operating systems that execute untrusted programs or, as we shall see in this thesis, web servers that run third-party web applications and interact with untrusted clients. The trusted platform is called the *trusted computing base* (abbreviated to TCB) and includes the hardware, the operating system and any other infrastructure tools such as compilers, interpreters or web browsers. Programs outside the TCB often need to use sensitive information and other system resources; access to these are moderated by access control lists in typical operating systems.

Consider the example of tax preparation software that inputs sensitive tax information of the user and prepares tax charts. In a regular OS, access control would allow the program to use the sensitive information, but it would not be able to prevent it from relaying that information to, say, its vendor (short of disallowing network communication altogether). On the other hand, a system that uses information flow methods would track the *flow* of information through the execution of the program, ensuring that sensitive information does not leak

to inappropriate locations, e.g., over the network to the program vendor.

This example also shows how other security tools are inadequate. Firewalls or antivirus tools cannot detect if such programs leak data maliciously. Cryptographic techniques can ensure authenticity of the program's source and that it was not modified in transit; but they cannot make statements about the program's behavior. Not being able to reason about program behavior would be unsatisfactory if the vendor is not particularly reputed. Even if the vendor were reputed to sell quality software, information flow methods would still help the vendor maintain its reputation, by ensuring that unintentional errors by its programmers do not compromise security.

Security enforcement methods can either be external or internal. External methods are non-technical methods that include the threat of legal action, the desire to maintain a clean reputation, interest in building a long term relationship with a client, etc. This thesis focuses on internal methods, which enforce security of *computers* and not *humans*. Thus, the information flow methods in this work are applied only to computing systems. However we note that the computing systems are always used within a social context and the social context is still important in determining the security policies. The information flow methods would provide a way to translate the security requirements of the social context into something that would be enforceable. The social process itself is not adequate for security verification [29].

Information flow methods require attaching a *security label* to each program value (and also, in some languages such as Jif, each memory location). Labels on new program values are computed as a function of the labels of values used to compute the new value. When a program value influences another program

value or, conversely, a program value *depends* on another program value, the labels on the two values are used to determine whether the dependency is appropriate. An ordering relation is defined on the set of labels, and data with a certain label can depend only on data with a lower label — dependencies on data with a higher label would be disallowed. A program with an inappropriate information dependency is rejected as being insecure, and is not allowed to execute. Thus, the labels in a program are an expression of the security policy, which is enforced by tracking and disallowing inappropriate information dependencies.

For instance, going back to the tax example, let us assume that the security policy requires the salary to be kept secret from the vendor. If the program variable s stores the user's salary and the program variable x stores an integer which is sent back to the vendor, an assignment $x = s$ would constitute an inappropriate dependency. The security labels on x and s would express this policy. Dually, for integrity, if the program variable y stores an integer supplied by the vendor, the assignment $s = y$ would also violate an integrity policy that says that the vendor is not trusted to influence the user's salary (that would be the job of the employer, presumably). In general, for an expressive language, these dependencies could be subtle (e.g., implicit flows, flows through exceptions, etc.).

As mentioned earlier, the security policy, expressed as labels in the program, requires an ordering relation to be defined on the set of labels, to express what constitutes inappropriate information dependencies. Typically, the relation is a preorder relation defined on the set of labels. Security policies expressed in this manner enforce a property called *noninterference* [29]. Noninterference provides

a framework to express strict information flow security policies that that do not allow data labeled with a higher label to influence data labeled with a lower label, through all kinds of overt and covert channels.

The space of security labels is described by a *label model*, which defines the internal structure of labels and the rules for the preorder relation on the set of labels. This work uses the decentralized label model [58] (abbreviated to DLM). In this model, each label contains an integrity policy and a confidentiality policy. A policy is a set of policy components, each component *owned* by a particular *principal*. A principal is any entity that has security concerns that it wants enforced. The DLM is appropriate for the purposes of this work, since it was designed for modeling the security of systems with mutual distrusting principals, who still want to run some computation cooperatively, without the involvement of a centralized authority. The DLM is explained in further detail in Section 2.2.3.

Information flow methods are either dynamic, static or a hybrid of the two. Dynamic methods track information dependencies *during* program execution. If an inappropriate dependency is found, the program's behavior is altered to mitigate or eliminate the bad information flow. Usually the program is simply terminated, but in some settings with transactional mechanisms, a rollback is also possible. Static methods track dependencies *prior* to program execution, using program analysis techniques such as type checking, data and control flow analysis, abstract interpretation and model checking. Static methods have recently become popular due to their various advantages over dynamic methods:

- they reduce the need for run-time checking and maintenance of labels on all values and locations, thereby improving execution speed.

- they ensure that the system will not get stuck in an insecure state at runtime from which it cannot recover.
- they can control implicit flows.

Static program certification for information flow security was first proposed by Denning and Denning [23] and was later applied to mainstream programming languages via the notion of information-flow types [94, 92]. This thesis extends existing static methods for information flow: for e.g., it introduces the ability to handle dynamically changing policies with static checking, as elaborated in Section 1.3.

1.2 Distributed Information Flow Control

Most previous work on information flow tools and techniques tracks information flow within a single host [61, 94], and is concerned about the potential maliciousness of untrusted code running on a trusted execution platform. In a distributed system, the threat is stronger. An interacting host could be entirely malicious, including its execution platform. Thus, the standard technique of tracking information dependencies through program execution is not sufficient.

Interaction with an untrusted host over the network introduces many challenges for information flow security. These interactions typically involve exchanging messages according to a pre-specified network protocol. Enforcing the security of such an execution would broadly involve checking three things:

- protocol messages to the untrusted host do not leak secret information, as specified by the security policy (confidentiality requirement).

- protocol messages from the untrusted host do not influence trusted data, as per the security policy (integrity requirement).
- the untrusted host is following the protocol correctly.

Confidentiality and Covert Channels. The confidentiality requirements on the protocol are that the untrusted host is not expected to run computation that needs secret data, and that no secret data is leaked to the untrusted host during protocol execution, directly or through covert channels. The following is an example of preventing a leak through a direct channel: before sending a message, the sending host ensures that the receiving host is allowed to view the message. Preventing leaks through direct channels is relatively straightforward and can be done by tracking security labels on each variable assignment and checking that the label on the message is such that the receiver is allowed to view the message. Preventing leaks through covert channels [38] is more complicated. A covert channel[1] is an information channel that is not intended for information transfer but can still leak secret information. For instance, a program can leak one bit of secret information by choosing to either hold or not hold a lock on a shared resource. Other processes can learn this bit by requesting a hold on the lock and checking to see if the request succeeded.

[1]The literature makes a distinction between covert channels (which are intentionally and maliciously used to leak secret information) vs. side channels (which unintentionally leak secret information). The distinction is not important in this work, since it assumes that all programmer intention w.r.t. information flow is explicit in the program code, and so the two terms are used interchangeably.

Another example of a covert channel is an implicit flow [23], which leaks information via the program control context. Implicit flows can occur in a distributed system too: e.g., the decision to send a network message could depend on secret data, even though the contents of the message is public. Thus, by knowing whether the message was received, the receiver can gain secret information.

Read Channels and Security by Construction. This work identifies a new kind of covert channel that arises only in distributed systems: a *read channel*, which leaks information through the pattern of fetches of public data from an untrusted host. Techniques for the prevention of both implicit flows and read channels are presented in this thesis. Other covert channels such as timing and termination channels are not the focus of this work. Ongoing work [105, 3] orthogonal to this work is addressing them. The work presented in this thesis also ensures that no computation on the untrusted host needs secret data. This is ensured during construction of the protocol from the high level program. The approach, called *security by construction* [108], is explained in further detail in Section 1.3.

Integrity and Protocol Correctness. Integrity concerns dictate that a protocol should not allow messages from the untrusted host to influence trusted data. Checking that the untrusted host is following the protocol is challenging, but tractable, provided it is precisely known what the untrusted host is expected to compute. Knowing this, we can deduce the set of possible message responses to a message sent. Before sending a message to the untrusted host, the sending host computes the set of valid responses and maintains this state until a response arrives. If the response is not in this set, the protocol is termi-

nated. This thesis shows how a combination of program analysis and runtime mechanisms are used to compute these sets and check against them. It is possible to know what the untrusted host is expected to compute since we use the security by construction [108] approach, in which all computation is generated from a single high-level program.

Related Work. Previous work on the Jif/Split system [103] has also considered the problem of constructing secure distributed systems in the presence of mutual distrust. The main component of Jif/Split is the *splitter* that receives two inputs: the high level program in a security typed language, with confidentiality and integrity policy annotations and the host configuration which specifies the set of hosts and the trust relationships between them and the principals in the system. The splitter outputs a separate program for each host that together simulate the execution of the original high level program; this process is called program partitioning. Although the work presented in this thesis is similar in spirit to Jif/Split, there are significant differences. In Jif/Split the space of principals, labels and hosts is static. This work addresses the problem of building practical, large scale distributed systems for environments such as the web. In such environments, the space of principals and labels is dynamic, with new principals being added and trust relationships between them changing. The set of hosts is also dynamic, e.g., new clients connecting to a server. Supporting dynamism introduces novel technical challenges. For instance, changing security requirements can create new covert channels, and this thesis shows how they are handled. This work also offers a more expressive language for the high level distributed programs, e.g., label parameters to reason about information flows through complex, dynamic user interfaces. This work also addresses the read channel problem. Good performance is crucial for practicality and this

thesis also shows how the protocols that are constructed are made efficient.

1.3 Programming Language Design and Analysis for Security and Convenience

In this work, language based techniques such as program analysis and program transformation are central to both security enforcement and distributed system development. Security enforcement is achieved through information flow and static information flow is known to be more expressive than dynamic information flow. Dynamic analysis can enforce only safety properties [75] whereas static analysis can enforce a larger class of hyperproperties [15]. In this work static analysis for information flow is done in the form of security type checking and abstract interpretation.

The usual method of distributed system development involves writing code separately for each host and typically in multiple different languages. The protocols for communication between the hosts need to be carefully designed and then implemented using a suitable networking library. Security relevant code is usually *implicit* and harder to reason about. Some systems, e.g., Hibernate[7], ease the task of communicating with a database by translating object access to SQL queries. However, they do not offer tight language integration or reasoning about security. This work enables a high-level programming model for distributed system development, which also incorporates reasoning about information flow security, enabling expression of *explicit*, end-to-end, declarative confidentiality and integrity policies.

The developer writes her program in a *security-typed* language, as if it were to execute sequentially on a single host. A *security label*, consisting of a confidentiality and integrity policy is attached to types of locations and values. These policies are enforced on the corresponding location or value. Typically the labels are arranged in a lattice with the $\top$ element representing the most restrictive policy and the $\bot$ element representing the least restrictive policy. Rules corresponding to policy enforcement are added to the type system, effectively providing a static program certification for programs that successfully type check.

On the surface, it might seem like security-typed languages have a higher annotation burden than conventional languages. Although this may be true, a closer look reveals that for the same amount of effort, security-typed languages can provide greater security assurance. Declarative annotations enable convenient reasoning about security at a higher conceptual level than manual inspection of the program source can offer, leading to fewer security related bugs. In general, the security assurance gap would widen as the software base becomes larger and more complicated.

Translation from the high level program to distributed code is done automatically. Placement of code and data is constrained by security concerns and physical constraints and fine tuned by performance concerns. Automating as much of the development process as possible is not only crucial for security assurance, but also increases programmer productivity. Code that checks whether untrusted hosts are following the protocol is also generated during this translation.

Many of these language techniques are inspired by the Jif/Split system [103]. In addition, this work introduces dynamic labels and principals in the language,

which provide the required expressiveness for programming in web environments. Dynamism also allow labels on labels, which are useful if the policies themselves have to be kept secret.

This work also introduces the use of a type system to prevent read channels and an abstract interpretation to eliminate read channels in distributed systems. Recall that read channels are covert channels that leak secret information via the pattern of fetches of public data from untrusted hosts. Enabling a high-level programming language allows the programmer to ignore details of data placement and write code as if it would execute on a single host. However, to enable reasoning about read channels in the language, some annotations have to be introduced on data to suggest where it might be placed, and whether it could be placed on an untrusted host. These annotations are called 'access policies' and are associated with object fields, along with their confidentiality and integrity policies. The access policy restricts the placement of that field to a host that is trusted to enforce the access policy. The type system is extended with a rule for checking that field accesses occur only in contexts that are at most as restrictive as the access policy on that field.

Experience with programming access policies suggests that it is too tedious to program with them. Often, the programmer response to a type error is to hoist out the illegitimate field accesses to an outer context where they are legal. A copy of the value obtained from the field access is maintained until its value is needed in the secret context. This work observes that the process of hoisting and maintaining copies of object fields (called prefetching) can be automated, thereby reducing the tediousness of programming with access policies. The automation hinges on an abstract interpretation that computes the set of object

fields needed by a program segment. The challenge is to perform an abstract interpretation that is precise yet tractable. Precision is required for optimal cache usage and tractability ensures that the automation overhead is reasonable. The abstract interpretation and the original program are run together, either interleaved into a sequential program or as two parallel threads sharing only the cache.

1.4 Contributions and Roadmap

In summary, this thesis makes the following contributions. Chapters 2 and 3 show how information flow techniques can be used to build simple forms of secure distributed systems: client-server applications on the web. Although distributed computing in general is fairly mainstream and has been adopted by various commercial, government, healthcare and military organizations, web applications in particular have become especially widespread. Consumer applications that previously would run on stand-alone desktops now routinely run as web applications: e.g., multiplayer games, word processing, etc. However, compared to running desktop software, running web applications are visible to a much wider audience, and there is a greater threat that their vulnerabilities will be exploited. In short, security of web applications deserves special focus. Since many of the general technical issues of distributed system security also arise in web applications, limiting ourselves to web applications does not reduce the scope of the work.

Chapter 2 presents work that allows developers to write web applications in a Java-like language (specifically Jif 3.0 [61]) with confidentiality and integrity

annotations. The web applications make use of the services of the Servlet Information Flow (abbreviated to SIF) framework, which is built on top of the Java Servlet Framework [18]. The SIF framework provides a higher-level interface to web applications by handling many of the details of session management, HTTP request processing and HTML generation in a secure fashion. Information flow is tracked to and from clients, preventing not only SQL injection and cross-site scripting attacks but also preventing inappropriate flows of information more generally: controlling the flow of confidential information to clients and the flow of low-integrity information from clients. In addition, application-defined mechanisms for access control and authentication, and a dynamically extensible space of labels is shown to be securely integrated with language-based information flow. SIF has been tested with two web applications: email and calendar. Chapter 2 is based on joint work with Stephen Chong and Andrew Myers [12].

Chapter 3 considers more sophisticated forms of web applications. Section 3.1 shows how information flow analysis can be performed across the persistent storage tier of a web application. For this purpose, the SIF framework is integrated with Fabric [44], a platform for secure distributed computation and storage that provides language constructs for persistence and atomic transactions. The integrated system is tested with an airline ticket purchasing application. The source code for the integrated system and the airline application is available in the first Fabric release [43]. Next, Section 3.2 presents an architecture and system called Swift, which allows mobile code (in the form of JavaScript) to be migrated to the client, while maintaining the security properties of the application. Swift also provides a higher level programming model and a unified language to develop web applications. This is starkly different from current practice which involves simultaneous development in multiple languages:

HTML, CSS, Javascript, Java, SQL, etc. Section 3.2 on Swift is based on joint work with Stephen Chong, Jed Liu, Andrew Myers, Xin Qi, Lantian Zheng and Xin Zheng [10].

Finally, Chapter 4 identifies an important problem in security of distributed systems. We formalize and offer solutions for protection against *read channels*. We first discuss an approach based on type systems and show its limitations. Then we look at automatic techniques to eliminate read channels, and evaluate the performance overhead of these techniques.

CHAPTER 2

INFORMATION FLOW IN WEB APPLICATIONS

2.1 Introduction

Web applications are now used for a wide range of important activities: email, social networking, on-line shopping and auctions, financial management, and many more. They provide services to millions of users and store information about and for them. However, a web application may contain design or implementation vulnerabilities that compromise the confidentiality, integrity, or availability of information manipulated by the application, with financial, legal, or ethical implications. In 2006 [87], web applications accounted for 69% of Internet vulnerabilities. Current techniques appear inadequate to prevent vulnerabilities in web applications.

Web applications are in fact a simple instantiation of distributed systems. Consider the example of an online two player chess game. The application logic for displaying the chess board and for inputting player moves and checking that they are legitimate execute within the web browsers of the two players. The application logic for determining the outcome of a move and saving the state of the game is located entirely on a web server being shared by the two players. In the pre-Internet era, all these components of a chess application would either execute on the same computer or on computers that are physically close and in the same trust domain.

Compared to running shrink-wrapped software, running web applications are visible to a much wider audience. As a result, there is a greater threat that

their vulnerabilities will be exploited. More formally, web applications have a larger *attack surface* [52] than traditional applications. Web application security is thus a very important problem.

In general, information security vulnerabilities arise from inappropriate information dependencies, so tracking information flows within applications offers a comprehensive solution. Confidentiality can be enforced by controlling information flow from sensitive data to clients; integrity can be enforced by controlling information flow from clients to trusted information—as a side effect, protecting against common vulnerabilities like SQL injection and cross-site scripting. In fact, recent work [34, 45, 95, 35] on static analysis of PHP and Java web applications has used dependency analyses to find many vulnerabilities in existing web applications and web application libraries. Dynamic tainting can detect some improper dependencies and has also proved useful in detecting vulnerabilities [97, 39]. However, static analyses have the advantage that they can conservatively identify information flows, providing stronger security assurance [72].

Therefore, we have developed Servlet Information Flow (SIF), a novel framework for building web applications that respect explicit confidentiality and integrity information security policies. SIF web applications are written in Jif 3.0, an extended version of the Jif programming language [56, 61] (which itself extends Java with information-flow control). The enforcement mechanisms of SIF and Jif 3.0 track the flow of information within a web application, and information sent to and returned from the client. SIF reduces the trust that must be placed in web applications, in exchange for trust in the servlet framework and the Jif 3.0 compiler—a good bargain because the framework and compiler are

shared by all SIF applications.

The security policies used in SIF are both strong and expressive. Information flow is tracked through a type system that tracks all information flows, not merely explicit flows. Security enforcement is *end-to-end*, because policies are enforced on information from when it enters the web application, to when it leaves, even as information flows between different client requests. The security policies are expressive, allowing complex security requirements of multi-user systems to be enforced. Unlike prior frameworks for tracking information flow in web applications, policies can express fine-grained requirements for *both* confidentiality and integrity. Further, the interactions between confidentiality and integrity are controlled.

The end-to-end security provided by information-flow control has long been appealing, but much theoretical work on language-based information flow has not yet been successfully put into practice. We have identified limitations of existing security-typed languages for reasoning about security in a dynamic external environment, and we have extended the Jif language with new features supporting these dynamic environments, resulting in a new version of the language, Jif 3.0.

Information-flow control mechanisms work by labeling information. In previous information flow mechanisms, the space of labels is essentially static. In earlier versions of Jif, for example, labels are expressed in terms of principals, but the set of principals is fixed at compile time. This is a serious limitation for web applications, which often add new users at run time. Jif 3.0 adds the ability for applications to create their own principals, dynamically extending the space of information labels. Moreover, Jif 3.0 allows applications to implement

their own authentication and authorization mechanisms for these application-specific principals—a necessity given the diversity of authentication schemes needed by different applications. Jif 3.0 also improves Jif's ability to reason about dynamic security policies, allowing, for example, web application users to specify their own security requirements at run time and have them enforced by the information flow mechanisms. These new mechanisms create new information channels, but Jif 3.0 tracks these channels and prevents their misuse.

To explore the performance and usability of SIF, we developed two web applications with non-trivial security requirements: an email application specialized for cross-domain communication, and a multiuser shared calendar. Both applications add new principals and policies at run time, and both allow users to define their own information security policies, which are enforced by the same mechanisms used for compile-time policies.

In summary, this work makes three significant contributions:

- It shows how to use language-based information flow to construct a practical framework for high-assurance web applications, in which information flow is tracked to and from clients, and users can specify and reason about information security. To our knowledge, this is the first implemented web application framework to strongly enforce both confidentiality and integrity.
- It shows that application-defined mechanisms for access control and authentication, and a dynamically extensible space of labels, can be integrated securely with language-based information flow.
- It describes the experience using these new mechanisms to build realistic web applications.

The remainder of the chapter is structured as follows. Section 2.2 gives an overview of the Servlet Information Flow framework, including some background on the DLM and Jif. Section 2.3 introduces the new dynamic features in Jif 3.0, which enhance Jif's ability to express and enforce dynamic security requirements. Our experience with building web applications in SIF is described in Section 2.4. Section 2.5 covers related work, and Section 2.6 concludes.

2.2 Servlets with Information Flow

SIF is built using the Java Servlet framework [18], but presents a higher-level interface to web applications. Through a combination of static and dynamic mechanisms, SIF ensures that web applications use data only in accordance with specified security policies, by tracking the flow of information in the server, and information sent to and from the client. Web applications in SIF are written entirely in Jif 3.0, an extended version of the *security-typed language* [92] Jif, in which types are annotated with information flow policies. Security policies are enforced on information as it flows through the system, giving stronger security assurance than ordinary (discretionary) access control.

In designing SIF, we faced two main challenges. The first was identifying information flows in web applications, including information that flows over multiple requests. For example, a request sent to a server by a user may contain information about the user's previous request and response. The second challenge was to restrict insecure information flows while providing sufficient flexibility to implement full-fledged web applications. The resulting framework is a principled approach to designing realistic, secure web applications.

SIF is implemented in about 4040 non-comment, non-blank lines of Java code. An additional 960 lines of Jif code provide signatures for the Java classes that web applications interact with. Jif signatures provide security annotations for Java classes, and expose only a subset of the actual methods and fields to clients. SIF web applications are compiled against the Jif signatures, but linked at run time against the Java classes. Some Java Servlet framework functionality makes reasoning about information security infeasible. Using signatures and wrapper classes, SIF necessarily limits access to this functionality, but without preventing implementation of full-fledged web applications.

In this section, we first describe the threat model that SIF addresses, and the security assurances that SIF provides. We present some background about Jif and the DLM before describing the design of SIF.

2.2.1 Threat Model and Security Assurance

Threat model. We assume that web application clients are potentially malicious, and that web application implementations are benign but possibly buggy. Thus, we aim to ensure that appropriate confidentiality and integrity security policies are enforced on server-side information regardless of the actions of clients, or the mistakes of well-meaning application programmers.

Although the Jif programming language prevents the unintentional violation of information security, it provides mechanisms for explicit intentional downgrading of security policies (see Section 2.4.3). While a well-meaning programmer will be unable to accidentally misuse these mechanisms, a malicious programmer may be able to subvert them, or use certain covert channels that Jif

does not track (see Section 2.2.3).

We do not address network threats, such as denial of service attacks, or the interception and alteration of data sent over the network.

The Jif compiler and SIF are added to the trusted computing base, which already includes the servlet container, and the software stack required to run the servlet container. Note that SIF web applications are not part of the trusted computing base, whereas in standard servlet frameworks, web applications must be trusted.

Security assurance. In a typical web application, security assurance consists of convincing each party with a stake in the system that the application enforces their security requirements. Obviously users would like to have assurance that information they input will be confidential, and information they view is not corrupted. The application provider (i.e., deployer) may also have confidentiality and integrity requirements for its information. Like other recent work on improving security of web applications (e.g., [34, 42, 95, 35]), we focus on providing assurance to deployers. The difference here is that SIF enforces rich policies for information integrity and confidentiality, including policies provided by the user.

Although we focus on providing assurance to deployers, it is worth considering security assurance from a web application user's perspective. Users must be convinced that they are communicating with an application that enforces their security requirements. The security validation offered by SIF effectively partitions the security assurance problem into two parts: first, ensuring that the application respects users' security requirements, and second, ensuring the

server users communicate with is correctly running the application.

SIF addresses the first part of the assurance problem: verifying the security properties of web application code. SIF does not address the second part: convincing a remote client they are communicating with verified code. This step is important if the web application provider might be malicious. However, remote attestation methods [90, 28, 77] seem likely to be effective in solving this second problem. Attestation methods could be used to sign application code, or alternatively, to sign a verification certificate from a trusted SIF compiler that has checked the code. We leave integration of attestation mechanisms till future work.

In any case, concern about malicious application providers should not be exaggerated; users' willingness to spend money via web applications suggests they already place a modicum of trust in them. This work aims to ensure this trust is justified. At a minimum, this means application deployers can be more confident in making possibly legally binding representations to their users.

The SIF framework provides the following security assurances to deployers of web applications.

- SIF applications enforce explicit information security policies. In particular, SIF ensures that information sent to the client is permitted to be read by the client, thus ensuring that confidential information held on the server is not inadvertently released to the client. Further, information received from the client is marked as tainted by the client, helping prevent inappropriate use of low-integrity information. Thus, useful confidentiality and integrity restrictions are enforced in SIF applications.

- The information security policies of back-end systems (e.g., a database, file system, or legacy application) are also enforced, provided these systems have appropriate interfaces annotated with Jif 3.0 security policies. Thus, adding a web front-end to an existing system does not weaken the security assurance of that system, modulo the assumptions of our threat model.
- Jif ensures that security policies on information are not unintentionally weakened, or *downgraded*. However, many web applications that handle sensitive information intentionally downgrade information as part of their functionality. As discussed further in Section 2.4.3, SIF web applications must satisfy rules that enforce *selective downgrading* [59, 68] and *robustness against all attackers* [11], security conditions that provide strong information flow guarantees in the presence of downgrading.
- SIF web applications can produce only well-formed HTML. While cascading style sheets and JavaScript may be used, they cannot be dynamically generated, and must be explicitly specified in the deployment descriptor, where they can be more easily reviewed by the application deployer. The deployer thereby gains assurance that a web application does not contain malicious client-side code.

2.2.2 Non-Interference and Decentralized Label Model Overview

The specification of allowed information flows in a system is given by *noninterference policies* [29]. One group of users (usually abstracted as *high*), using a certain set of commands is noninterfering with another group of users (abstracted as *low*) if what the first group does with the commands has no effect on what the second group of users can see. As it happens, noninterference is

an idealized security requirement that is satisfied by almost no real system. In most practical systems high data needs to influence low data for correct operation. For instance, in a password checking program the password influences whether the login attempt was successful, which is visible to the attacker.

Although real systems do not fully satisfy noninterference, noninterference is useful as a *starting point* for characterizing the security of any system. In most systems, high data is allowed to influence low data in a *principled* fashion. Systems using cryptographic protocols need to downgrade information once it is encrypted [1]. Admissibility [19] is another weakening of noninterference, where the security policy states exactly which dependencies between data are allowed, including those caused by downgrading. Other weakenings of noninterference include probabilistic noninterference [93], approximate noninterference [25], quantitative bounds on information flow [22, 79, 48, 14] and information flow with computationally bounded adversaries [40].

In this work we use the *decentralized* information flow control model [58]. This model allows downgrading of the security level of information from high to low, provided certain delegation relationships hold between principals. This style of relaxing strict noninterference has been called selective declassification [68].

2.2.3 Java Information Flow (Jif)

The decentralized information flow control [58] model provides security to users and groups instead of a monolithic organization. Such a model is more suitable for our use of information flow mechanisms for distributed systems

with mutual distrust. This model is amenable to *practical* static analysis and has been integrated with the Java language [85] as a new language called Jif (Java Information Flow [56, 57]). Jif allows programmers to annotate data with security policies, as extensions of the type annotation on that data. The Jif compiler is responsible for ensuring that the program enforces the policies. We now provide a quick primer on the Jif language. The reader who is familiar with Jif may skip to the next section (Section 2.2.4). Detailed documentation and downloadable software can be found online [61].

Security policy annotations in Jif are called *labels* and are based on the decentralized label model [59]. A label describes the confidentiality and integrity constraints on the data it annotates. These constraints are expressed by users and groups, each abstractly represented by a *principal*. A principal is an entity with security concerns, and the power to observe and change certain aspects of the system. A principal $\mathtt{p}$ can delegate to another principal $\mathtt{q}$, in which case $\mathtt{q}$ is said to *act for* $\mathtt{p}$. The principal $\top$ acts for all principals and all principals act for $\bot$. The *acts-for* relation is a relation between principals and is similar to the *speaks-for* relation [37]. This language of principals and acts-for relationships can encode groups and roles. The application developer may choose which entities in the system are modeled as principals, much like a database designer modeling his data with entities and relationships between them or a programmer modeling his system with objects.

Jif labels consist of two components: a reader policy (for confidentiality) and a writer policy (for integrity). A reader policy $o \rightarrow r_1, ..., r_n$ means that principal o owns the policy and o permits any principal that can act for any r_i (or o itself) to read the data. A writer policy $o \leftarrow w_1, ..., w_n$ is owned by principal o and o has

permitted any principal that can act for any of $w_1, ..., w_n$, or o to have influenced the data.

The set of confidentiality policies is formed by closing reader policies under conjunction and disjunction, denoted $\sqcup$ and $\sqcap$ respectively. The conjunction of two confidentiality policies, $c_1 \sqcup c_2$, enforces the restrictions of both c_1 and c_2. Thus, the readers permitted by $c_1 \sqcup c_2$ is the *intersection* of readers permitted by c_1 and c_2. Similarly, the readers permitted by the disjunction $c_1 \sqcap c_2$ is the *union* of readers permitted by c_1 and c_2. Integrity policies are formed by closing writer policies under conjunction and disjunction. Dually to confidentiality, conjunction and disjunction are respectively denoted $\sqcap$ and $\sqcup$. The sets of confidentiality and integrity policies are lattices. The top and bottom elements of the confidentiality lattice are $\top \rightarrow \top$ (secret) and $\bot \rightarrow \bot$ (public) respectively. The top and bottom elements of the integrity lattice are $\bot \leftarrow \bot$ (untrusted) and $\top \leftarrow \top$ (trusted) respectively. The reason the integrity lattice might appear reversed is that this is an *information flow* lattice where policies higher in the lattice are more restrictive. Since untrusted data is more restrictive i.e. can be used in fewer places, it is higher in the lattice.

A Jif label is written as $\{c; d\}$ where c and d are confidentiality and integrity policies respectively, separated by a semicolon. Secure information flow requires that the label on a piece of information can only become more restrictive as the information flows through the system. Given labels L and L', we write $L \sqsubseteq L'$ if the label L' restricts the use of information at least as much as L does. To handle computations that combine information from different sources, the label $L_1 \sqcup L_2$ imposes the restrictions of both L_1 and L_2. Thus the set of labels also forms a lattice which is the product lattice of confidentiality and integrity

lattices.

Jif labels are attached to types of variables and expressions, making Jif a security-typed language [94, 92]. For example, a value with type `int{`$o \rightarrow r$`;` $\bot \leftarrow \bot$`}` is an integer with label $\{o \rightarrow r\ ;\ \bot \leftarrow \bot\}$: it can be read only by principals that can act for r or o, and has the lowest possible integrity (untrusted). A Jif programmer may annotate the type declarations of fields, variables, and methods with labels; use of fields, variables, and methods must comply with the label annotations. For types left unannotated, the Jif compiler either chooses default labels, or automatically infers labels, thus reducing the annotation burden on the programmer. Since labels express security requirements explicitly in terms of principals, and keep track of whose security is being enforced, they are useful for systems where principals need to co-operate despite mutual distrust.

The Jif compiler uses labels to statically check that information flows within Jif code are secure. For example, consider the following code fragment:

```
1  int {alice→bob,alice; bob←alice} y;
2  int {bob→bob} x;
3  int {alice→bob; bob→alice} z;
4  if (x == 0)
5      z = y;
```

The policy annotation on the type of `y` on line 1 means that the information in `y` is considered sensitive by `alice`, who considers that it can be released securely only to `bob` and `alice`; and further that it is considered trustworthy by `bob`, who believes that only `alice` should be allowed to affect it. The semicolon operator is overloaded to act as a separation marker between the confidentiality and integrity policies, as well as to syntactically represent the join ($\sqcup$) opera-

tor. For instance, the policy on the type of `z` on line 3 is actually alice→bob ⊔ bob→alice.

The code above causes an explicit information flow from `y` to `z`. For the code to be secure, the label on `z` must restrict the use of data in `z` as least as much as the label on `y` restricts the use of `y`. This is true if (1) for every confidentiality policy on `y`, there is one at least as restrictive on `z` (which is the case because `alice→bob` is at least as restrictive as `alice→bob,alice`) and (2) for every integrity policy on `z`, there is one at least as restrictive on `y` (which is the case because `z` has no integrity policy; the integrity of `y` (`bob←alice`) is extra.

More subtly, the code also causes an *implicit* information flow from `x` to `z`, because inspecting `z` after the code runs may impart information about `x`, even if the assignment from `y` never happens. Implicit flows are important. A failure to control this implicit flow would mean that an attacker could violate confidentiality by improperly learning about the value of `z`, or could violate integrity by changing `z` and improperly affecting the control flow of the program. Compared to purely dynamic taint tracking mechanisms, a static analysis of information flow can detect implicit flows with greater precision [23]. This precision is necessary for the applications described later in this chapter.

Applying the above rule, the implicit flow from `x` to `z` is secure if `alice→bob` is at least as restrictive as `bob→bob`. In general, this condition does not hold, because the second policy is owned by `bob`, who would not trust any enforcement of the second policy on behalf of its owner (`alice`). However, the implicit flow *would* be secure if `alice` *acts for* `bob`, meaning that `bob` trusts `alice` completely, and as a result, `alice→bob` is at least as restrictive as `bob→bob`. Acts-for relationships increase the expressive power of labels and

allow static information flow checking to work even though trust relationships change over time.

As discussed earlier, real systems do not satisfy strict information flow security. Occasionally, confidential data needs to be leaked to public variables and untrusted data needs to influence trusted variables. Jif provides `declassify` and `endorse` statements for this purpose; also known as downgrade statements. For example the result of checking the password in a login program is declassified so that it can be displayed on the login screen. Similarly, the transfer amount entered by the user in a banking application needs to be endorsed so that it can influence the balance amounts in both accounts. A Jif programmer cannot indiscriminately downgrade information. A downgrade statement requires the *authority* of all principals whose reader or writer policies are weakened or removed as a result of the downgrading. Because labels express security requirements explicitly in terms of principals, and keep track of *whose* security is being enforced, they are useful for systems where principals need to cooperate despite mutual distrust. Web applications are examples of such systems.

Although a Jif programmer may annotate a program with arbitrary labels, he does not have complete control over security. Labels must be internally consistent for the program to type-check, and moreover, the labels must be consistent with security policies from the external environment.

Jif was developed assuming a single-threaded model. Jif does not support any form of distribution or concurrency. As stated in Chapter 1, this is the current state of the art in the area of information flow control. Also, the Jif type system does not track information flow via timing or termination channels. There is ongoing work that addresses these issues [105].

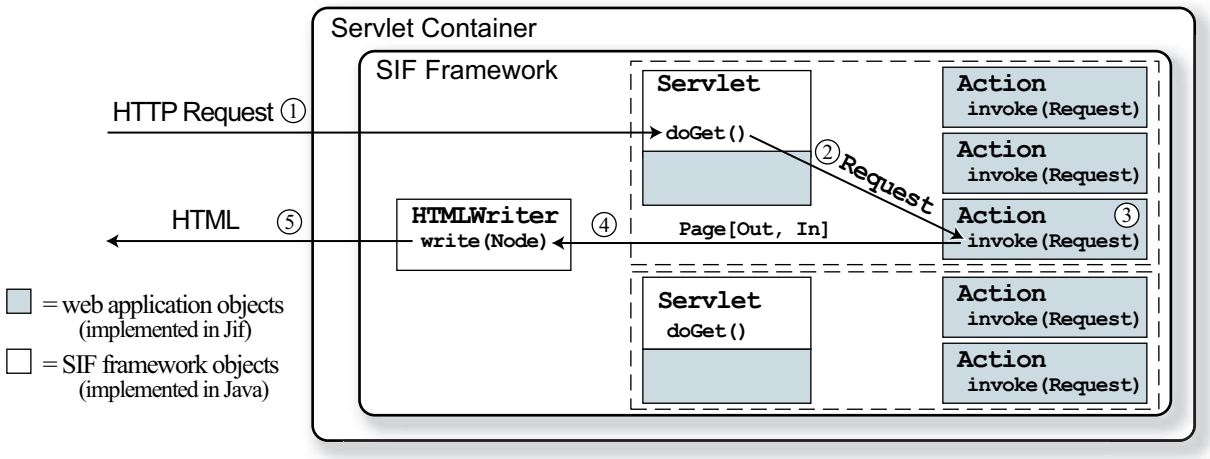

Figure 2.1: Handling a request in SIF.

2.2.4 System Design

Like the Java Servlet framework, SIF allows application code to define how client requests are handled. However, there are some structural differences that facilitate the accurate tracking of information flow. Figure 2.1 presents an overview of how SIF handles a request from a web client:

1. An HTTP request is made from a web client to a servlet;
2. The HTTP request is wrapped in a `Request` object;
3. An appropriate `Action` object of the servlet is found to handle the request, and its `invoke` method called with the `Request` object;
4. The action's `invoke` method generates a `Page` object to return for the request;
5. The `Page` object is converted into HTML, which is returned to the client.

Step 1: HTTP request from web client to servlet. Web applications must extend the class `Servlet`, which is similar to the `HttpServlet` class of the Java

```
abstract class Servlet {
    // allows servlets to specify
    // a default action
    protected Action{req} defaultAction(Request req);
    // allows servlets to create a
    // servlet-specific SessionState object
    protected SessionState createSessionState();
    public void setReturnPage{*:req.session}(
                Request{*:req.session} req,
                label out, label in,
                Node[out,in]{*in} page)
         where {*out;*in} <= {*:req.session};
}
abstract class Action {
    public abstract void
          invoke{*lbl}(label{*lbl} lbl,
                       Request{*lbl} req)
        where caller(req.session);
}
// base class of HTML elements
abstract class Node[label Out, label In] {  }
```

Figure 2.2: Jif signatures for the SIF Servlet class

Servlet framework. Figures 2.2 and 2.3 show a simplified Jif signature for the `Servlet` class, as well as other key classes of SIF. The important aspects of these signatures are explained as they arise, but because of space limitations, the syntax of Jif methods and fields are not fully explained.

Web clients establish *sessions* with the servlet; sessions are tracked by the servlet container, as in the Java Servlet specification. The SIF framework creates a *session principal* for each session, which can be thought of as corresponding to the session key shared between the client and server [37], if such a key exists. The application would typically define its own user principals, which can delegate to the session principal.

Step 2: HTTP request wrapped in a `Request` object. The class `Request` is a SIF wrapper class for an HTTP request, providing restricted access to informa-

```
final class Request {
    // principal representing the session
    // between client and server
    public final principal session;

    // reference to the Servlet
    public final Servlet servlet;

    // acquire a parameter value from the Request
    public String{*inp.L ⊔ inp ⊔ (⊥→⊥; T←session)}
              getParam(Input inp);

    // obtain a reference to
    // the SessionState object
    public SessionState getSessionState();
}
final class Input {
    private final Nonce n;
    public final label L;
}
abstract class InputNode[label Out, label In]
    extends Node[Out,In] {
    // framework statically enforces Out ⊔ In ⊑ inp.L
    private final Input{L} inp;
}
```

Figure 2.3: Jif signatures for the SIF Request and other classes

tion in the request, via the `getParam` method. The restricted interface ensures that web applications are unable to circumvent the security policies on data contained in the request, as described below.

Step 3: An `Action` is found and invoked. Web applications implement their functionality in *actions*, which are application-defined subclasses of the SIF class `Action`. A SIF servlet may have many action objects associated with it; each action object belongs to a single servlet.

Actions can be used as the targets of forms and hyperlinks. For example, the target of a form is an action object responsible for receiving and processing the data the user submits via the form. This mechanism differs from the standard Java servlet interface, which requires the application implementor to write ex-

plicit request dispatching code (the `doGet` method). However, explicit dispatch code in the application makes precise tracking of information flow difficult, as the dispatch code is executed for all requests, even though different requests may reveal different information. By avoiding dispatch code, the action mechanism permits more precise reasoning about the information revealed by client requests to the server, as discussed further in Section 2.2.5.

Action objects may be *session-specific actions*, which can only ever be used by a single session, or they may be *external actions* not specific to any given session. All action objects within a given servlet have a unique identifier. For session-specific actions, the identifier is a secure nonce, automatically generated by the framework on construction of the action. For external actions, the identifier is a (human-readable) string specified by the web application. Since external actions have fixed identifiers, they may be the target of external hyperlinks, such as a hyperlink in static HTML on a different web site.

When an HTTP request is received by a servlet, the framework finds a suitable action to handle it. Typically, the HTTP request contains a parameter value specifying the unique identifier of the appropriate action; for example, forms generated by the servlet identify the action to which the form is to be submitted. If the HTTP request does not contain an action's unique identifier, then a default action specified by the `Servlet.defaultAction` method is used to handle the request. This default is useful for handling the first request of a new session. If the HTTP request contains an invalid action identifier (e.g., the identifier of a session-specific action of an expired or invalidated session), an error page is returned, which then redirects the user to the default action.

Actions allow web applications to maintain control over application control

flow. Because session-specific actions are named with a nonce, other sessions cannot invoke them. In addition, SIF tracks the *active set* of actions for each session. An error page is returned if a request tries to invoke an action that is not active. The active set contains all external actions, and all session-specific actions that were targets of hyperlinks and forms of the last response. Thus, a client by default cannot resubmit a form by replaying its (inactive) action identifier.

Once the appropriate action object has been found, the `invoke` method is called on it with a `Request` object as an argument. The `invoke` method executes with the authority of the session principal, as shown by the `where caller(req.session)` annotation in Figure 2.2.

Web applications implement their functionality in the action's `invoke` method, as Jif 3.0 code. If required, the `invoke` method can access back-end services (e.g., a database) provided that suitable Jif interfaces exist for the services. For example, web applications can access the file system since the Jif run-time library provides a Jif interface for it, which translates file system permissions into Jif security policies.

SIF web applications can provide secure web interfaces to legacy systems, by accessing the legacy systems as back-end services. The information security of these systems is not compromised by allowing SIF applications to access them, since all accesses from Jif code must conform to the system's Jif interface.

Step 4: The `invoke` method generates a `Page` object. An object of the class `Page` is a representation of an HTML page. SIF uses the class `Node` to represent HTML elements; the class `Page`, and other HTML elements, such as

`Paragraph` and `Hyperlink`, are subclasses of `Node`. Nodes may be composed to form trees, which represent well-formed HTML code. The class `Node` is parameterized by two labels, `Out` and `In`. The `Out` label is an upper bound on the labels of information contained in the node object and its children. For example, an HTML body may contain several paragraphs, each of which contains text and hyperlinks; the `Out` parameter of each `Paragraph` node is at least as restrictive as the `Out` parameters of its child `Node`s. The `In` parameter is used to bound information that may be gained by from subsequent requests originating from this page, and is discussed further in Section 2.2.5.

The `Action.invoke` method must generate a `Page` object, and call `Servlet.setReturnPage` with that `Page` object as an argument. The signature for `Servlet.setReturnPage` ensures that the `Out` parameter of the `Page` is at most as restrictive as the label $\{\top \rightarrow \texttt{req.session;} \bot \leftarrow \bot\}$, where `req.session` is the session principal. This label is an upper bound on all labels that permit the principal `req.session` to read information, and thus the `Page` object returned for the request can contain only information that the session principal is permitted to view. This restriction is enforced statically through the type-system, and requires no runtime examination of labels by the SIF framework. Thus, assurance is gained prior to deployment that confidential information on the server is not inadvertently released.

In addition, by requiring the application to produce `Page` objects instead of arbitrary byte sequences, SIF can ensure that each input field on a page has an appropriate security policy associated with it (see Section 2.2.5), and that the web application serves only well-formed HTML that does not contain possibly malicious JavaScript.

Step 5: The `Page` is converted into HTML. SIF converts the `Page` object into HTML, which is sent to the client. The `Page` object may contain hyperlinks and forms whose targets are actions of the servlet; SIF ensures that the HTML output for these hyperlinks and forms contain parameter values specifying the appropriate actions' unique identifiers; if the user follows a hyperlink or submits a form, the appropriate action is invoked.

2.2.5 Information Flow Across Requests

The Jif compiler ensures that security policies are enforced end-to-end within a servlet, that is, from when a request is submitted until a response is returned. However, information may flow over multiple requests within the same session, for example, by being stored in session state, or by being sent to a (well-behaved) client that returns it in the next request. SIF tracks information flow over multiple requests, to ensure that appropriate security labels are enforced on data at all times.

Information flow through parameter values. SIF requires each input field on a page to have an associated security label to be enforced on the input when submitted. This label is statically required to be at least as restrictive as the label of any default value for the input field, to prevent a default value from being sent back to the server with a less restrictive policy enforced on it.

SIF ensures that the submitted value of an input field has the correct label enforced on it by preventing applications from arbitrarily accessing the HTTP request's map from parameter keys to parameter values. Instead, when an input field is created in the outgoing `Page` object, an `Input` object is associated

with it. An `Input` object is a pair $(\mathtt{n}, \mathtt{L})$, where `n` is a freshly generated nonce, and `L` is the label enforced on the input value. An application can retrieve a data value from an HTTP request only by presenting the `Input` object to the `Request.getParam(Input inp)` method, which checks the nonce, and returns the submitted value with label `inp.L` enforced on it. This "closes the loop," ensuring that data sent to the client has the correct security enforced on it when the client subsequently sends it back.

SIF does not try to protect against the user copying sensitive information from the web page, and pasting into a non-sensitive input field. That is impossible in general, and the application should define labels that prevent the user from seeing information that they are not trusted to see. By keeping track of input labels, SIF prevents web applications from laundering away security policies by sending information through the client. As discussed in Section 2.2.6, the user can also inspect the labels on inputs to see how the application will treat the information.

The `getParam` method signature also ensures that the label $\{\bot \rightarrow \bot\ ;\ \top \leftarrow \mathtt{session}\}$ is enforced on values submitted by the user. This label indicates that the value has been influenced by the session principal. Thus, SIF ensures that the integrity policy of any value obtained from the client correctly reflects that the client has influenced it; the Jif 3.0 compiler then ensures that this "tainted," or low-integrity, information cannot be incorrectly used as if it were "untainted," or high-integrity. This helps avoid vulnerabilities such as SQL injection, where low-integrity information is used in a high-integrity context.

Information flow through session state. Java servlets typically store session

state in the session map of the class `javax.servlet.http.HttpSession`. However, direct access to the session map would allow SIF applications to bypass the security policies that should be enforced on values stored in the map. Instead, SIF web applications may store state in fields of session-specific actions, or in an application-defined subclass of `SessionState`. Since fields must have labels, the Jif compiler ensures that web applications honor labels associated with values stored in the state. Web applications may override the method `Servlet.createSessionState` to create an appropriate `SessionState` object; SIF ensures at run time that this method is called exactly once per session.

Information flow through action invocation. A subtlety of the framework is that the very act of invoking an action, by following a hyperlink or submitting a form, may reveal information to the web application. For example, if a hyperlink to some action *a* is generated if and only if some secret bit is 1, then knowing that *a* is invoked reveals the value of the secret bit.

To account for this information flow, the `Action.invoke` method takes two arguments: a label `lbl`, and a reference to the `Request` object. The label `lbl` is an upper bound on the information that may be gained by knowing which action has been invoked. This means that `lbl` must be at least as restrictive as the output information for the hyperlink or form used to invoke the action. In our example, the value of `lbl` when invoking *a* would be at least as restrictive as the label of the secret bit. In general, the value for `lbl` is the value of the `In` parameter of the `Node` that contains the link to the action; the constructors for the `Node` subclasses ensure that the parameter `In` correctly bounds the information that may be gained by knowing the node was present in the `Page` returned for the request.

The method signature for `Action.invoke` ensures that the security label `lbl` is enforced on the reference to the `Request` object ("...`Request{*lbl} req`...") and that `lbl` is a lower-bound for observable side-effects of the method ("`invoke{*lbl}(...)`"), meaning that any effects of the method (such as assignments to fields) must be observable only at security levels bounded below by `lbl`. These restrictions ensure that SIF correctly tracks the information that may be gained by knowing which actions were available for the user to invoke.

2.2.6 Deployment

SIF web applications may be deployed on standard Java Servlet containers, such as Apache Tomcat, and thus may be used in a multi-tier architecture wherever Java servlets are used. The SIF and Jif run-time libraries must be available on the class path, but deployment of SIF web applications is otherwise similar to deployment of ordinary Java servlets. The deployer of a SIF web application is free to specify configuration information in the application's deployment descriptor (the `web.xml` file). For example, the deployer may require all connections to use SSL, thus protecting the confidentiality and integrity of information in transit between client and server. Additionally, there are several SIF-specific options that a deployer may specify in the deployment descriptor.

Cascading style sheets. SIF applications must use the `Node` subclasses to generate responses to requests, which allows them to generate only well-formed HTML. To allow flexibility in presentation details such as colors and font attributes, SIF permits the deployment descriptor to specify a cascading style sheet (CSS) to use in the presentation of all HTML pages generated by the ap-

plication; SIF adds this URL in the head of all generated HTML pages. `Node` objects can specify a `class` attribute, allowing style sheets to provide almost arbitrary formatting. While this allows great flexibility, care must be taken that the CSS does not contain misleading formatting. For example, inappropriate formatting might lead a user to enter sensitive information into a non-sensitive input field, such as a social security number into an address field. The deployer should review the CSS before deploying the application.

JavaScript. Dynamically generated JavaScript can provide rich user interfaces, but introduces new possibilities for security violations and covert channels. SIF does not allow web applications to send dynamic JavaScript to the client. However, as with CSSs, SIF allows deployment descriptors to specify a URL containing (static) JavaScript code to be included on all generated HTML pages. Explicit inclusion of JavaScript permits easy review by the deployer. Ideally, SIF should automatically check included JavaScript code (or perhaps an extension of JavaScript with information-flow control); we leave this to future work.

Policy visualization. User awareness of security policies is an important aspect of secure systems. Since SIF tracks the policies of information sent to the user, SIF can augment the user interface to inform the user of the security policies of data they view and supply. Provided the user trusts the interface (see Section 2.2.1), this helps prevent, for instance, a user from inappropriately copying sensitive information from the browser into an email, or from following an untrusted hyperlink.

Web applications may opt to allow SIF to automatically color-code information sent to the client, based on policy annotations. When the user presses a hotkey combination, JavaScript code recolors the page elements to reflect their

confidentiality, varying from red (highly confidential) to green (low confidentiality). Both displayed information and inputs are colored appropriately. An additional hotkey colors the page based on the integrity policies of information. A third hotkey shows a legend of colors and corresponding labels so the user can identify the precise security policy for each page element.

2.3 Language Extensions

Web applications have diverse, complicated, and dynamic security requirements. For example, web applications display a plethora of authentication schemes, including various password schemes, password recovery schemes, biometrics, and CAPTCHAs to identify human users. Web applications often enforce dynamic security policies, such as allowing users to specify who may view and update their information. Moreover, the security environment of a web application is dynamic: new users are being created, users are starting and ending sessions, and authenticating themselves.

In order both to accommodate diverse, complicated, and dynamic security requirements, and to provide assurance that these requirements are met, we have produced a new version of Jif. Section 2.2.3 describes the previous version of Jif; this section presents new features that support dynamic security requirements: integration of information flow with application-defined authentication and authorization, and improved ability to reason about and compute with dynamic security labels and principals.

Care was needed in the design and implementation of these language extensions, since there is always a tension in language-based security between

```
interface Principal {
    String name();
    // does this principal delegate authority to q?
    boolean delegatesTo(principal q);
    // is this principal prepared to authorize the
    // closure c, given proof object authPrf?
    boolean isAuthorized(Object authPrf,
                         Closure[this] c);
    // methods to guide search for acts-for proofs
    ActsForProof findProofUpTo(Principal p);
    ActsForProof findProofDownTo(Principal q);
}
interface Closure[principal P] authority(P) {
    // authority of P is required to
    // invoke a Closure
    Object invoke() where caller(P);
}
```

Figure 2.4: Signatures for application-specific principals

expressiveness and security. In particular, the new dynamic security mechanisms in Jif 3.0 create new information channels, complicating static analysis of information flow. Importantly, Jif 3.0 tracks these channels to prevent their misuse.

2.3.1 Application-Specific Principals

Principals are entities with security concerns. Applications may choose which entities to model as principals. Principals in Jif are represented at run time, and thus can be used as values by programs during execution. Jif gives run-time principals the primitive type `principal`. Jif 3.0 introduces an open-ended mechanism that allows applications great flexibility in defining and implementing their own principals.

Applications may implement the Jif 3.0 interface `jif.lang.Principal`,

shown in simplified form in Figure 2.4. Any object that implements the `Principal` interface is a principal; it can be cast to the primitive type `principal`, and used just as any other principal. The `Principal` interface provides methods for principals to delegate their authority and to define authentication.

Delegation is crucial. For example, user principals must be able to delegate their authority to session principals, so that requests from users can be executed with their authority. The method call `p.delegatesTo(q)` returns `true` if and only if principal `p` delegates its authority to principal `q`. The implementation of a principal's `delegatesTo` method is the sole determiner of whether its authority is delegated. An *acts-for proof* is a sequence of principals $p_1, \ldots, p_n$, such that each p_i delegates its authority to p_{i+1}, and is thus a proof that p_n can act for p_1. Acts-for proofs are found using the methods `findProofUpTo` and `findProofDownTo` on the `Principal` interface, allowing an application to efficiently guide a proof search. Once an acts-for proof is found, it is verified using `delegatesTo`, cleanly separating proof search from proof verification.

The authority of principals is required for certain operations. For example, the authority of the principal *Alice* is required to downgrade information labeled $\{\mathit{Alice} \rightarrow \mathit{Bob}\ ;\ \top \leftarrow \top\}$ to the label $\{\mathit{Alice} \rightarrow \mathit{Bob}, \mathit{Chuck}\ ;\ \top \leftarrow \top\}$ since a policy owned by *Alice* is weakened. The authority of principals whose identity is known at compile time may be obtained by these principals approving the code that exercises their authority. However, for dynamic principals, whose identity is not known at compile time, a different mechanism is required.

We have extended Jif with a mechanism for dynamically authorizing closures.

An *authorization closure* is an implementation of the interface `jif.lang.Closure`, shown in Figure 2.4. The `Closure` interface has a single method `invoke`, and is parameterized on a principal `P`. The `invoke` method can only be called by code that possesses the authority of principal `P`, as indicated by the annotation `where caller(P)`. Code that does not have the authority of principal `P` can request the Jif run-time system to execute a closure for `P`; the run-time system will do so only if `P` authorizes the closure.

The `Principal` interface provides a method for authorizing closures, `isAuthorized`. It takes two arguments: a `Closure` object instantiated with the principal represented by the `this` object, and an application-specific proof of authentication and/or authorization. For example, the proof might be a password, a checkable proof that the closure satisfies certain safety requirements, or a collection of certificates or capabilities. The application-specific implementation of the `isAuthorized` method examines the closure and the proof object, and returns `true` if the principal grants its authority to the closure.

The `Principal` interface and authorization closures provide a flexible mechanism for web applications to implement their own authentication and authorization mechanisms. For example, in the case studies of Section 2.4, closures are used to obtain the authority of application users after they have authenticated themselves with a password. Other implementations of principals are free to choose other authentication and authorization mechanisms, such as delegating the authorization decision to a XACML service. Dynamic authorization tests introduce new information flows that are tracked using Jif's security-type system. To prevent the usurpation of a principal's authority, the Jif run-time library cannot execute a closure unless appropriately authorized.

Legacy systems may have their own abstractions for users, authentication, and authorization. Application-specific principals allow legacy-system security abstractions to be integrated with web applications. For example, when integrating with a database with access controls, database users can be represented by suitable implementations of the `Principal` interface; web applications can then execute queries under the authority of specific database users, rather than executing all queries using a distinguished web server user.

2.3.2 Dynamic labels and principals

Jif can represent labels at run time, using the primitive type `label` for run-time label values. Following work by Zheng and Myers [107], Jif 3.0's type system has been extended with more precise reasoning about run-time labels and principals. It is now possible for the label of a value (or a principal named in a label) to be located via a *final access path expression*. A final access path expression is an expression of the form $\texttt{r.f}_1\texttt{.}\ldots\texttt{.f}_n$, where `r` is either a final local variable (including final method arguments), or the expression `this`, and each $\texttt{f}_i$ is an access to a final field. For example, in Figure 2.3, the signature for the method `Request.getParam(Input inp)` indicates that the return value has the label `inp.L` enforced on it. Therefore, the Jif 3.0 compiler can determine that the label of the result of the `getParam` method is found in the object `inp`. The additional precision of Jif 3.0 is needed to capture this relationship.

This additional precision allows SIF web applications to express and enforce dynamic security requirements, such as user-specified security policies. SIF web applications can also statically control information received from the currently

authenticated user, whose identity is unknown at compile time.

The use of dynamic labels and principals introduces new information flows, because which label is enforced on information may itself reveal information. Jif 3.0's type system tracks such flows, and prevents dynamic labels and principals from introducing covert channels.

2.3.3 Caching Dynamic Tests

To allow efficient dynamic tests of label and principal relations, the Jif 3.0 runtime system caches the results of label and principal tests. Separate caches are maintained for positive and negative results of acts-for and label tests. Care must be taken that the use of caches does not introduce unsoundness. When a principal delegation is added, the negative acts-for and label caches are cleared, as the new delegation may now enable new relationships. When a principal delegation is removed, entries in the positive acts-for and label caches that depend upon that delegation are removed, as the relationship may no longer hold.

When principals add or remove delegations, they should notify the Jif 3.0 runtime system, which updates the caches appropriately. Although an incorrectly or maliciously implemented principal p may fail to notify the runtime system, lack of notification can hurt only the principal p, since p (and only p) determines to whom its authority is delegated.

2.4 Case Studies

Using SIF, we have designed and implemented two web applications. The first is a cross-domain information sharing system that permits multiple users to exchange messages. The second is a multi-user calendar application that lets users create, edit, and view events.

This section describes the key functionality of these applications, their information security requirements, and how we reflected these requirements in the implementations. Real applications must release information, reducing its confidentiality. In SIF, this is implemented by *downgrading* to a lower security label. We discuss and categorize downgrades that occur in the applications. Based on our experience, we make some observations about programming with information-flow control.

2.4.1 Application Descriptions

Cross-domain information sharing (CDIS). CDIS applications involve exchange of information between different entities with varying levels of trust between them. For example, organizational policy may require the approval of a manager to share information between members of certain departments. Many CDIS systems provide an automatic process; for example, they determine what approval is needed, and delay information delivery until approval is obtained.

We have designed and implemented a prototype CDIS system. The interface is similar to a web-based email application. The application allows users to log in and compose messages to each other. A message may require review and ap-

Figure 2.5: Screenshot of the Calendar application.

proval by other users before it is available to its recipients. The review process is driven by a set of system-wide mandatory rules: each rule specifies for a unique sender-recipient pair which users need to review and approve messages. Once all appropriate reviewers have approved a message, it appears in the recipient's inbox. Each user also has a "review inbox," for messages requiring their approval or rejection. In this prototype, all messages are held centrally on the web server; a full implementation would be integrated with an SMTP server.

	Lines	Annotated Lines	Downgrade Annotations	Functional downgrades			
				Access control	Imprecision	Application	Total
CDIS	1325	277	76	11	0	3	14
Calendar	1779	443	73	12	0	5	17
User	925	283	31	3	1	4	8

Figure 2.6: Summary of case studies.

Calendar. We have also implemented a multi-user calendar system. Authenticated users may create, edit, and view events. Events have a time, title, list of attendees, and description. Events are controlled by expressive security policies, customizable by application users. A user can edit an event only if the user acts for the creator of the event (recall that the *acts-for* relation is reflexive). A user may view the details of an event (title, attendees, and description) if the user acts for either the creator or an attendee. An event may specify a list of additional users who are permitted to view the time of the event—to view an event, a user must act for the creator, for an attendee, or for a user on this list.

A user's calendar is defined to be the set of all events for which the user is either the creator or an attendee. When a user u views another user v's calendar, u will see only the subset of events on v's calendar for which u is permitted to see the details or time. If the user is permitted to view the time, but not the details of an event, the event is shown as "Busy."

Measurements. Measurements of the applications' code are given in Figure 2.6, including non-blank non-comment lines of code, lines with label annotations, and the number of `declassify` and `endorse` annotations, which indicate intentional downgrading of information (see Section 2.4.3).

Performance tests indicate that the overhead due to the SIF framework is modest. We compared the calendar case study application to a Java servlet we

implemented with similar functionality, using the same back-end database; the Java servlet does not offer the security assurances of the SIF servlet. Tests were performed using Apache Tomcat 5.5 in Redhat Linux, kernel version 2.6.17, running on a dual-core 2.2GHz Opteron processor with 3GB of memory. As the number of concurrent sessions varies between 1 and 245, the SIF servlet exhibits at most a 29% reduction in requests processed per second, showing that SIF does not dramatically affect scalability. At peak throughput, the Java servlet processes 2010 requests per second, compared with 1503 for the SIF servlet. Of the server processing time for a request to the SIF servlet, about 17% is spent rendering the `Page` object into HTML, and about 9% is spent performing dynamic label and principal tests.

2.4.2 Implementing Security Requirements

Many of the security requirements of both applications can be expressed using Jif's security mechanisms, including dynamic principals and security labels, and thus automatically enforced by Jif and SIF's static and run-time mechanisms. Other security requirements are enforced programmatically.

Principals. Users of the applications are application-specific principals (see Section 2.3.1). We factored out much functionality from both applications relating to user management, such as selecting users and logging on and off. The sharing of code across both case studies shows that SIF permits the design and implementation of reusable components. Figure 2.6 also shows measurements of the reusable user library.

The login process works as follows: a user and password are specified on the

login screen, and if the password is correct, the authority of the user is dynamically obtained via a closure; the closure is used to delegate the user's authority to the session principal, who can then act on behalf of the now logged-in user.

In addition to user principals, the two applications define principals `CDISApp` and `CalApp`, representing the applications themselves. These model the security of sensitive information that is not owned by any one user, such as the set of application users. This information is labeled $\{p \rightarrow \top \; ; \; p \leftarrow \top\}$, where p is one of `CDISApp` or `CalApp`, and relevant portions are downgraded for use as needed. In particular, information in the database has this label. Since all information sent to and from the database (including data used in SQL queries) must have this label, the authority of the application principal (`CDISApp` or `CalApp`) is required to endorse information sent to the database and to declassify information received from it. This provides a form of access control, ensuring that only code authorized by the application principal is able to access the database. The need to explicitly endorse data used in SQL queries also helps to prevent SQL injection attacks, by making the programmer aware of exactly what information may be used in SQL queries.

Dynamic security labels. The security labels of Jif 3.0 are expressive enough to capture the case studies' information-sharing requirements. In particular, we are able to model the confidentiality and review requirements for CDIS messages by enforcing appropriate labels on the messages. For instance, suppose sender s is sending a message to recipient t. The confidentiality policy $s \rightarrow t$ would allow both s and t to read the message. However, before t is permitted to read the message, it may need to be reviewed. Suppose reviewers $r_1, r_2, ..., r_n$ must review all messages sent from s to t. When s composes the message,

it initially has the following confidentiality policy: $(s \to t, r_1, \ldots, r_n) \sqcup (r_1 \to r_1, \ldots, r_n) \sqcup \ldots \sqcup (r_n \to r_1, \ldots, r_n)$. In this policy, s permits t and all reviewers to read the message, and each reviewer permits all other reviewers to read the message. This label allows the message to be read by each reviewer, but prevents t from reading it. As each reviewer reviews and approves the message, their authority is used to remove their reader policy from the confidentiality policy using `declassify` annotations. Eventually the message is declassified to the policy $s \to t, r_1, \ldots, r_n$, which permits t to read it.

The calendar application also enforces user-defined security requirements by labeling information with appropriate dynamic labels. Event details have the confidentiality policy $c \to a_1, \ldots, a_n$ enforced on them, where c is the creator of the event and $a_1, \ldots, a_n$ are the event attendees. The time of an event has confidentiality policy $c \to a_1, \ldots, a_n \sqcap c \to t_1, \ldots, t_m$, where $t_1, \ldots, t_m$ are the users explicitly given permission by c to view the event time. Event labels ensure that times and details flow only to users permitted to see them; run-time label tests are used to determine which events a user can see.

2.4.3 Downgrading

Jif prevents the unintentional downgrading of information. However, most applications that handle sensitive information, including the case study applications, need to downgrade information as part of their functionality. Jif provides a mechanism for deliberate downgrading of information: *selective declassification* [59, 68] is a form of access control, requiring the authorization of the owners of all policies weakened or removed by a downgrade. Authorization can be

acquired statically if the owner of a policy is known at compile time; or authorization can be acquired at run time through a closure (see Section 2.3).

Jif 3.0 programs must also satisfy typing rules to enforce *robust declassification* [101, 60, 11]. In the context of Jif, robustness ensures that no principal p (including attackers) is able to influence either *what* information is released to p (a *laundering attack*), or *whether* to release information to p. For a web application, robustness implies that users are unable to cause the incorrect release of information. Selective declassification and robust declassification are orthogonal, providing different guarantees regarding the downgrading of information.

In Jif programs, downgrading is marked by explicit program annotations. A `declassify` annotation allows confidentiality to be downgraded, whereas an `endorse` annotation downgrades integrity.

Downgrading annotations are typically clustered together in code, with several annotations needed to accomplish a single "functional downgrade." For example, declassifying a data structure requires declassification of each field of the structure [4]. The two applications had a combined total of 39 functional downgrades, with an average of 4.6 annotations per functional downgrade.

Figure 2.6 shows a more detailed breakdown of the use of downgrading in each case study. We found that downgrading could be divided into three broad categories: access control, imprecision, and application requirements.

The first category is downgrades associated with discretionary access control. Discretionary access control is used as a mechanism to mediate information release between different application components; any information release requires explicit downgrading. For example, in the calendar application, the set

of all events has the label {CalApp → ⊤ ; CalApp ← ⊤}; thus, downgrading is required both to extract events to display to the user, and to update events edited by the user; the authority of CalApp is required for these downgrades, and thus the downgrades serve as a form of discretionary access control to the event set. The choice of the label {CalApp → ⊤ ; CalApp ← ⊤} for the event set necessitates these downgrades; using other labels may result in fewer downgrades, but without the benefits of this discretionary access control.

Imprecision is another reason for downgrading: sometimes the programmer can reason more precisely than the compiler about security labels and information flows. For example, suppose a method is always called with a non-null argument: Jif 3.0 has no ability to express this precondition, and conservatively assumes that accessing the argument may result in a NullPointerException. Since the exception may reveal information, a spurious information flow is introduced, which may require explicit downgrading later. Few downgrades fall into this category, giving confidence that Jif 3.0 is sufficiently expressive. Some imprecision could be removed entirely by extending the compiler to accept and reason about additional annotations, as in JML [41].

Security requirements of the application provide the third category of downgrade reasons. These downgrades are inherent in the application, and cannot and should not be avoided. For example, in the calendar application, when users are added to the list of event attendees, more users are able to see the details of the event, an information release that requires explicit downgrading.

2.4.4 Programming with Information Flow

During the case studies' development, we obtained several insights into the design and implementation of applications with information flow control.

Abstractions and information flow. Information flow analysis tends to reveal details of computations occurring behind encapsulation boundaries, making it important to design abstractions carefully. Unless sufficient care is taken during design, abstractions will need to be modified during implementation. For example, we sometimes needed to change a method's signature several times, both while implementing the method body (and discovering flows we hadn't considered during design), and while calling the method in various contexts (as method invocation may reveal information to the callee, which we hadn't considered when designing the signature).

Coding idioms. We found that certain coding idioms simplified reasoning about information flow, by putting code in a form that either allowed the programmer to better understand it, or allowed Jif's type system to reason more precisely about it. As a simple example, consider the following (almost) equivalent code-snippets for assigning the result of method call `o.m()` to `x`, followed by an assignment to `y`:

1. `x = o.m();  y = 42;`
2. `if (o != null) { x = o.m(); }  y = 42;`

The first snippet throws a `NullPointerException` if `o` is null, and thus information about the value of `o` flows to `x`, and also to `y` (since the assignment to `y` is executed only in the absence of an exception). The information flow to `y` is subtle, and a common trap for new Jif programmers. In the second snippet,

no exception can be thrown (the compiler detects this with a data-flow analysis), and so information about `o` does not flow to `y`. This snippet avoids the subtle implicit flow to `y`. More generally, making implicit information flow explicit simplifies reasoning about information flow.

Declarative security policies. Many of the case studies' security requirements were expressed using Jif labels. SIF and the Jif compiler ensure that these labels (and thus the security requirements) are enforced end-to-end. In general, Jif's declarative security policies can relieve the programmer of enforcing security requirements programmatically, and give greater assurance that the requirements are met. This argues for even greater expressiveness in security policies, to allow more application security requirements to be captured, and to verify that programs enforce these requirements.

2.5 Related Work

The most closely related work is Li and Zdancewic's [42], which proposes a security-typed PHP-like scripting language to address information-flow control in web applications. Their system has not been implemented. It assumes a strongly-typed database interface, and, like SIF, ensures that applications respect the confidentiality and integrity policies on data sent to and from the database. Their security policies can express *what* information may be downgraded; in contrast, the decentralized label model used in Jif specifies *who* needs to authorize downgrading. In a multi-user web application with mutually distrusting users, the concept of *who* a session or process is executing on behalf of is crucial to security. We believe that practical information-flow control will

ultimately need to specify multiple aspects of downgrading [73]; extending the decentralized label model to reason about other downgrading aspects is ongoing work.

Huang et al. [34], Xie and Aiken [95], and Jovanovic et al. [35] all present frameworks for statically analyzing information flow in PHP web applications. Xie and Aiken, and Jovanovic et al. track information integrity using a dataflow analysis, while Huang et al. extend PHP's type system with type state. Livshits and Lam [45] use a precise static analysis to detect vulnerabilities in Java web applications. Each of these frameworks has found previously unknown bugs in web applications. Xu et al. [96], Halfond and Orso [32] and Nguyen-Tuong et al. [62] use dynamic information-flow control to prevent attacks in web applications. All of these approaches use a simple notion of integrity: information is either *tainted* or *untainted*. While this suffices to detect and prevent certain web application vulnerabilities, such as SQL injection, it is insufficient for modeling more complex, application-level integrity requirements that arise in applications with multiple mutually distrusting principals. Also, they do not address confidentiality information flows, and thus do not control the release of sensitive server-side information to web clients.

Xu et al. [97] propose a framework for analyzing and dynamically enforcing client privacy requirements in web services. They focus on web service composition, assuming that individual services correctly enforce policies. Their policies do not appear suitable for reasoning about the security of mutually distrusting users. Otherwise, this work is complementary, as we provide assurance that web applications enforce security policies.

While there has been much recent work on language-based information flow

(see [72, 73] for recent surveys), comparatively little has focused on creating real systems with information flow security, or on languages and techniques to enable this. No prior work has built real applications that enforce both confidentiality and integrity policies while dealing securely with their interactions.

The most realistic prior application experience is that of Hicks et al. [33], who use an earlier version of Jif to implement a secure CDIS email client, JPmail. Although there are similarities between JPmail and the CDIS mail application described here, SIF is a more convincing demonstration of information flow control in three ways. First, SIF is a reusable application framework, not just a single application. Second, SIF applications enforce integrity, not just confidentiality, and they ensure that declassification is robust [11]. Third, SIF applications can dynamically extend the space of principals and labels and define their own authentication mechanisms; JPmail relies on mechanisms for principal management and authentication that lie outside the scope of the application.

Askarov and Sabelfeld [4] use Jif to implement cryptographic protocols for mental poker. They identify several useful idioms for (and difficulties with) writing Jif code; recent extensions to Jif should assuage many of the difficulties.

Praxis High Integrity System's language SPARK [6] is based on a subset of Ada, and adds information-flow analysis. SPARK checks simple dependencies within procedures. FlowCaml [69] extends the functional language OCaml with information-flow security types. Like SPARK, it does not support features needed for real applications: downgrading, dynamic labels, and dynamic and application-defined principals.

Asbestos [26], Histar [104], and SELinux [47] are operating systems that track

information flow for confidentiality and integrity. To varying degrees, they provide flexible security labels and application-defined principals. However, these systems are coarse-grained, tracking information flow only between processes. Information flow is controlled only dynamically, which is imprecise, and creates additional information flows from run-time label checking. By contrast, Jif checks information flow mostly statically, at the granularity of program variables, providing increased precision and greater assurance that a program is secure prior to deployment. Asbestos has a web server that allows web applications to isolate users' data from one another, using one process per user. All downgrades are performed by trusted processes. Unlike Jif, this granularity of information flow tracking does not permit different security policies for different data owned by a single user.

Tse and Zdancewic [91] present a monadic type system for reasoning about dynamic principals, and certificates for authority delegation and downgrading. Jif 3.0's dependent type system for dynamic labels and principals allows similar reasoning. Tse and Zdancewic assume that certificates are contained in the external environment, and do not provide a mechanism to dynamically create them. Closures in Jif 3.0 can be dynamically authorized, and may perform arbitrary computation, whereas Tse and Zdancewic's certificates permit only authority delegation and downgrading.

Swamy et al. [86] consider dynamic policy updates, and introduce a transactional mechanism to prevent *unintentional transitive flows* that may arise from policy updates. In Jif, policies are updated dynamically by adding and removing principal delegations, and unintentional transitive flows may occur. Their techniques are complementary to our work, and should be applicable to Jif to

stop these flows.

2.6 Conclusions

We have designed and implemented Servlet Information Flow (SIF), a novel framework for building high-assurance web applications. Extending the Java Servlet framework, SIF addresses trust issues in web applications, moving trust out of web applications and into SIF and the Jif compiler.

SIF web applications are written entirely in the Jif 3.0 programming language. At compile time, applications are checked to see if they respect the confidentiality and integrity of information held on the server: confidential information is not released inappropriately to clients, and low-integrity information from clients is not used in high-integrity contexts. SIF tracks information flow both within the handling of a single request, and over multiple requests—it closes the loop of information flow between client and server.

Jif 3.0 extends Jif in several ways to make web applications possible. It adds sophisticated dynamic mechanisms for access control, authentication, delegation, and principal management, and shows how to integrate these features securely with language-based, largely static, information-flow control.

We have used SIF to implement two applications with interesting information security requirements. These web applications are among the first to statically enforce strong and expressive confidentiality and integrity policies. Many of the applications' security requirements were expressible as security labels, and are thus enforced by the Jif 3.0 compiler.

CHAPTER 3

INFORMATION FLOW CONTROL ACROSS WEB APPLICATION TIERS

The SIF framework described in Chapter 2 uses a third party SQL database as its persistent back-end store. The calendar application, for example, stores the details of events in the database, through a Java module that performs the format conversion between objects and relational tables. Although SIF provides the necessary components of a three-tier web application architecture, it does not provide true end-to-end information flow control. Code running in the database, in the form of SQL queries, does not track security labels. This poses a significant threat to web applications, especially those with substantial application logic encoded in SQL queries.

SIF also does not track information flow through arbitrary code running on the client. Information flow is tracked only through simple UI widgets such as form fields and submit buttons. SIF allows deployers to include JavaScript code to run on the client, but expects the code to be already checked for security properties. This can be problematic if, for instance, the client side JavaScript copies text from a form field with a certain security label to another form field with a lower security label.

This chapter demonstrates how both the problems can be solved. Section 3.1 presents a system that integrates SIF with a distributed persistent storage system called Fabric [44]. A web application server in this system runs as a Fabric worker, storing persistent objects in a Fabric store. SQL queries are replaced with remote method calls to the store. This enables an end-to-end information flow analysis of the entire web application. Section 3.2 onwards till the end of the chapter presents the Swift system which tracks information flow to and

from the client, as well as information flow within client side JavaScript code. Swift also provides a higher-level programming abstraction where the developer writes a single sequential program, which is automatically distributed between the client and the server by the compilers. The distribution is constrained by security policies and fine-tuned by performance concerns.

3.1 Tracking Information Flow through the Persistence Tier

This section discusses how information flow can be tracked between the server and the persistent storage tier. Section 3.1.1 provides a summary of the Fabric system and language. Section 3.1.2 describes the integration of SIF with Fabric. The use of Fabric instead of an SQL database enables true end-to-end information flow tracking. Section 3.1.3 presents an evaluation of the integrated system using an Travel application that was built using the integrated system.

3.1.1 The Fabric System

Fabric is a system for building distributed applications with secure sharing of information, computation and storage between heterogeneously trusted hosts [44]. There are two kinds of hosts in the system: *workers* and *stores*. Workers engage in distributed transactions with other workers and perform the bulk of the computation in the system. Stores are responsible for persistent storage of objects. Stores can also have a co-located worker that runs computations that use a large number of objects from the associated store. This enables *function shipping* which is useful when moving the computation to the data is better for performance (and sometimes security) than moving the data to the host per-

forming the computation. There is also a third kind of host called a *dissemination node* that keeps object replicas to enhance the availability of objects; however, this work will not be concerned about them. Hosts can join and leave the system freely without the need for permission from a centralized authority. Thus, Fabric is a decentralized system, similar to the World Wide Web.

The Fabric language is an extension of Jif [61], with the same kind of label annotations on variables, method arguments, object fields, method begin and end contexts, return values, exceptions, etc. In addition, each worker and store has an associated principal that can be used within labels. The following constructs for distribution, transactions and persistence are added in Fabric:

- Remote method calls for performing distributed computations and function shipping.
- Atomic blocks for marking statements that need to run as an atomic transaction, to preserve data consistency. Transactions can be nested.
- Store annotations on constructor calls for specifying persistence of objects, and for checking if the store is trusted to enforce the security of the object.
- Transparent access to objects everywhere, regardless of whether the object is local or remote, persistent or transient. The system automatically performs the required data shipping.

To enable the new language constructs, the Fabric language integrates information flow with persistence, transactions and distributed computation. Integration with persistence requires the definition of the *trust ordering* between labels, and the addition of associated label checks in the source. Section 2.2.3 explains how the set of labels in Jif, based on the DLM [58], forms a lattice with the

partial order between labels describing which labels are more restrictive than which other labels. This ordering between labels is called the *information flow ordering*, denoted by $\sqsubseteq$. Fabric extends this model, and thus the DLM, by defining a second ordering on labels called the *trust ordering*, denoted by $\preccurlyeq$. The trust ordering is useful for reasoning about the enforcement of policies by a partially trusted platform. For instance, an object can be constructed on a store only if the store's label is higher in the trust ordering than the object's label.

Integration with atomic transactions requires changes to the compiler that stem from the observation that a transaction abort can be a potential implicit flow. To track this, we require the *pc* label at the abort to be bounded above by the *pc* label at the start of the atomic block, i.e. $L_{pc}^{abort} \sqsubseteq L_{pc}^{atomic}$. Integration with distributed computation requires label checks at each remote method call that ensure that the callee is allowed to view the method arguments and that the return value has at most the integrity of the callee. During a distributed transaction between mutually distrusting hosts, the Fabric runtime automatically ships objects and object updates to hosts where they are needed. This is done using a novel data structure called a *writer map* that encrypts object contents to ensure that only hosts that are allowed to read an object receive updates for that object.

Fabric can be used to build complex distributed information systems such as those used by banks, hospitals, enterprises and government agencies. The mechanisms offered by Fabric help integrating information from different domains securely, enabling new kinds of services and capabilities. Next, we see how the SIF system can benefit from using Fabric as a back end persistent store, instead of a database.

3.1.2 Integrating SIF and Fabric

We call the system that integrates SIF and Fabric as SIF-Fabric. In SIF-Fabric, the web application server runs as a Fabric worker. Similar to the SIF system described in Chapter 2, the framework classes are Java classes and the application logic is encoded partly in Jif classes and partly in Fabric classes. Both Java and Jif classes are non-persistent; application data that is required to be persistent and the computation over the persistent data is encoded in persistent Fabric objects. For instance, going back to the Calendar example from Section 2.4, the event and user information would be stored as Fabric objects on a store, instead of relational tables in a database. The `Action` classes would be non-persistent Jif objects and the `Servlet` and `Request` framework classes would be non-persistent Java objects.

The Fabric language allows persistent objects to interoperate with Jif and Java objects. Jif and Java objects are both non-persistent; the difference between them is that Jif objects are statically checked against an information flow policy, whereas Java objects are not. Both Java and Jif classes have associated Fabric *signatures* that provide an interface to interact with instances of the class as if they were Fabric objects. The interoperability enables running the SIF framework classes, the web application logic and code from persistent Fabric objects all on the same Fabric worker. In addition, a Servlet container such as Tomcat also executes within the same Java VM of the Fabric worker, hosting the web application and serving web pages to a browser on any client over the network.

The integration of SIF with Fabric required modifications to various levels of the web application stack. The SIF framework classes were modified to ensure conformance to Fabric transaction protocols. Database queries in the appli-

cation code were replaced with remote method calls to the appropriate Fabric store. Object persistence required modifications to the sublanguage of security annotations. Each of these modifications is discussed in further detail.

A client request from a browser is handled in almost the same way as in SIF. Referring to Figure 2.1 and Section 2.2.4, steps 1 and 2 are identical. In step 3, the appropriate `Action` object is invoked within a top level atomic transaction. This is to ensure that construction of and comparisons between dynamic label objects happen within a transaction, required because labels in Fabric are always persistent objects. This top level transaction also nests inner transactions created by the application for manipulating persistent objects. If the `Action` fails for some reason (e.g. failed label check, failure of the persistent store, etc.) the transaction is aborted and an empty HTML page is sent back to the client. The transaction abort also rolls back any changes to persistent objects, including label objects, incurred within the transaction. No network messages are sent during the `Action` and so their rollback is not an issue. Fabric does not roll back side-effects by Java and Jif objects. However, in this case, it is not an issue since Java and Jif objects do not write to any persistent storage, nor do they send messages over the network. However, Java code in the servlet container does send HTTP response messages to the client, but only after the completion of an `Action`. Thus, the state of all Java and Jif objects can safely be ignored on a transaction abort. The ability to roll back a failed `Action` is a new feature in SIF-Fabric, that ensures that persistent data will not enable a storage covert channel.

Using Fabric in place of a relational database enables the programmer to encode database queries as Java-like code using iteration constructs within Fabric

objects. An SQL query is replaced with a remote method call to a method in a Fabric object that encodes the logic of the query and a store that has the bulk of the objects that are queried over. The programmer also has to ensure that the remote call satisfies all the necessary compiler and runtime checks for security. At the call site, this involves statically checking that the method arguments are readable by the callee host and that the return value is not more trusted than the label of the callee. The callee of the remote call has no way of statically knowing who the caller is. Thus, each potential remote call is surrounded by runtime label checking code that ensures that the caller is trusted to enforce the labels of method arguments as well as the label of the return value. This extra wrapper code is statically label checked to ensure that it does not leak information itself. The extra label checking for remote method calls is in addition to the regular method call label checking done by Jif. Replacing SQL queries with Fabric code not only removes the "impedance mismatch" between the general purpose language and the query sublanguage resulting in cleaner and easy to maintain code, but also enables end-to-end information flow enforcement.

Fabric adds two built-in principals: `store$` and `worker$`. The principal `store$` refers to the store that has the authoritative copy of the current object and `worker$` refers to the current worker performing the computation. Both the built-in principals can be used in labels as regular principals. They are especially useful in carrying out dynamic label and principal tests to ensure that all security constraints are being met.

In Fabric, each object has an *object label* that determines the encryption key used to encrypt it. The object label is computed as the join of the labels on all its fields. Labels and Principals in Fabric are themselves first class persistent

objects and have an object label of their own. Since the object label has to be maintained at runtime to determine encryption/decryption keys, the labels on fields need to have a runtime representation. This creates a problem for any use of the `{this}` label on final fields, since the `{this}` label is not runtime representable. Fabric resolves this problem by performing static checking exactly as in the Jif language, but translates all occurrences of `{this}` to $\{\bot \rightarrow; \top \leftarrow\}$ i.e. public and trusted. The reasoning that this is secure is that the programmer intended the field to be protected at the same level as the reference to the object, and thus intended the object reference to be a capability for access to the field. Since Fabric uses cryptographically unguessable object ids at runtime, they can be used as capabilities in such a manner.

3.1.3 The Travel Example

This section discusses the evaluation of SIF-Fabric by implementing an airline ticket purchasing web application using the system. The application allows a customer to purchase an airline ticket if he has enough account balance. Both the customer and the airline maintain their accounts with the same bank. Once the customer submits a purchase request, an amount of money equal to the price of the ticket is deducted from the customer's account and credited to the airline's account. Since the customer, airline and bank do not completely trust each other, a broker is required to interact with each of them and carry out the transaction.

The application comprises three web interfaces one each for the customer, airline and bank, with each web interface running on a worker for the cor-

responding party. For instance, the web interface for bank runs on the bank worker, requires the bank principal to log in before it displays a list of all accounts at the bank, and their balances. Similarly, the web interface for airline runs on the airline worker, requires the airline principal to log in before it displays the number of tickets sold to the customer. The web interface for the customer runs on the customer worker and does not require the customer to log in, since he is the sole user of the interface. The customer web application provides an interface to enter a payment amount and issues a ticket to the customer. The code for the application can be found in the Fabric 0.1 release [43] in the `examples/travel` directory. The directory also contains a `README` that explains how to run the application.

A typical run of the application proceeds as follows. First, as part of the initial setup, the following three stores are created: `bank`, `airline` and `broker` and the following three workers are created: `customer`, `bankweb` and `airlineweb`. A static principal `BankPrincipal` representing the bank principal is created. The store `bank` and the worker `bankweb` run on the same host, and the principals (corresponding to) `bank` and `bankweb` act for the `BankPrincipal`. Similarly, the principals `airline` and `airlineweb` act for `AirlinePrincipal` and `customer` acts for `CustomerPrincipal`. The `Broker` class implements the principal for the broker, which acts for `AirlinePrincipal` and `BankPrincipal`. All principal objects are stored on the `broker` store. The `bank` store is initialized with the account information of the airline and the customer. The `airline` store is initialized with a record of the number of tickets sold. After the stores are initialized, the web applications for the customer, bank and the airline are started on the `customer`, `bankweb` and `airlineweb` workers respectively.

After the initial setup, the customer can walk up to his worker `customer` at any time and connect to the local web server using a standard web browser. A login is not required, since the customer is assumed to be the only user of the `customer` worker. The web interface asks the customer to enter the payment amount and click on the `Buy Ticket` button. Clicking the button initiates the `StartTransaction` action, which first fetches the instance of `Broker` from the store `broker`. The `Broker` instance maintains a reference to the customer and airline accounts on the `bank` store and a reference to the tickets object on the `airline` store. After fetching the `Broker` object, the action performs sanity checks to ensure that the delegation relationships between principals is as expected and invokes the `coordinatePurchase` method on the `Broker` instance remotely, to have it run on the worker colocated with the store `broker`. The amount entered by the customer is passed as a method argument. The remote call is needed since only the broker has the authority to perform all the coordination tasks.

Inside the `coordinatePurchase` method, the broker performs the same sanity checks since for delegation relationships between principals, since it does not trust the customer to have done it correctly. If the checks are successful, it first invokes a remote method call to the `debit` method in the customer account on the worker colocated with `bank`. This debits the amount from the customer's account. Next, a remote call is made to the `credit` method in the airline account object on the worker colocated with `bank`. This credits the amount to the airline's account. Next, the `incTickets` method in tickets object is invoked on the worker colocated with `airline`. This increments the number of tickets sold. The return value of each of these methods is a boolean that indicates whether the task was completed successfully.

All the three tasks happen within an atomic transaction, ensuring an all-or-none semantics. The `coordinatePurchase` method returns true if all the three tasks were successful. This returns control to the `customer` worker within the `StartTransaction` action. Depending on the return value of `coordinatePurchase`, a web page is created to report the result to the customer and sent to the customer's browser.

Simultaneously, at any time, a person authorized by the bank can connect to the webserver running on the `bankweb` worker using a standard web browser. All `Action` objects in the bank web application inherit the `AuthenticatedAction` class, which requires authentication by the `BankPrincipal` before any action is performed. The person can authenticate himself by supplying the username and password for `BankPrincipal`, exactly like in SIF (see Section 2.4.2). On a successful login, all account objects from the `bank` store are fetched and a web page is returned to the browser containing the details of the accounts found. No interaction with any other stores or the broker is required. Similarly, a person authorized by the airline can connect to the `airlineweb` worker and login to view the total number of tickets and the number of tickets unsold.

The example demonstrates that real web applications can be developed securely, using persistent objects and atomic transactions and that the language of SIF-Fabricis expressive for this purpose. Moreover, in comparison to a similar implementation using a relational database for persistence, the SIF-Fabricimplementation is more concise and enforces true end-to-end information flow security.

3.2 Tracking Information Flow through Client-side Code

Web applications are client–server applications in which a web browser provides the user interface. They are a critical part of our infrastructure, used for banking and financial management, email, online shopping and auctions, social networking, and much more. The security of information manipulated by these systems is crucial, and yet these systems are not being implemented with adequate security assurance. In fact, web applications are recently reported to comprise 69% of all Internet vulnerabilities [88]. The problem is that with current implementation methods, it is difficult to know whether an application adequately enforces the confidentiality or integrity of the information it manipulates.

Recent trends in web application design have exacerbated the security problem. To provide a rich, responsive user interface, application functionality is pushed into client-side JavaScript [27] code that executes within the web browser. JavaScript code is able to manipulate user interface components and can store information persistently on the client side by encoding it as cookies. These web applications are distributed applications, in which client- and server-side code exchange protocol messages represented as HTTP requests and responses. In addition, most browsers allow JavaScript code to issue its own HTTP requests, a functionality used in the Ajax development approach (Asynchronous JavaScript and XML).

With application code and data split across differently trusted tiers, the developer faces a difficult question: when is it secure to place code and data on the client? All things being equal, the developer would usually prefer to run code

and store data on the client, avoiding server load and client–server communication latency. But moving information or computation to the client can easily create security vulnerabilities.

For example, suppose we want to implement a simple web application in which the user has three chances to guess a number between one and ten, and wins if a guess is correct. Even this simple application has subtleties. There is a confidentiality requirement: the user should not learn the true number until after the guesses are complete. There are integrity requirements, too: the match between the guess and the true number should be computed in a trustworthy way, and the guesses taken must also be counted correctly.

The guessing application could be implemented almost entirely as client-side JavaScript code, which would make the user interface very responsive and would offload the most work from the server. But it would be insecure: a client with a modified browser could peek at the true number, take extra guesses, or simply lie about whether a guess was correct. On the other hand, suppose guesses that are not valid numbers between one and ten do not count against the user. Then it is secure and indeed preferable to perform the bounds check on the client side. Currently, web application developers lack principled ways to make decisions about where code and data can be securely placed.

We introduce the Swift system, a way to write web applications that are *secure by construction*. Applications are written in a higher-level programming language in which information security requirements are explicitly exposed as declarative annotations. The compiler uses these security annotations to decide where code and data in the system can be placed securely. Code and data are partitioned at fine granularity, at the level of individual expressions and object

fields. Developing programs in this way ensures that the resulting distributed application protects the confidentiality and integrity of information. The general enforcement of information integrity also guards against common vulnerabilities such as SQL injection and cross-site scripting.

Swift applications are not only more secure, they are also easier to write: control and data do not need to be explicitly transferred between client and server through the awkward extralinguistic mechanism of HTTP requests. Automatic placement has another benefit. In current practice, the programmer has no help designing the protocol or interfaces by which client and server code communicate. With Swift, the compiler automatically synthesizes secure, efficient interfaces for communication.

Of course, others have noticed that web applications are hard to make secure and awkward to write. Prior research has addressed security and expressiveness separately. One line of work has tried to make web applications more secure, through analysis [34, 95, 35] or monitoring [32, 62, 96] of server-side application code. However, this work does not help application developers decide when code and data can be placed on the client. Conversely, the awkwardness of programming web applications has motivated a second line of work toward a single, uniform language for writing distributed web applications [31, 16, 76, 99, 98]. However, this work largely ignores security; while the programmer controls code placement, nothing ensures the placement is secure.

Swift thus differs from prior work by addressing both problems at once. Swift automatically partitions web application code while also providing assurance that the resulting placement enforces security requirements. Addressing both problems at the same time makes it possible to do a better job at each of

them.

Prior work on program partitioning in the Jif/split language [103, 106] has explored using security policies to drive code and data partitioning onto a general distributed system. Applying this approach to the particularly important domain of web applications offers both new challenges and new opportunities. In the Swift trust model, the client is less trusted than the server. Code is placed onto the client in order to optimize interactive performance, which has not been previously explored. Swift has a more sophisticated partitioning algorithm that exploits new replication strategies. And because Swift supports a richer programming language with better support for dynamic security enforcement, it can control information flow even as a rich, dynamic graphical user interface is used to interact with security-critical information.

The remainder of the chapter is structured as follows. Section 3.3 gives an overview of the Swift architecture. Section 3.4 describes the programming model, based on an extension of the Jif programming language [61] with support for browser-based user interfaces. Sections 3.5 and 3.6 explain how high-level Swift code is compiled into an intermediate language, WebIL, and then partitioned into Java and JavaScript code. Section 3.7 presents results and experience using Swift, Section 3.8 discusses related work, and Section 3.9 concludes.

3.3 Architecture

Figure 3.1 depicts the architecture of Swift. The system starts with annotated Java source code at the top of the diagram. Proceeding from top to bottom,

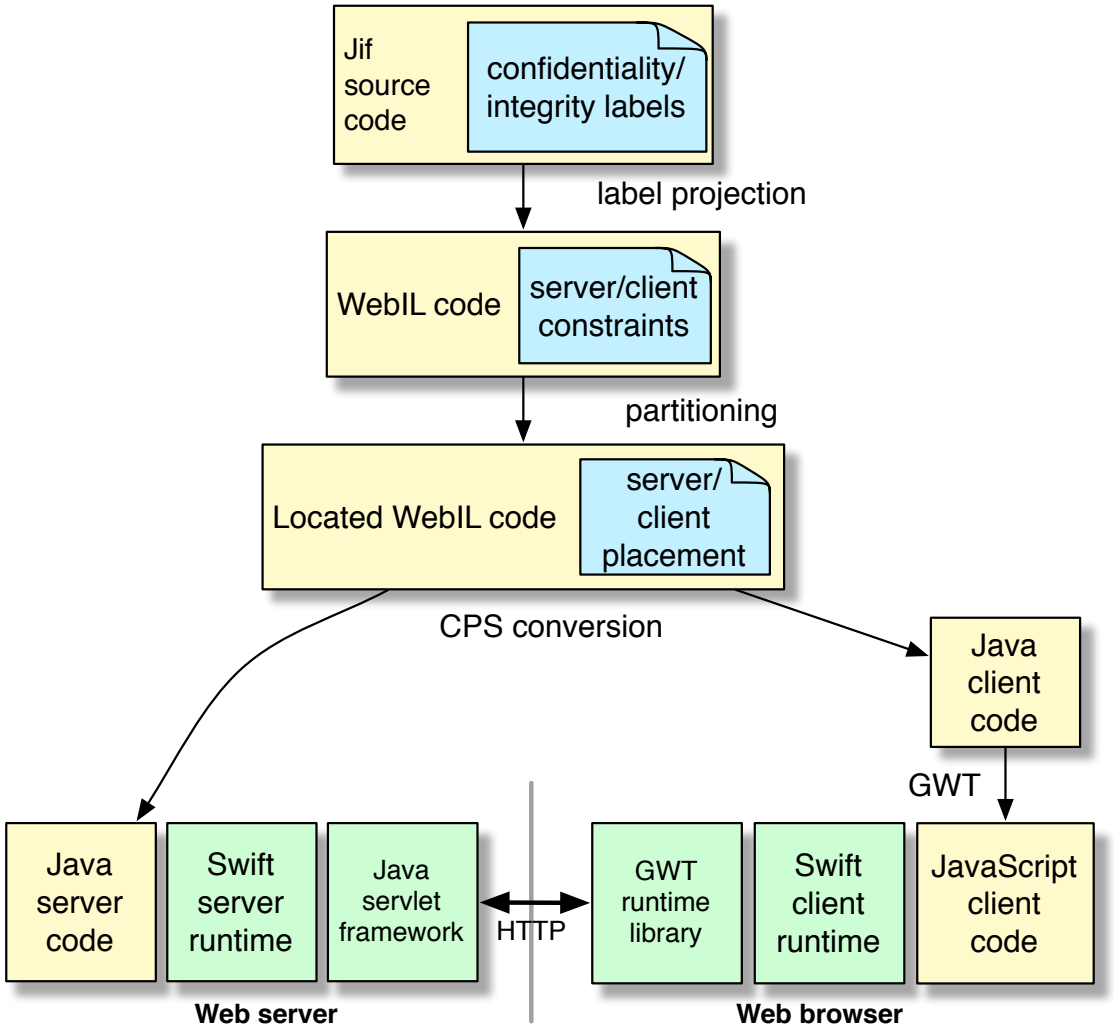

Figure 3.1: The Swift architecture

a series of program transformations converts the code into a partitioned form shown at the bottom, with Java code running on the web server and JavaScript code running on the client web browser.

Jif source code. The source language of the program is an extended version of the Jif 3.0 programming language [56, 61]. Jif extends the Java programming language with language-based mechanisms for information flow control and

access control. Information security policies can be expressed directly within Jif programs, as *labels* on program variables. By statically checking a program, the Jif compiler ensures that these labels are consistent with flows of information in the program.

The original model of Jif security is that if a program passes compile-time static checking, and the program runs on a trustworthy platform, then the program will enforce the information security policies expressed as labels. For Swift, we assume that the web server can be trusted, but the client machine and browser may be buggy or malicious. Therefore, Swift must transform program code so that the application runs securely, even though it runs partly on the untrusted client.

WebIL intermediate code. The first phase of program transformation converts Jif programs into code in an intermediate language we call WebIL. As in Jif, WebIL types can include annotations; however, the space of allowed annotations is much simpler, describing constraints on the possible locations of application code and data. For example, the annotation `S` means that the annotated code or data must be placed on the web server. The annotation `C?S` means that it must be placed on the server, and *may* optionally be replicated on the client as well. WebIL is useful for web application programming in its own right, although it does not provide security assurance.

WebIL optimization. The initial WebIL annotations are merely constraints on code and data placement. The second phase of compilation decides the exact placement and replication of code and data between the client and server, in accordance with these constraints. The system attempts to minimize the cost of

the placement, in particular by avoiding unnecessary network messages. The minimization of the partitioning cost is expressed as an integer programming (IP) problem, and maximum flow methods are then used to find a good partitioning.

Splitting code. Once code and data placements have been determined, the compiler transforms the original Java code into two Java programs, one representing server-side computation and the other, client-side computation. This is a fine-grained transformation. Different statements within the same method may run variously on the server and the client, and similarly with different fields of the same object. What appeared as sequential statements in the program source code may become separate code fragments on the client and server that invoke each other via network messages. Because control transfers become explicit messages, the transformation to two separate Java programs is similar to a conversion to *continuation-passing style* [70, 80].

JavaScript output. Although our compiler generates Java code to run on the client, this Java code actually represents JavaScript code. The Google Web Toolkit (GWT) [31] is used to compile the Java code down to JavaScript. On the client, this code then uses the GWT run-time library and our own run-time support. On the server, the Java application code links against Swift's server-side run-time library, which in turn sits on top of the standard Java servlet framework.

The final application code generated by the compiler uses an Ajax approach to securely carry out the application described in the original source code. The application runs as JavaScript on the client browser, and issues its own HTTP

requests to the web server, which responds with XML data.

From the browser's perspective, the application runs as a single web page, with most user actions (e.g., clicking on buttons) handled by JavaScript code. This approach seems to be the current trend in web application design, replacing the older model in which a web application is associated with many different URLs. One result of the change is that the browser "back" and "forward" buttons no longer have the originally intended effect on the web application, though this can be largely hidden, as is done in the GWT.

Partitioning and replication. Compiling a Swift application puts some code and data onto the client. Code and data that implement the user interface clearly must reside on the client. Other code and data are placed on the client to avoid the latency of communicating with the server. With this approach, the web application can have a rich, highly responsive user interface that waits for server replies only when security demands that the server be involved.

In order to enforce the security requirements in the Jif source code, information flows between the client and the server must be strictly controlled. In particular, confidential information must not be sent to the client, and information received from the client cannot be trusted. The Swift compilation process generates code that satisfies these constraints.

One novel feature of Swift is its ability to selectively replicate computation onto *both* the client and server, improving both responsiveness and security. For example, validation of form inputs should happen on the client so the user does not have to wait for the server to respond when invalid inputs are provided. However, client-side validation should not be trusted, so input validation must

also be done on the server. In current practice, developers write separate validation code for the client and server, using different languages. This duplicates effort and makes it less likely that validation is done correctly and consistently. With Swift, the compiler can automatically replicate the same validation code onto both the server and the client. This replication is not a special-purpose mechanism; it is simply a result of applying a general-purpose algorithm for optimizing code placement.

In the next few sections, we more closely examine the various compilation phases illustrated in Figure 3.1.

3.4 Writing Swift applications

3.4.1 Extending Jif 3.0

Programming with Swift starts with a program written in the Jif programming language [56, 61], with a few extensions. Section 2.2.3 gives a background on the basic language constructs in Jif.

In addition, two principals are already built into Swift programs. The principal $\star$ (also `server`) represents the maximally trusted principal in the system. The principal `client` represents the other end of the current session—in ordinary, non-malicious use, a web browser under the control of a user. When reasoning about security, we can only assume that the client is the other end of a network connection, possibly controlled by a malicious attacker. Because the server is trusted, the principal $\star$ acts for `client`. The client may see infor-

mation whose confidentiality is no greater than $\star \rightarrow$client, and can produce information with integrity no greater than $\star \leftarrow$client.

A Swift program may use and even create additional principals, for example to represent different users of a web application. The language constructs to express the creation of application-specific principals and dynamic principals are also part of the Jif language, as explained in Sections 2.3.1 and 2.3.2. For a user to log in as principal bob, server-side application code trusted by bob must establish that the principal named by client acts for bob. Applications can define their own authentication methods for this purpose. Once the relationship exists, the client can act for bob; for example, information labeled alice→bob could be released to that client.

There are actually multiple principals denoted by client, whose identity is determined by which client initiated the current request. To prevent different session principals named as client in the code from being confused with each other, the Swift compiler requires that the types of static variables not reference the principal client, even indirectly. This works because different Swift sessions can only interact or access shared persistent state through static variables.

3.4.2 A Sample Application

The key features of the Swift programming model can be seen by studying a simple web application written using Swift. Figure 3.2 shows key fragments of the Jif source code of the number-guessing web application described in Section 3.2. Java programmers will recognize this code as similar to that of an ordinary single-machine Java application that uses a UI library. For example,

```
1  public class GuessANumber {
2    final label{⋆←⋆} cl = new label{⋆→client};
3    int{⋆→⋆; ⋆←⋆} secret;
4    int{⋆→client; ⋆←⋆} tries;
5    ...
6    private void setupUI{⋆→client}() {
7      guessbox = new NumberTextBox("");
8      message = new Text("");
9      button = new Button("Guess");
10     ...
11     rootpanel.addChild(cl, cl, guessbox);
12     rootpanel.addChild(cl, cl, button);
13     rootpanel.addChild(cl, cl, message);
14   }
15   void makeGuess{⋆→client}(Integer{⋆→client} num)
16     where authority(⋆), endorse({⋆←⋆})
17     throws NullPointerException
18   {
19     int i = 0;
20     if (num != null) i = num.intValue();
21     endorse (i, {⋆←client} to {⋆←⋆})
22     if (i >= 1 && i <= 10) {
23       if (tries > 0 && i == secret) {
24         declassify ({⋆→⋆} to {⋆→client}) {
25           tries = 0;
26           finishApp("You win!");
27         }
28       } else {
29         declassify ({⋆→⋆} to {⋆→client}) {
30           tries--;
31           if (tries > 0) message.setText("Try again");
32           else finishApp("Game over");
33         }
34       }
35     } else {
36       message.setText("Out of range:" + i);
37     }
38   }
39 }
40 class GuessListener
41   implements ClickListener[{⋆→client}, {⋆→client}] {
42   ...
43   public void onClick{⋆→client} (
44         Widget[{⋆→client}, {⋆→client}]{⋆→client} w)
45         : {⋆→client}
46   {
47     if (guessApp != null) {
48       NumberTextBox guessbox = guessApp.guessbox;
49       if (guessbox != null)
50         guessApp.makeGuess(guessbox.getNumber());
51     }
52   }
53 }
```

Figure 3.2: Guess-a-Number web application

it has a user interface dynamically constructed out of widgets such as buttons, text inputs, and text labels. Swift widgets are similar to those in the Google Web Toolkit [31], communicating via events and listeners. The crucial difference is that Swift controls how information flows through them.

The core application logic is found in the `makeGuess` method (lines 15–39). Aside from various security label annotations, this method is essentially straight-line Java code. To implement the same functionality with technologies such as JSP [9] or GWT requires more code, in a less natural programming style with explicit control transfers between the client and server.

The code contains various labels expressing security requirements. Because this example is very simple, just the principals `client` and `*` are used in these labels. For example, on line 3, the variable `secret` is declared to be completely secret (`*→*`) and completely trusted (`*←*`); the variable `tries` on the next line is not secret (`*→client`) but is just as trusted. Because Jif checks transitively how information flows within the application, the act of writing just these two label annotations constrains many of the other label annotations in the program. The compiler ensures that all label annotations are consistent with the information flows in the program.

The user submits a guess by clicking the button. A listener attached to the button passes the guess (line 50) to `makeGuess`. The listener reads the guess from a `NumberTextBox` widget that only allows numbers to be entered.

The `makeGuess` method receives a guess `num` from the client. The variable `num` is untrusted and not secret, as indicated by its label `{*→client}` on line 15. The label after the name of the method, also `{*→client}`, is the *be-*

gin label of the method. It bounds what might be learned from the fact that the method was invoked, by preventing callers from causing any greater implicit flow. Jif also keeps track of implicit flows out of methods using *end labels*; in the case of `makeGuess`, no additional annotations are required for this purpose because the end label is the same as the begin label. Jif aims to protect the confidentiality and integrity of program data rather than of program code. However, end labels and begin labels can be used to respectively protect the confidentiality and integrity of code.

The code of `makeGuess` checks whether the guess is correct, and either informs the user that he has won, or else decrements the remaining allowed guesses and repeats. Because the guess is untrusted, Jif will prevent it from affecting trusted variables such as `tries`, unless it is explicitly *endorsed* by trusted code. Therefore, lines 21–37 have a *checked endorsement* that succeeds only if `num` contains an integer between one and ten. If the check succeeds, the number `i` is treated as a high-integrity value within the "then" clause. If the check fails, the value of `i` is not endorsed, and the "else" clause is executed. Checked endorsements are a Swift-specific Jif extension that makes the common pattern of validating untrusted inputs both explicit and convenient.

By forcing the programmer to use `endorse`, the potential security vulnerability is made explicit. In this case, the endorsement of `i` is reasonable because it is intrinsically part of the game that the client is allowed to pick any value it wants (as long as it is between one and ten).

Similarly, some information about the secret value `secret` is released when the client is notified whether the guess `i` is equal to `secret`. Therefore, the bodies of both the consequent and the alternative of the `if` test on line 23

must use an explicit `declassify` to indicate that information transmitted by the control flow of the program may be released to the client. Without the `declassify`, client-visible events—showing messages, or updating the variable `tries`—would be rejected by the compiler.

The `declassify` and `endorse` operations are inherently dangerous. Jif controls the use of `declassify` and `endorse` by requiring that they occur in a code marked as trusted by the affected principals; hence the clauses `authority(*)` and `endorse({*←*})` on line 16. The latter, *auto-endorse* annotation means that an invocation of `makeGuess` is treated as trusted even if it comes from the client. Jif also enforces a security property of *robust declassification* [11], in which declassification cannot be performed without sufficient integrity. Untrusted information is not allowed to affect security-critical operations such as declassification, even indirectly.

3.4.3 Swift User Interface Framework

Swift programs interact with the user via a user interface framework. This framework abstracts away the details of the underlying HTML and JavaScript, allowing programming in a event-driven style familiar to users of UI frameworks such as Swing. The control of information flow in a rich, interactive, dynamically changing graphical user interface is a novel feature of Swift.

Figure 3.3 presents part of the signatures of several Swift UI framework classes. The class `Widget` is the ancestor of all user interface widgets, such as `TextBox` (which allows a user to enter text), `Button` (which represents a clickable button), and `Panel` (which contains other widgets).

```
1  class Widget[label Out, label In] { ... }
2  class Panel[label Out, label In]
3   extends Widget[Out,In] {
4    void addChild{Out}(label wOut,
5                       label wIn,
6                       Widget[wOut,wIn]{Out} w)
7      where {*wOut} <= Out, {In;w} <= {*wIn};
8  }
9  class ClickableWidget[label Out, label In]
10  extends Widget[Out,In] {
11   void addListener{In}
12     (ClickListener[Out,In]{In} li);
13 }
14 class Button[label Out, label In]
15  extends ClickableWidget[Out,In] {
16   String{Out} getText();
17   void setText{Out}(String{Out} text);
18 }
19 interface ClickListener[label Out, label In] {
20   void onClick{In}(Widget[Out, In]{In} b);
21 }
```

Figure 3.3: UI framework signatures

All classes in the framework are annotated with security policies that track information flow that may occur within the framework. The framework ensures that the client is permitted to view all information that the user interface displays. Conversely, all information received from the user interface is annotated as having been tainted by the client.

The user interface classes demonstrate an important feature of Jif. Classes may be parameterized with respect to principals or labels, as indicated by the parameters in brackets following the name of each class. The Jif parameterization mechanism is superficially similar to the parameterized type mechanism in recent versions of Java, but differs in that parameter values are usable at run time.

All widget classes are parameterized on two security labels, `Out` and `In`. The parameter `Out` is an upper bound on the security labels of information that is contained in the widget, or its children. Thus, given labels ℓ and ℓ', the text displayed on a `Button[`ℓ`,`ℓ'`]` object must have a security label no more restrictive than ℓ. This restriction is evidenced by the annotations on the `getText` and `setText` methods, on lines 16–17. Similarly, given a `Panel[`ℓ`,`ℓ'`]` object to which we are adding a child `Widget[`ℓ_w`,`ℓ'_w`] w`, the label of the child's contents, ℓ_w, must be no more restrictive than the upper bound of the panel's content, ℓ. This requirement is expressed in the annotation "`where {*wOut} <= Out`" on the `addChild` method (line 7). This annotation means that the method can be called only if it is known at the call site that the label contained *in* the variable `wOut` is no more restrictive than the label `Out`. (Because `wOut` is a program variable, unlike `Out`, the label *in* `wOut` is written `{*wOut}` to distinguish it from the label *of* `wOut`, written `{wOut}`.)

The parameter `In` of a widget is an upper bound on information that may be gained by knowing an event occurred on the widget. Thus, if a `ButtonListener[`ℓ`,`ℓ'`]` is added as a listener to a `Button[`ℓ`,`ℓ'`]` object, ℓ' is an upper bound on information that the listener may learn by having the `onClick` method invoked. This is shown by the occurrences of the label `{In}` in the `addListener` and `onClick` method signatures on lines 12 and 20. For example, the first `{In}` in the `onClick` signature means that the method can be called only if the implicit information flow into the method is bounded above by `In`.

What information do we learn by knowing an event occurs on a widget? We can at least infer that the widget is displayed to the user, and thus that the

widget is reachable from the root panel. For example, suppose an application creates button `bt` if the value of a secret boolean `v` is `true`, and button `bf` if the value is `false`; a listener to `bt` can then infer the value of `v` upon invocation of the `onClick` method. Thus, the `In` parameter for `bt` must be at least as restrictive as the security label for the boolean `v`. More generally, if a `Widget[`ℓ_w`,`ℓ'_w`]` `w` is added to a `Panel[`ℓ`,`ℓ'`]` `p`, the security label ℓ'_w must be at least as restrictive as the security label of widget `w`. In addition, since an event on `w` can only occur if the panel `p` is itself added to the UI, we also require that ℓ'_w is at least as restrictive as ℓ'. Both of these restrictions are expressed in the annotation "`where {In;w} <= {*wIn}`", on line 7.

3.5 WebIL

After the Swift compiler has checked information flows in the Jif program, the program is translated to an intermediate language, WebIL. WebIL extends Java with placement annotations for both code and data. Placement annotations define constraints on where code and data may be replicated. These constraints may be due to security restrictions derived from the Jif code, or to architectural restrictions (for example, calls to a database must occur on the server, and calls to the UI must occur on the client).

Whereas Jif allows expression and enforcement of rich security policies from the decentralized label model (DLM) [59], the WebIL language is concerned only with the placement of code and data onto two host machines, the server and the client. Thus, when translating to WebIL, the compiler projects annotations from the rich space of DLM security policies down to the much smaller space of

```
1  auto void makeGuess(Integer num) {
2    C?S?: int i = 0;
3    C?S?: if (num != null)
4    C?S?:   i = num.intValue();
5    C?Sh: boolean b1 = (i >= 1);
6          boolean b2;
7    C?Sh: if (b1) b2 = (i <= 10); else b2 = false;
8    C?Sh: if (b2) {
9      Sh:   boolean c1 = (tries > 0);
10           boolean c2;
11     Sh:   if (c1) c2 = (i == secret);
12     Sh:     else c2 = false;
13     Sh:   if (c2) {
14   C?Sh:     tries = 0;
15   C?S?:     finishApp("You win!");
16             } else {
17   C?Sh:     tries--;
18   C?S?:     if (tries > 0) {
19     C :       message.setText("Try again");
20             } else {
21   C?S?:       finishApp("Game over");
22             }
23           }
24         } else {
25     C :   message.setText("Out of range:"+i);
26         }
27 }
```

Figure 3.4: Guess-a-Number web application in WebIL

placement constraints.

Using the placement constraint annotations, the compiler chooses a partitioning of the WebIL code. A partitioning is an assignment of every statement and field to a host machine or machines on which the statement will execute, or the field be replicated. To optimize performance, partitioning uses an efficient algorithm based on a reduction to the maximum flow problem. A novel feature of WebIL is that code or data may be replicated in order to improve the performance of the application. The partitioned code is then translated into two Java programs, one to run on the server, and the other to run on the client.

Annotation	Possible placements	High integrity
`C`	{*client*}	N
`S`	{*server*}	N
`Sh`	{*server*}	Y
`CS`	{*both*}	N
`CSh`	{*both*}	Y
`CS?`	{*client*, *both*}	N
`C?S`	{*server*, *both*}	N
`C?Sh`	{*server*, *both*}	Y
`C?S?`	{*client*, *server*, *both*}	N

Table 3.1: WebIL placement constraint annotations

WebIL can be used as a source language in its own right, allowing programmers to develop web applications in a Java-like programming language with GUI support, while mostly ignoring issues of code and data placement, and client-server coordination. This approach has many benefits over traditional web application programming, but lacks the full security benefits of Swift.

3.5.1 Placement Annotations

Each statement and field declaration in WebIL is preceded immediately by one of nine possible placement annotations, shown in Table 3.1: `C`, `S`, `Sh`, `C?Sh`, `C?S?`, `CS`, `CS?`, `C?S`, and `CSh`. Placement annotations define the possible placements for each field or statement, as shown in the table. There are three possible placements: *client*, *server*, and *both*. The intuition is that `C` and `S` mean the statement or field must be placed on the client and server respectively, whereas `C?` and `S?` mean it is optional. An `h` signifies high integrity. Figure 3.4 shows the result of translating Guess-a-Number into WebIL, including placement con-

straints.

The placement of a field declaration indicates which host or hosts the field data stored is replicated onto. For example, if a field has the placement *server*, that field is stored only on the server; if it has the placement *both*, it is replicated on both client and server.

The placement of a statement indicates onto which host or hosts the computation of the statement is replicated. For compound statements such as conditionals and loops, the placement indicates the hosts for evaluating the test expression. On line 11 of Figure 3.4, the comparison of the guess to the secret number is given the annotation `Sh`, meaning that it must occur only on the server. Intuitively, this is the expected placement: the secret number cannot be sent to the client, so the comparison must occur on the server. On line 3, the annotation `C?S?` indicates that there is no constraint on where to test that `num` is non-null; that test may occur on the client, on the server, or on both.

For a statement that must execute on the server, the annotation may indicate that it is high-integrity. The annotations `Sh`, `C?Sh` and `CSh` denote high-integrity code. When translating to WebIL code, the Swift compiler will mark a statement as high-integrity if its execution may affect data that the client should not be able to influence. Thus, the client's ability to initiate execution of high-integrity statements must be restricted. As discussed in Section 3.6, run-time mechanisms prevent this.

Lines 5–14 of Figure 3.4 are annotated as high-integrity because the execution of these statements may alter or influence the values of the high-integrity variables `tries`, `b1`, `b2`, `c1`, and `c2`. Note that the start of the high-integrity

statements, line 5, corresponds to the start of the `endorse` statement of the original Jif program of Figure 3.2; it is due to this endorsement that the temporary local variables `b1`, `b2`, `c1`, and `c2` are regarded as high-integrity, and they therefore need to be protected from malicious clients. Note that the ability of the client to cause execution of these high-integrity statements comes from the `endorse` annotation at line 16 in the source, reflected in the WebIL code by the `auto` annotation on `makeGuess`.

3.5.2 Translation from Jif to WebIL

When the compiler translates from Jif to WebIL code, it replaces DLM security policies with corresponding placement constraint annotations, and translates Jif-specific language constructs into Java code. Based on the security policies of the Jif code, the compiler chooses annotations that ensure code and data are placed on the client only if the security of the program will not be violated by a malicious client.

In particular, the translation ensures that data may be placed on a client only if the security policies indicate that the data may be read by the principal `client`; data may originate from the client only if the security policies indicate that the data is permitted to be written by the principal `client`. Similar restrictions apply to code: code may execute on the client only if the execution of the code reveals only information that the principal `client` may learn; the result of a computation on the client can be used on the server only if the security policies indicate that the computation result is permitted to be written by the principal `client`.

The translation to WebIL also translates Jif-specific language features. Uses of the primitive Jif type `label` are translated to uses of a class `jif.lang.Label`. Declassifications and endorsements are removed, as they have no effect on the run-time behavior of the program. However, they do affect the labels of code and expressions, and therefore affect their placement annotations.

WebIL code is annotated at statement granularity. To allow fine-grained control over the placement of code, compound expressions are translated into a sequence of simple expressions whose results are stored in temporary local variables. Thus, subexpressions of the same source code expression may be computed on different hosts.

3.5.3 Goals and Constraints

The compiler decides the partitioning by choosing a placement for every field and statement of the WebIL program. Placements are chosen to satisfy both the placement constraints and also certain consistency requirements. Once placements are chosen, the WebIL program is split into two communicating programs, one running on the client, and the other running on the server. The goal of choosing placements is to optimize overall performance without harming security. Since network latency is typically the most significant component of web application run time, fields and statements are placed in order to minimize latency arising from messages sent between the client and server. For example, it is desirable to give consecutive statements the same placement.

Replicating computation can also reduce the number of messages. Consider

lines 5–8 of the Guess-a-Number application in Figure 3.4, which check that the user's input `i` is between 1 and 10 inclusive. To securely check that the client provides valid input, these statements must execute on the server. If the value entered by the user is not in the valid range, the server sends a message to the client to execute line 25, informing the user of the error. However, if lines 5–8 execute on *both* the client and server, no server–client message is needed, and the user interface is more responsive.

The placements of a field and of a statement that accesses the field must be consistent. In particular, if a statement writes to a field, then the statement and the field must have the same placement; if a statement reads a field, then the statement must be replicated on a subset of the hosts that the field is replicated on. These consistency requirements simplify the treatment of field accesses in the run-time system, ensuring that every replicated copy of a field is updated correctly, and that every read from a field occurs on a host on which the field is present. These requirements do not reduce the expressiveness of WebIL. Fields can be partitioned from their uses because a simple program transformation rewrites every field access as an assignment to or from a temporary local variable.

Figure 3.5 shows the `GuessANumber.makeGuess` method after partitioning. A placement has been chosen for each statement. The field `tries` has been replicated on both client and server, requiring all assignments to it to occur on both hosts (lines 14 and 17). Also, the compiler has replicated on both client and server the validation code to check that the user's guess is between 1 and 10 (lines 2–8). The validation code must be on the server for security, but placing it on the client allows the user to be informed of errors (on line 25) without

```
1  auto void makeGuess(Integer num) {
2     CS : int i = 0;
3     CS : if (num != null)
4     CS :   i = num.intValue();
5     CSh: boolean b1 = (i >= 1);
6          boolean b2;
7     CSh: if (b1) b2 = (i <= 10); else b2 = false;
8     CSh: if (b2) {
9      Sh:   boolean c1 = (tries > 0);
10           boolean c2;
11     Sh:   if (c1) c2 = (i == secret);
12     Sh:     else c2 = false;
13     Sh:   if (c2) {
14    CSh:     tries = 0;
15     S :     finishApp("You win!");
16            } else {
17    CSh:     tries--;
18    CS :     if (tries > 0) {
19     C :       message.setText("Try again");
20              } else {
21     S :       finishApp("Game over");
22              }
23           }
24         } else {
25     C :   message.setText("Out of range: "+i);
26         }
27 }
```

Figure 3.5: Guess-a-Number after partitioning

waiting for a server response.

3.5.4 Partitioning Algorithm

The compiler chooses placements for statements and fields in two stages. First, it constructs a weighted directed graph that approximates the control flow of the whole program. Each node in the graph is a statement, and weights on the graph edges are static approximations of the frequency of execution following

that edge. Second, the weighted directed graph and the annotations of the statements and field declarations are used to construct an instance of an integer programming problem, which is then reduced to an instance of the maximum flow problem. The solution for the integer programming problem directly yields the placements for fields and statements.

Cost Model. The key performance goal in Swift is to enhance responsiveness of the interactive web application UI. On a wide area network such as the web, where network latencies are high, optimal responsiveness is achieved by minimizing the number of round trip messages from the client to the server. Minimizing the size of the messages is not a concern, since network bandwidth on the web is usually cheap.

Swift places data and code statically. Since data transfer messages are piggybacked onto control transfer messages, it is enough to minimize the number of control transfers between the client and the server. If a statement S is statically placed on one host and many control transfers can be expected between S and the consecutive statement S', then S' must be placed on the same host as S. Fields are placed subsequently based on the placement of statements reading and writing to that field. Performing such a reasoning on the entire program requires building a control flow graph, computing the number of control transfers on each edge and finding an optimal partition. Unfortunately, the edge weights on the control flow graph and the optimal partition could be different for different combinations of the UI operation initiated by the user and the input values, and statically we have no information about which one of those combinations will happen at runtime.

To achieve a partition of the control flow graph practically, we need to compute a static approximation of the likely control transfers in the program over all UI operations and input values. Our cost model for this purpose is as follows:

- We assume each UI operation is equally likely to be initiated (presumably, a well designed simple UI would have this property)
- We assume each branch of a conditional statement is equally likely (unless a conditional statement is being used for error checking, this is approximately true)
- Each loop is assumed to iterate 10 times. The exact count is not important – as long as it is high enough, there is pressure to put the entire loop on the same host. If due to architectural reasons, the loop needs to be partitioned, the actual count does not matter since the total number of control transfers would be the same.

Control-flow graph. For each method in the program, a control-flow graph (CFG) is constructed, and, assuming that the method is invoked n times, non-negative, real weights are assigned to edges in the method's CFG. Edge weights are multipliers of n, representing how often that edge is taken. The multiplier of n is estimated using our cost model: each branch of an `if` statement is assumed to be taken the same number of times, and each loop is assumed to execute ten times before it exits. Exceptions, `break` and `continue` statements are ignored – they are assumed to be relatively infrequent, so ignoring them should not affect the solution much.

An interprocedural analysis is then performed to construct a call graph of the whole program. For dynamically dispatched methods, the analysis con-

servatively finds all possible method bodies that may be invoked. Recursive methods are ignored, so the resulting call graph is acyclic. Recursive methods tend to be uncommon in web applications. Consistent with our cost model, the application's `main` method and each UI event handler is assumed to be called exactly once, and the weights are propagated through the call graph using each method's CFG with edge weights. At method calls, every possible target is assumed to be invoked the same number of times, similar to conditionals. The result of the construction is a control flow graph of the entire program, with edge weights that approximate how often the edge is followed.

Integer programming problem. Using the weighted directed graph and placement constraint annotations on field declarations and statements, the placement problem is expressed as an instance of an integer programming (IP) problem. A solution to the problem assigns all variables in the problem a value in $\{0, 1\}$. Each statement u is associated with two variables, s_u and c_u. The variable s_u is 1 if the statement u is replicated on the server, and c_u is 1 if u is replicated on the client. For each u, the constraint $s_u + c_u \geq 1$ ensures that every statement has to be replicated somewhere. Also, linear constraints are used to ensure consistency in the annotations between statements that access the same field.

For each edge $e = (u, v)$ in the weighted directed graph, two variables x_e and y_e are used. The variable x_e is 1 if a message is sent from the client to the server when program execution transitions from statement u to statement v. This occurs when v executes on the server, but u does not, and therefore there is a constraint $x_e \geq s_v - s_u$. Similarly, y_e is 1 if a message is sent from the server to the client when program execution transitions on edge e; therefore, there is a

constraint $y_e \geq c_v - c_u$.

Let w_e be the weight of edge e. The goal is to find an assignment to all variables that satisfies all constraints, and minimizes the cost of the messages sent. This cost is $\sum_e w_e(x_e + y_e)$.

Although integer programming problems are in general NP-complete, this particular problem has the nice property that its linear relaxation (obtained by replacing the constraint $s_v, c_v, x_e, y_e \in \{0, 1\}$ with $s_v, c_v, x_e, y_e \geq 0$) always has an integral optimal solution. Therefore, placement is polynomial-time solvable, because an integral optimal solution to the linear relaxation is an optimal solution to the IP problem, and linear programming problems are polynomial-time solvable.

An efficient algorithm for the placement problem is designed by reducing the integer programming problem to an instance of the maximum flow problem. The key to the algorithm is the construction of the flow graph H on which maximum flow is computed. It is constructed from the weighted directed graph G that approximates the control flow. Using the preflow-push method [17], the algorithm runs in $O(V^3)$, where V is the number of statements. The algorithm also implements the gap heuristic [24], and achieves a satisfactory performance for compiling the test cases in the chapter.

The graph H is constructed from the graph G as follows. First, create G', which is a copy of G, with all edges reversed, i.e., for each edge $e = (u, v) \in G$, there is an edge (v', u') with weight w_e in G'. All nodes and edges of G and G' are in H. For each node u in G with corresponding node u' in G', add an edge (u, u') to H with infinite weight. Add two distinguished nodes to H, t_s and t_c,

representing the server side and the client side respectively. Finally, add edges with infinite weights for every statement u that has a known placement: if u is only on the server, add an edge (t_s, u); if it is only on the client, add an edge (u', t_c); if it is on both sides, add two edges (t_s, u') and (u, t_c).

According to the max-flow min-cut theorem [17], there is a maximum flow from t_s to t_c on H, and a minimum cut (S, C), where S and C are two disjoint sets that cover all nodes in H, and $t_s \in S$, $t_c \in C$; the cost of the maximum flow equals that of the minimum cut, and it gives the optimal solution $s^*_u, c^*_u, x^*_e, y^*_e$ to the integer programming problem:

$$s^*_u = \begin{cases} 1 & u' \in S \\ 0 & \text{otherwise} \end{cases} \qquad c^*_u = \begin{cases} 1 & u \in C \\ 0 & \text{otherwise} \end{cases}$$

$$x^*_{(u,v)} = \max\{0, s^*_v - s^*_u\}$$

$$y^*_{(u,v)} = \max\{0, c^*_v - c^*_u\}$$

The above solution is legal, because it disallows $s^*_u = c^*_u = 0$: if that were the case, it would imply that $u \in S$ and $u' \in C$, which is impossible as (u, u') has an infinite weight.

Of course, the accuracy of this approach is limited by how closely the weighted directed graph approximates actual run-time behavior. More sophisticated static analysis techniques or profiling data could yield more precise weighted directed graphs. However, in practice the current placements appear to be good.

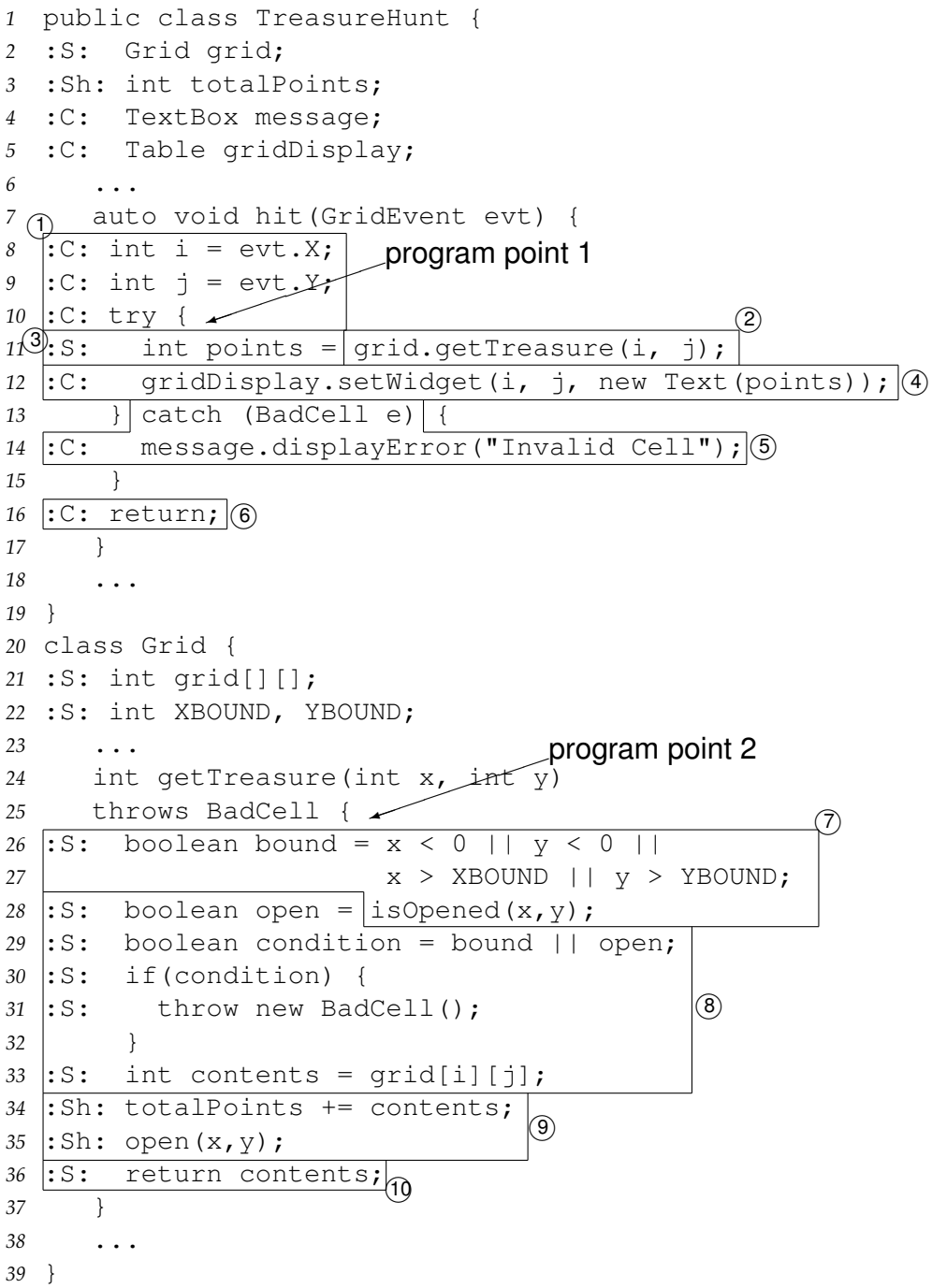

Figure 3.6: Part of the Treasure Hunt application, in WebIL

3.6 The Swift runtime

From a partitioning of a WebIL program, the Swift compiler produces two Java programs. One executes on the server, and the other on the client (after translation to JavaScript). Each statement and field declaration of the WebIL program is represented in one or both of these programs, according to its placement. Concurrent execution of these two programs simulates execution of the original Jif program while enforcing its security requirements.

Both programs rely on Swift's run-time support, which manages communication and synchronization. The client and the server have separate run-time systems, which are similar but not identical, since the trust model is asymmetric. The client's run-time system trusts all messages from the server, but the server does not trust any messages from the client.

This section describes the Swift run-time support and shows how WebIL code is translated into Java. It also explains how GWT is used to compile client-side code into JavaScript.

3.6.1 Execution Blocks and Closures

Methods in WebIL are divided into units called *execution blocks*, which are contiguous segments of code with the same placement annotation. Execution blocks have a single entry point and one or more exit points. Each execution block has a unique identifier. For example, Figure 3.7 shows the execution blocks of the Guess-a-number `makeGuess` method, in which blocks `block2` and `block3` have two exit points and the other blocks have just one. In gen-

eral, an execution block contains more than one basic block; a simple dataflow analysis finds execution blocks of maximal size.

Execution blocks are executed sequentially, following branches from each block to the next. Suppose a branch is taken from execution block *s* to execution block *t*. If *t* is to run on a host that did not run *s*, the other host invokes it by sending a message containing the identifier of *t*. We call this a *control transfer messages*, although the original host may also continue executing *t*. If *s* is placed on both hosts, no message need be sent.

An execution block runs in the context of an activation record, which stores the state of local variables and method arguments. For a given activation record, the client and server have distinct views, where each view stores only the variables used on that host. The two views share the same unique activation record identifier.

For example, Figure 3.6 shows two methods of another web application called "Treasure Hunt". This game has a secret grid in which some cells contain bombs and others contain treasure. The user explores the grid by digging up cells, exposing their contents. Figure 3.8 shows the state of the run-time system on the client and server during execution. The activation record for the method `Dig.hit` has variables `i`, `j`, and `points` in the client view but only the variable `points` in the server view.

When one host invokes an execution block on the other, it supplies both the identifier of the next execution block and the the identifier for the appropriate activation record. The pair of execution block and activation record identifiers is a *closure*, a self-contained executable unit.

The client and server run-time systems each maintain a stack of closures. Two kinds of closures are kept on the stack: *exception handler closures* and *return closures*. A return closure is pushed onto the stack when a method is called, and popped when it returns. Exception handler closures are pushed at the entry to a `try...catch` block, and popped off at the exit. They are invoked if an exception is thrown within the block.

Figure 3.8 shows the state of the client stack at program points 1 and 2 in Figure 3.6.

At program point 1, execution of the `try` statement at line 30 has just pushed the handler for the `BadCell` exception onto the stack. The handler has three fields: the type of exception it handles (in this case, `BadCell`), the execution block identifier, and the activation record identifier of the current call to `Dig.hit`. If a `BadCell` exception is thrown within the call to `getTreasure`, the run-time system walks up the stack, finds this exception handler closure, and executes it.

At program point 2, the `Grid.getTreasure` method has just begun executing. The return closure for the caller (`hit`) is on the top of the stack. It points to the execution block (3) right after the method call returns, and the activation record for the latest invocation of `hit`.

On the server side, the stack of closures serves a second, crucial function: it enforces the integrity of control flow. As discussed in Section 3.6.4, closures may also be pushed onto the stack to ensure that control flow passes through them.

The closure stacks on the client and server are synchronized by piggybacking stack updates onto control transfer messages. A stack update contains a set of

```
1  // auto void makeGuess(Integer num)
2  block1: (CS)
3    int i = 0;
4    if (num != null) i = num.intValue();
5    goto block2;
6  block2: (CSh)
7    boolean b1 = (i >= 1);
8    boolean b2;
9    if (b1) b2 = (i <= 10); else b2 = false;
10   if (b2) goto block3; else goto block10;
11 block3: (Sh)
12   boolean c1 = (tries > 0);
13   boolean c2;
14   if (c1) c2 = (i == secret); else c2 = false;
15   if (c2) goto block4; else goto block6;
16 ...
17 block10: (C)
18   call message.setText("Out of range: "+i);
```

Figure 3.7: Guess-a-Number execution blocks

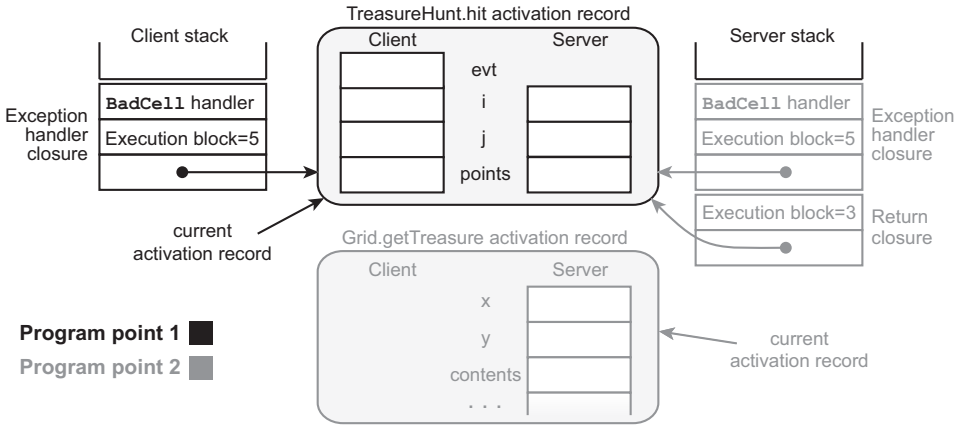

Figure 3.8: Run-time state at program points 1 and 2 in Figure 3.6

new closures and an update depth. A stack update is applied by popping off existing closures to the specified depth, and then pushing on the new closures.

Returning to the example in Figure 3.8, when control goes from the client to the server at program point 1, the stack update consists of the `BadCell` handler

and any closures above it that were created on the client since the last control transfer from the server. After the update is applied on the server, the handler appears on the server stack, as shown in gray in Figure 3.8.

Stack updates also include updates to the activation records that new closures refer to. Only variables that that other host should receive are sent, which is important for security. The client does not receive updates to confidential variables, and the server does not accept updates to high-integrity variables.

3.6.2 Closure Results

When a closure *s* runs, it produces a *result* that includes the closure *t* to run next. A closure may have one of four kinds of results: a *simple result*, an *exception result*, a *method call result*, or a *method return result*. A simple result identifies a closure *t* within the same method as the closure *s* that returned it, in which case *s* and *t* share the same activation record. The run-time system simply invokes *t*, sending a message to the other host if needed.

To invoke a method, a closure returns a method call result, which contains a reference to the receiver object, the method identifier, and a return closure. The runtime system pushes the return closure onto the closure stack, creates a new activation record with the argument values, and using the run-time class of the receiver object, and dynamically dispatches to the execution block that implements the method.

Figure 3.8 shows this sequence of steps. Figure elements in dark black show the state at program point 1 on the client, before the invocation of

`Grid.getTreasure`. Executing until program point 2, just before the first execution blockin `getTreasure` begins, adds the elements in gray, in the following sequence of steps:

1. A stack update containing the `BadCell` handler closure is sent by the client and applied at the server, piggybacked onto a control transfer message that invokes execution block 2.
2. The return closure for `Dig.hit` is pushed onto the stack.
3. The run-time type of the receiver object is used to determine the execution block (7) to run.
4. A new activation record for the `getTreasure` method is created with some identifier *id*, and filled in with values for `x` and `y`.
5. The current activation record pointer is updated to point to *id*
6. The closure $\langle t, id \rangle$ is executed by the runtime.

When a method return result or an exception result is produced by a closure, the run-time system walks up the stack to find the first return closure or the first matching handler closure respectively to execute. In either case, the result contains a value (the value returned or thrown) which is passed to the closure found.

3.6.3 Classes and Objects

A Jif class C is translated into two classes: C_s for use by the server Java program, and C_c for the client Java program. For each field f of class C, the placement of f

determines whether the field declaration should be placed in C_s or C_c (or both). Each object has a unique object identifier. An object o of class C is represented by a pair of objects o_s and o_c, where o_s of class C_s is on the server, o_c of class C_c is on the client, and o_s and o_c share the same object identifier.

When an object reference is sent from one machine to the other (for example, when forwarding the value of a local variable), it suffices to send the object identifier. If the receiving machine is not aware of the object identifier, a *heap update* is also sent, informing the receiving machine of the runtime class of the object; the receiving machine's runtime system will create an object of the appropriate class with the specified object identifier.

Label checking on the original Jif source program ensures that heap updates do not violate confidentiality of information: if the server needs to send a heap update to the client for a particular object, then the client is permitted to know about the existence of that object. Conversely, before applying a heap update received from the client, the server checks it for consistency; for example, it checks uniqueness of the object identifier. Fields of an object will never be read before they are initialized.

3.6.4 Integrity of Control Flow

A *high integrity closure* has an execution block which has high-integrity side effects, and is therefore annotated `Sh`. A misbehaving client might try to send a control transfer message specifying a high-integrity execution block, and thereby compromise the integrity of variables affected by that execution block. A simple-minded approach would be to prevent the client from invoking high-

integrity closures. However, in some situations, the client should be allowed to invoke a high integrity closure on the server. Consider the following WebIL code, after partitioning:

```
1  Sh: this.f = 7;
2  C : this.g = 8;
3  Sh: m(this.f);
```

Lines 1 and 3 are both high-integrity execution blocks, but line 2 must execute on the client. Thus, correct control flow of the program requires the client to invoke the high-integrity closure for line 3.

To control how the client invokes high-integrity closures, high-integrity closures are pushed onto the closure stack. A client may invoke a high-integrity closure only if it is at the top of the closure stack. For example, the execution of line 1 pushes a closure for line 3 onto the closure stack, which allows the client execution block at line 2 to invoke line 3, but no other high-integrity closure. Further, a client cannot pop a high-integrity closure without executing it. The server checks that closure invocations and closure stack updates from the client obey these rules. As a result, the client has no way to control the execution of high-integrity closures.

A dataflow analysis is used to statically determine when high integrity closures should be pushed onto the closure stack. When control flow may pass from a low-integrity execution block u to a high-integrity execution block t, the analysis finds the high-integrity execution blocks s that immediately precedes the low-integrity execution leading to u. The execution of s then pushes the closure for t onto the closure stack. Because the WebIL code was generated from

a Jif program with secure information flows, a suitable execution block s exists for each such u and t.

3.6.5 Other Security Considerations

The fact that WebIL programs are generated from Jif programs with secure information flows is important to ensuring translated code is secure. For example, the client does not learn any secret information by knowing which closures the server requests the client to execute. Static checking of the Jif program prevents these *implicit flows* [23] (covert storage channels arising from program control structure). Similarly, stack updates, activation record updates and heap updates do not leak information covertly.

Care must also be taken in the runtime system to ensure that no new information channels are introduced in translated code. In particular, the unique identifiers used for activation records and objects form a potential information channel. If the identifiers of objects and activation records follow a predictable sequence, and confidential information may affect the number of objects or activation records created on the server, then the client may be able to infer confidential information based on the object and activation record identifiers it sees.

To ensure that object and activation record identifiers do not reveal confidential information, a cryptographic hash function is used to generate unpredictable identifiers for computation in server-only closures. Thus, sending the identifier of an object or activation record to the client does not reveal any confidential details of the server's execution history.

3.6.6 Concurrency Issues

A sequential Swift program can still translate to concurrent WebIL code, since some code can be replicated on the client and the server, meant to execute in parallel. In some situations, the client could get ahead of the server and end up sending a second request to the server, before the response from the first request arrives. To avoid such race conditions, the Swift client run-time ensures that there is only one outstanding request to the server at any point in time, by using a queue of requests. A mis-behaving client can still cause race conditions in this manner — however, data confidentiality and integrity is never affected. The analogous situation does not arise on the server, since the server only responds to client requests — it never initiates a request to the client.

Much of the discussion of Swift has implicitly assumed a single client and a single server. Swift, however, does support multiple clients communicating with the same server, by simply leveraging the relevant machinery from the Java Servlet framework. To support multiple clients, the language is restricted to disallow specifying a security label containing the `client` keyword, on a static field, since it would be ambiguous which client is being referred to.

3.6.7 GWT and Ajax

We use the Google Web Toolkit [31] (GWT) compiler and framework to translate the client Java programs (and the Swift client runtime system) to JavaScript. GWT provides browser-independent support for Ajax and JavaScript user interfaces. This implementation choice facilitates the development of the Swift runtime system and compiler, but is not fundamental to the design of Swift.

Example	Jif	Java target code		JavaScript		
		Server	Client	All	Framework	App
Null program	6 lines	0.7k tokens	0.6k tokens	73 kB	70 kB	3 kB
Guess-a-Number	142 lines	12k tokens	25k tokens	267 kB	104 kB	162 kB
Shop	1094 lines	139k tokens	187k tokens	1.21 MB	323 kB	889 kB
Poll	113 lines	8k tokens	17k tokens	242 kB	104 kB	137 kB
Secret Keeper	324 lines	38k tokens	38k tokens	639 kB	332 kB	307 kB
Treasure Hunt	92 lines	11k tokens	11k tokens	211 kB	99 kB	112 kB
Auction	502 lines	46k tokens	77k tokens	503 kB	116 kB	387 kB

Table 3.2: Code size of example applications

Ajax permits an elegant implementation of our runtime protocol. Communication between client and server occurs mostly invisibly to the user. The Swift server runtime system implements a service interface that accepts requests for closure invocations. GWT automatically generates asynchronous proxies that the client can access, and provides marshaling of data sent over the network.

The Ajax model has an inherent asymmetry: only the client is able to initiate a dialogue with the server. Any message sent from the server to the client (such as a request to invoke a closure) must be a response to a previous client request. With minor modifications to our runtime system, we can ensure that whenever the server needs to send a message to the client, the client has an outstanding request.

3.7 Evaluation

The Swift compiler extends the Jif compiler with about 20,000 lines of non-comment non-blank lines of Java code. Both the Swift and Jif compilers are written using the Polyglot compiler framework [63]. The Swift server and client

Example	Task	Actual		Optimal	
		S→C	C→S	S→C	C→S
Guess-a-Number	guessing a number	1	2	1	1
Shop	adding an item	0	0	0	0
Poll	casting a vote	1	1	0	1
Secret Keeper	viewing the secret	1	1	1	1
Treasure Hunt	exploring a cell	1	2	1	1
Auction	bidding	1	1	1	1
Key: S→C=Server to Client message, C→S=Client to Server message					

Table 3.3: Network messages required to perform a core UI task

run-time systems together comprise about 2,600 lines of Java code. The UI framework is implemented in 1,400 lines of WebIL code and an additional 560 lines of Java code that adapt the GWT UI library. We also ported the Jif run-time system from Java to WebIL, resulting in about 3,900 lines of WebIL code. The Jif run-time system provides support for run-time representations of labels and principals.

Although Swift shares some ideas and techniques with previous work on Jif/split [106], no compiler or run-time code was reused from Jif/split, because of significant differences between the systems. These differences include a richer source language, use of the intermediate language WebIL, simplified protocols for field access and control transfer, and the optimization of partitioning.

To evaluate our system, we implemented six web applications with varying characteristics. None of these applications is large, but because they test the functionality of Swift in different ways, they suggest that Swift will work for a wide variety of web applications. Because the applications are written in a higher-level language than is usual for web applications, they provide much

functionality (and contain many security issues) per line of code. Overall, the performance of these web applications is comparable to what can be obtained by writing applications by hand.

3.7.1 Example Web Applications

Guess-a-Number. This running example demonstrates how Swift uses replication to avoid round-trip communication between client and server. Figure 3.5, lines 5–8, show that the compiler automatically replicates the range check onto the client and server, thus saving a network message from the server to the client at line 25. Potential insecurities are also avoided by automatically placing the `tries` field on the server so a malicious client cannot corrupt it, and by placing `secret` on the server where it cannot be leaked or corrupted.

Shop. This program models an important class of real-world web applications, and is the largest Swift program written to date. It is an online shopping application with a back-end PostgreSQL database. Items may be added to and removed from a shopping cart (automatically updating the total cost), orders can be placed, and users can update their billing information. Users must log in before shopping; new users can register themselves. The database contains both confidential authentication and billing information for each user, and high-integrity inventory information.

Poll. This application is an online poll that allows users to vote for one of three options and view the current winner. Server-side static fields are used

to provide persistence and sharing across multi-user Swift applications. The current count for each choice is kept as a secret on the server, and an explicit declassification makes the result available to users who request to see it.

Secret Keeper. This simple application allows users to store a secret on the server and retrieve the secret later by logging in. In the source program, the secret of a user has a strong confidentiality policy that only allows that user principal to read it. Once the user logs in, the acts-for relationship established between the client and the user principal permits the secret to be released securely without declassification. This example shows that Swift can handle complex policies with application-defined principals, and that it can automatically generate protocols for password-based authentication and authorization from high-level information security policies.

Treasure Hunt. This game is described in Section 3.6.1. It has a relatively rich user interface that is dynamically and incrementally updated as the user discovers what lies beneath cells in the secret grid. Because the grid is secret, it is placed on the server and accessed via Ajax calls as it is explored.

Auction. This online auction application allows users to list items for sale and bid on items from other users. Once a seller starts an auction, it is visible to other users, and the current bid as well as the bidder's username is shown. The application automatically polls the server to retrieve auction status updates and updates the display. Buyers can enter higher bids until the seller ends the auction. Information about each auction is considered public to users but is maintained with high integrity on the server.

3.7.2 Code Size Results

Table 3.2 shows the code size of the example applications and the generated target code. Generated code size is reported in non-comment tokens rather than in lines, as line counts are not meaningful. However, as a point of comparison, the Jif source programs use 9–11 tokens per line. The "Java target code" columns report the size of the Java output for the server and client. Note that this does not include the Swift run-time systems, nor the UI framework and Jif runtime. (Recall that the UI framework and Jif runtime are both implemented in WebIL.) The "JavaScript All" column reports the size of the code generated by GWT compiling the client Java target code, including the parts of the UI framework and Jif runtime that are partitioned onto the client, and the Swift client runtime; the "JavaScript Framework" column gives the size of code produced by using GWT to compile just the Swift client runtime and the parts of the UI framework and Jif runtime placed on the client. The difference, in the "JavaScript App" column, indicates how much JavaScript code is specific to the application.

The size of the application JavaScript code is approximately linear in the size of the Jif source. For these applications, about 800 bytes of JavaScript is generated per line of application Jif code. Much of the expansion occurs when Java code is compiled to JavaScript by GWT, so translating WebIL directly to JavaScript might reduce code size.

3.7.3 Performance Results

We studied the performance of the example applications from the user's perspective. We expect network latency to be the primary factor affecting appli-

cation responsiveness, so we measured the number of network round trips required to carry out the core user interface task in each application. For example, the core user interface task in Guess-a-Number is submitting a guess. We also compared the number of actual round trips to the optimum that could be achieved by writing a secure web application by hand.

Table 3.3 gives the number of round trips required for each of the applications. To count the number of round trips, we measure the number of messages sent from the server to the client. These messages are the important measure of responsiveness because it is these messages that the client waits for. The table also reports the number of messages sent from the client to the server. Because the client does not block when these messages are sent, the number of messages from client to server is not important for responsiveness.

The total number of round trips in the example applications is always optimal or nearly so. For example, in the Shop application, it is possible to update the shopping cart without any client–server communication. The optimum number of round trips is not achieved for Poll because the structure of Swift applications currently requires that the client hear a response to its vote request. For Guess-a-Number and Treasure Hunt, there are extra client–server messages triggering server-side computations that the client does not wait for, but server–client messages remain optimal.

3.7.4 Automatic Repartitioning

One advantage of Swift is that the compiler can repartition the application when security policies change. We tested this feature with the Guess-a-Number ex-

ample: if the number to guess is no longer required to be secret, the field that stores the number and the code that manipulates it can be replicated to the client for better responsiveness. Lines 9–13 of Figure 3.5 all become replicated on both server and client, and the message for the transition from line 13 to 14 is no longer needed. The only source-code change is to replace the label `{⋆→⋆; ⋆←⋆}` with `{⋆→client; ⋆←⋆}` on line 3 of Figure 3.2. Everything else follows automatically.

3.8 Related work

In recent years there have been a number of attempts to improve web application security. At the same time, there has been increasing interest in unified frameworks for web application development. The goals of these two lines of work are in tension, since moving code to the client affects security. Because it provides a unified programming framework that enforces end-to-end information security policies, Swift is at the confluence of these two lines of work.

3.8.1 Information Flow in Web Applications

Several previous systems have used information flow control to enforce web application security. This prior work is mostly concerned with tracking information integrity, rather than confidentiality, with the goal of preventing the client from subverting the application by providing bad information (e.g., that might be used in an SQL query). Some of these systems use static program analysis (of information flow and other program properties) [34, 95, 35], and some use

dynamic taint tracking [32, 62, 96], which usually has the weakness that the untrusted client can influence control flow. Concurrent work uses a combination of static and dynamic information flow tracking and enforces both confidentiality and integrity policies [12]. Unlike Swift, none of this prior work addresses client-side computation or helps decide which information and computation can be securely placed on the client. Most of the prior work (except [12]) only controls information flows arising from a single client request, and not information flow arising across multiple client actions or across sessions.

Instrumenting JavaScript with dynamic security checks [100] has been proposed to protect sensitive client information from cross-site scripting attacks and similar vulnerabilities. In these attacks, a malicious website attempts to retrieve information from another browser window or session to which it should not have access. The usual avenue of attack is via JavaScript's ability to interpret and execute user-provided input as unchecked code, using the `eval` operation. Because Swift does not expose these "higher-order scripting" capabilities of JavaScript, it is not vulnerable to these attacks.

3.8.2 Uniform Web Application Development

Several recently proposed languages provide a unified programming model for implementing applications that span the multiple tiers found in web applications. However, none of these languages helps the user automatically satisfy security requirements, nor do they support replication for improved interactive performance.

Links [16] and Hop [76] are functional languages for writing web applica-

tions. Both allow code to be marked as client-side code, causing it to be translated to JavaScript. Links does this at the coarse granularity of individual functions, whereas Hop allows individual expressions to be partitioned. Links supports partitioning program code into SQL database queries, whereas Hop and Swift do not. Swift does not have language support for database manipulation, though a back-end database can be made accessible by wrapping it with a Jif signature. To keep server resource consumption low, Links stores all state on the client, which may create security vulnerabilities. Neither Links nor Hop helps the programmer decide how to partition code securely.

Hilda [99, 98] is a high-level declarative language for developing data-driven web applications. The most recent version [98] also supports automatic partitioning with performance optimization based on linear programming. Hilda does not support or enforce security policies, or replicate code or data. Hilda's programming model is based on SQL and is only suitable for data-driven applications, as opposed to Swift's more general Java-based programming model. Swift partitions programs on a much finer granularity than on Hilda's "Application Units", which are roughly comparable to classes; fine-grained partitioning is critical to resolve the tension between security and performance. The performance optimization problem in Hilda is NP-complete, and is solved with a bicriteria approximation algorithm, while Swift has a problem that is solvable in polynomial time, and an efficient algorithm is presented.

A number of popular web application development environments make web application development easier by allowing a higher-level language to be embedded into HTML code. For example, JSP [9] embeds Java code, and PHP [65] and Ruby on Rails [89] embed their respective languages. None of

these systems help to manage code placement, or help to decide when client-server communication is secure, or provide fully interactive user interfaces (unless JavaScript code is used directly). Programming is still awkward, and reasoning about security is challenging.

The Google Web Toolkit [31] makes construction of client-side code easier by compiling Java to JavaScript, and provides a clean interface for Ajax requests. However, GWT neither unifies programming across the client–server boundary, nor addresses security.

3.8.3 Security by Construction

An important aspect of Swift is that it provides security by construction: the programmer specifies security requirements, and the system transforms the program to ensure that these requirements are met. Prior work has explored this idea in other contexts.

The Jif/split system [103, 106] also uses Jif as a source language and transforms programs by placing code and data onto sets of hosts in accordance with the labels in the source code. Jif/split addresses the general problem of distributed computation in a system incorporating mutual distrust and arbitrary host trust relationships. Swift differs in exploring the challenges and opportunities of web applications. Web applications have a specialized trust model, and therefore specialized construction techniques are used to exploit this trust relationship. In particular, replication is used by Jif/split to boost integrity, whereas Swift uses replication to improve performance and responsiveness. In addition, Swift uses a more sophisticated algorithm to determine the placement and repli-

cation of code and data to the available hosts. Swift applications support dynamic user interfaces (represented as complex, compositional data structures) and control the information flows that result. No Jif/split applications contain data structures or control flow of comparable complexity. Jif's label parameterization is needed to reason about information flow in complex data structures, as in Figure 3.3, but Jif/split lacks the necessary support for label parameters.

Program transformation has also been applied to implementing secure function evaluation in a distributed system, in Fairplay [51]. Its compiler translates a two-party secure function specified in a high-level language into a Boolean circuit. Fairplay provides strong, precise security guarantees for simple computations, but does not scale to general programs. However, its techniques might be applicable within a larger framework such as Swift.

3.9 Conclusions

We have shown that it is possible to build web applications that enforce security by construction, resulting in greater security assurance. Further, Swift automatically takes care of some awkward tasks: partitioning application functionality across the client–server boundary, and designing protocols for exchanging information.

Writing Swift code does require writing security label annotations. These annotations are mostly found on method declarations, where they augment the information specified in existing type annotations. In our experience, the annotation burden is clearly less than the current burden of managing client–server communication explicitly, even ignoring the effort that should be expended on

manually reasoning about security. More sophisticated type inference algorithms might further lessen the annotation burden, but we leave this to future work.

Swift satisfies three important goals: enforcement of information security; a dynamic, responsive user interface; and a uniform, general-purpose programming model. No prior system delivers these capabilities. Because web applications are being used for so many important purposes by so many users, better methods are needed for building them securely. Swift appears to be a promising solution to this important problem.

CHAPTER 4

READ CHANNELS

We have argued about the advantages of raising the level of abstraction at which the programmer develops a system. However, reasoning about the system at a more abstract level can lose sight of covert channels [38]. A covert channel is an information channel that is not intended for information transfer but can still leak secret information. For instance, a program can leak one bit of secret information by choosing to either hold or not hold a lock on a shared resource. Other processes can learn this bit by requesting a hold on the lock and checking to see if the request succeeded.

In this chapter, we focus on a particular kind of covert channel: the distributed read channel [54], also simply called the *read channel* [102]. A read channel is a covert communication channel that often occurs in distributed systems and involves a trusted host fetching *public* data from an untrusted host during a sensitive computation. Secret information can be leaked via the *pattern* of data fetches, rather than through the data fetched.

For example, imagine a credit score mashup application A credit score is computed for a user and the browser displays whether or not it is a good score, using images from an untrusted third-party server. If the score is less than 750, it displays a thumbs-down, otherwise it displays a thumbs-up. A simple fetch of public image data might seem benign. However, based on which image is fetched, the untrusted server can learn whether the user has a good credit score, thereby violating site security policy.

As individuals and organizations are moving their activities online, their data and code is increasingly interacting with untrusted hosts. As a result, applications on the web are increasingly becoming vulnerable to read channels. Developing techniques for addressing read channels is important. This chapter first describes the problem through its manifestation in the Fabric system, and then places the problem in the context of related work. Section 4.2 presents an extension of the Fabric type system that rejects programs with read channels. Section 4.3 discusses the limitations of the type systems approach and presents a program transformation algorithm based on abstract interpretation that automatically eliminates read channels in a given program. Section 4.3.4 evaluates the performance overhead incurred by using the program transformation technique.

4.1 Problem Definition

The problem we address is that of constructing distributed systems in a way that provides assurance that it is not vulnerable to leakage of sensitive information through read channels. We develop a solution based on existing work on language based security [61]. In this setting, a read channel is defined as a fetch of public data from a low host, done within a high-pc context. The read channel is thus an implicit flow of high information to a low host. The extensions of Jif [61] we considered in Chapter 3 were not vulnerable to the read channel problem. Swift statically and conservatively places data objects in a way that read channels never occur. In SIF-Fabric, there were no messages exchanged between mutually distrusting domains, and so the threat of read channels does not arise. However, in the general Fabric [44] system read channels are an issue

```
1  if (h) {
2           y = o.f;
...
6  } else {
7           y = r.f;
...
13 }
```

Figure 4.1: An example of a read channel in Fabric

since data objects are placed dynamically. As a result, we need extra mechanisms to ensure that objects are placed only on hosts that are allowed to view all the fetches of those objects.

4.1.1 Read Channels in Fabric

Consider Figure 4.1 showing code fragment from an application written using Fabric (see Section 3.1.1 for a description of the Fabric system).

Let us assume that this code is executing on a worker W and that the objects referred to by o and r are instances of the same class and are kept on the same store S, such that W and S do not necessarily trust each other. Furthermore, let us assume that the security label on the boolean variable h is L_h, such that $L_h \not\sqsubseteq \{\top \rightarrow S\}$. In other words, S is not allowed to learn information about h. Also, since h is being used in code running on W, we have $L_h \sqsubseteq \{\top \rightarrow W\}$, indicating that W is allowed to learn the value of h.

The dereference of o on line 2 and of r on line 7 requires fetching the corresponding objects from S. Since the *pc* label at these points is L_h, each object fetch (acting as a side effect) leaks information at L_h to S. As per the constraints on L_h

in the last paragraph, such a leak is not allowed. Intuitively, S learns the value of h depending on whether o or r was requested. This particular kind of covert information channel is called a read channel.

4.1.2 Related Work

Previous work has addressed problems very similar to the read channel problem just described. Private Information Retrieval (PIR) [13] addresses the problem of protecting the privacy of users while they are querying a publicly accessible database. For the purpose of this discussion, assume that the database is simply a bit string of length n and a user's query is an integer i such that $1 \leq i \leq n$. In other words, the user seeks to find the value of the i^{th} bit in the database. The threat model is that a curious database operator can track a user's queries and get a sense of the user's intent, which is meant to be kept secret. The goal of PIR is to execute a given query in a way that the user learns the value of the i^{th} bit without the database knowing the value of i. In the case that information theoretic secrecy is desired and only a single copy of the database exists, the entire database needs to be downloaded by the user before executing the query. The communication complexity in this case is $O(n)$ where n is the size of the database. If multiple replicas of the database are available, and the hosts maintaining those replicas do not collude, the user can cleverly code his requests to each database separately in a way that the responses can be used to infer the answer to his query. Each database individually sees a request for a random subset of data elements, which may or may not contain a request for the element that the user is looking for. This way, the database does not learn anything about the user's query. With replicated databases, the communication complexity can

always be sublinear; e.g. for the two-replica case, a communication complexity of $O(n^{1/3})$ is possible [13].

Simultaneously, it was shown that a sublinear communication complexity can be achieved without replicating the database [36]. This method, however, assumes that the adversary (database) is computationally bounded and relies on the quadratic residuosity assumption, i.e. the computational hardness of testing for quadratic residues [53].

A related but different problem is that of designing an oblivious machine, motivated by the problem of protecting software from illegitimate duplication [30]. The solution to the problem of software protection is to distribute a software-hardware package (instead of only software) consisting of an encrypted program and a physically shielded CPU storing the decryption key. The shielded CPU is installed in a conventional computer system by connecting it to the address and data buses and the encrypted program is loaded into one of the memory devices. The program is executed by fetching the next encrypted instruction and having the CPU decrypt the instruction and execute it. Memory I/O is done by decrypting after reading and encrypting the data before writing. The physical shielding of the CPU is meant to make it tamper resistant in a way that the encryption key is lost if the CPU is tampered with (similar to smart cards). Thus, the encryption key never leaves the CPU. The above scheme, however, does not fully protect the program against reverse engineering by a determined adversary (even if we assume that the physical shielding works as intended). In particular, by reading the list of memory *addresses* that the program accesses, the adversary can infer "essential properties" of the program that could not have been inferred from only the specification of

the software [30]. Preventing information leakage through the sequence of fetch addresses can be done by making the CPU fetch the same sequence of addresses, independent of the program input – this is called making the CPU *oblivious* [67].

Designing an *efficient* oblivious machine for a given random access machine (RAM) can be done only by interpreting obliviousness in a probabilistic manner and assuming that the CPU has access to a random oracle [30]. Obliviousness, interpreted probabilistically, means that the *probability distribution* of the sequence of fetch addresses is independent of the program input. A given RAM can be simulated by a probabilistic oblivious RAM with a polylogarithmic overhead in both time and memory space. On the other hand, a given one-tape Turing machine can be transformed into an equivalent oblivious *deterministic* two-tape Turing machine [67]. In this case, the time overhead is logarithmic. A lower overhead is possible in the case of a Turing machine since the head movement is always local, in contrast to a random access machine in which the next memory address accessed could be anywhere in memory. A more restrictive head movement limits the range of possible fetch address sequences for a Turing machine and thus makes it easier to hide access patterns. Even though the asymptotic order of complexity of the overhead seems reasonable, the constants involved are too high for oblivious RAMs to be of any practical value.

However, recent developments [66, 83, 78, 81, 50, 49, 84, 46, 20] have brought ORAMs closer to being practically realizable. Overheads between 35X and 100X have become fairly standard. The overhead of using our program transformation technique is comparable and often better as shown in our results. However, we must note that our technique is qualitatively different from current work on ORAMs. ORAMs use a probabilistic notion of secrecy and make standard

cryptographic assumptions whereas we aim for perfect information theoretic secrecy. ORAMs typically optimize for both bandwidth and latency, whereas we assume a web like network where bandwidth is cheap and latency is optimized for. Concurrency is also not adequately handled in ORAMs. The threat models are also subtly different for ORAMs and read channels in Fabric. ORAMs assume that the attacker is computationally bounded and can view *all* reads and writes to the memory (including reads and writes of secret encrypted data). Fabric, on the other hand, makes no assumptions about the attacker's computational resources, but does assume that the attacker cannot view reads or writes of secret data. Despite these differences that make the work on ORAMs somewhat orthogonal, they are still useful as an extension to our prefetching technique. For instance, instead of prefetching all the objects necessary for executing a sensitive code block safely, we can use an ORAM that contains those objects and use an ORAM protocol for fetching from it as necessary.

Information leaks through memory access patterns is also a concern in system architectures with a cache shared between mutually distrusting applications. Two common examples of such an architecture are modern personal computers installed with a multi-user operating system and cloud computing platforms that instantiate, and sell to different clients, multiple virtual machines running on the same physical infrastructure. In such systems, sharing the cache across multiple applications enhances performance [55]. However, it introduces security problems. One of the applications can learn information about another application's memory access patterns by issuing fetch requests for various memory addresses and measuring the response time for each request. A quick response would very likely mean that the memory address was already in cache and that a fetch request for it was issued by the other application. Learning the

memory access pattern of an application can be problematic if, for example, the application is performing an AES encryption. The entire encryption key can be learnt without knowledge of the plaintext or the ciphertext [64]. Cloud computing platforms are somewhat resistant to the fine-grained information leaks required to steal cryptographic keys. However, the amount of cache usage (cache load) itself is enough to mount a keystroke timing attack from one VM to another so as to steal passwords entered through the keyboard [71] in the other VM. There has also been a lot of recent work on adapting oblivious RAMs to the cloud [82], targeting other applications that are vulnerable to read channels: behavioral advertising [5], location and map services, web search, etc.

These previous works indicate that the problem of information leaks via the pattern of remote fetches is an important and relevant, yet unsolved problem. This work addresses the problem by introducing development methodologies that provide assurance that a given distributed program does not have read channels, in line with the philosophy of security by construction. In contrast to PIR, the goal of this work is not to hide *all* memory (database) accesses, but to enforce the security policies associated with data values in the distributed program. For instance, going back to the example in Section 4.1.1, the fetch of o is itself not so much the problem as fetching o inside a high-pc context. The solution would not be to fetch all the objects stored on S (that still hides the fact that o was being dereferenced, as in PIR protocols) but to prefetch into the local cache objects referred to by o and r just before the start of the `if` statement (that hides the value of h from S as required by the policy). In general, all objects that could *possibly* be read in a block of code with a high pc need to be prefetched before entering the code block. Section 4.3 discusses how the set of such objects can be computed using an abstract interpretation. PIR protocols could still

be exploited by altering our notion of the database i.e. assume that the set of all possible objects read by a high-pc block is the entire database and that each dereference in a particular run would correspond to a query. For instance, if the possible objects are present on one or more dissemination nodes, the PIR protocol for replicated databases can be used to significantly reduce communication overhead. These enhancements, however, rely on the non-collusion of dissemination nodes and stores and is orthogonal to this work.

This work also focuses on perfect information theoretic secrecy, in contrast to computational PIR schemes and oblivious RAMs, which focus on computational indistinguishability and probabilistic secrecy respectively. In contrast to oblivious Turing Machines, this work provides a more practical methodology for avoiding read channels in programs written in a general purpose language. The problem of cache side channels can potentially be addressed using this work. Each application can be developed using the development methodology presented here, so that its cache access patterns do not reveal sensitive information. Evaluating the efficacy of this approach is left to future work.

4.2 A Type System for Controlling Read Channels

In this section, we present a type system that checks whether a Fabric program has a read channel. We focus on object dereferences, since they are the only program points which need to fetch remote data (from stores or dissemination nodes). For instance, referring to the code fragment in Section 4.1.1, the program points of interest are lines 2 and 7 (assuming there are no object dereferences elsewhere). Since we are interested in perfect secrecy, we can prevent

information leaks through the pattern of object fetches by ensuring that there is no information leak from each of the object fetches.

4.2.1 Threat Model

Similar to previous work on language-based security, the security of the system is characterized relative to an adversary A. The threat model is that the adversary has complete knowledge of the source programs running on all the hosts and all the data that is labeled l such that $l \sqsubseteq \{\top \rightarrow A\}$. The public visibility of all source code is assumed in order to avoid the pitfalls of security by obscurity. In addition, A controls all hosts H such that A actsfor H. An adversary can attempt to learn secret information by simulating a run of the program and following the sequence of object fetches performed by the actual execution to infer which branch was taken on each conditional. For instance, in the code fragment from Section 4.1.1, the adversary can simulate the program, keeping track of the possible values of o and r. After the last object fetch before line 1, if the program issues a fetch for an object that o (respectively r) can point to, then the adversary can infer that the value of h is true (respectively false).

4.2.2 A Simple Type System with Access Labels

In the original Fabric system, objects are placed onto stores that can enforce their labels, including their confidentiality. However, this does not prevent read channels. According to this rule, public data can be stored on a low (adversary-controlled) node. But then accesses to the object from a high context would

violate confidentiality.

Read channels are not controlled in the original Fabric system, but they become easy to exploit once the adversary can provide mobile code that generates such accesses [2]. Read channels are not a Fabric-specific problem, either—holes in the same-origin policy also permit read channels: for example, via images fetched from ad servers, as explained in the introduction to this chapter.

When an object is accessed during computation on a worker, but is not yet cached at the worker, the worker must fetch the object data from the node where it is stored. Thus, the contacted node learns that an access to the object has occurred. The access results in a read channel only if it is a read. To prevent adversaries from exploiting read channels i.e. learning secret information through object fetches, we need to *statically* check Fabric programs to ensure that each object access is secure. Static checking is necessary since read channels are a kind of implicit flow and require reasoning about all possible program paths rather than only the current program path.

An object dereference has an insecure information leak if $L_{pc} \not\sqsubseteq \{\top \rightarrow S\}$ where L_{pc} is the program counter label just before the object dereference and S is the store on which the object is stored. In this context, S is the adversary and L_{pc} is the upper bound on the information contained in the program execution context. The constraint says that if the execution context depends on information that the store is not allowed to view, then an fetching an object from the store in that context is insecure. To prevent such information leaks, we can simply require $L_{pc} \sqsubseteq \{\top \rightarrow S\}$ to be true for *each* object dereference. However, given an object reference, statically, we do not know which store contains the object that the reference points to. The Fabric language does support an expression of

the form `o.store` that returns a reference to the store containing the object `o`. It is possible to evaluate `o.store` without fetching `o`, since the runtime representation of the Fabric reference `o` contains the details of the store, along with the oid. However, `o.store` can only be evaluated at runtime and asserting $L_{pc} \sqsubseteq \{\top \rightarrow \texttt{o.store}\}$ for each dereference of `o` requires tracking the *pc* label at runtime – a prohibitively expensive operation.

To address this issue, we extend the programming language and introduce an *access label* associated with each object. It is a confidentiality-only label that bounds what can be learned from the fact that the object has been accessed. The access label ensures that the object is stored on a node that is trusted to learn about all the accesses to it, and it prevents the object from being accessed from a context that is too high. The access label has no integrity component because there is no integrity dual to read channels.

The access label can also be thought of as a static approximation of the object's location, i.e. a static approximation of the dynamic label $\{\top \rightarrow \texttt{o.store}\}$, which enables static reasoning for read channels. Although, the access label is logically associated with an object, in practice it is declared within the class definition of the object, as part of the label of its fields. Given object label l_u and access label l_a, a label annotation $l_u@l_a$ on a field means that the field, and by extension all instances of that class and all the references to them, have the corresponding access label l_a. Since access labels protect reads and updates and update labels protect only updates, it will always be the case that access labels are more restrictive than update labels, i.e. $l_u \sqsubseteq l_a$.

For example, to declare an object containing public information (in field `data`) that can be accessed without leaking information (according to any prin-

cipal that trusts node n to enforce its confidentiality), we can write code like this:

```
1  class Public {
2          int{}@{⊤→ n} data;
...
11 }
```

Even though the information is public and untrusted (label `{}`), objects of this class can be stored only on nodes that are at least as trusted as node `n`. Conversely, if we had given the field `data` the annotation `{}@{}`, the object could be stored on any node, but the type system would prevent accesses from non-public contexts.

Access labels require two new static checks in Fabric code, that are implemented by extending the Fabric type system:

1. The access label on fields allows the compiler to check all reads from and writes to fields to ensure that they occur in a low context. The program-counter label pc must be lower than the access label (i.e., $pc \sqsubseteq l_a$) at each field access (read or update). This is in addition to the existing check, inherited from Jif [56], that requires $pc \sqsubseteq l_u$ at each update.

2. At the point where an object is constructed using `new`, the node at which the object is created must be able to enforce the access label. In Fabric, an object of class `C` is explicitly allocated at a node `n` using the syntax `new C@n(...)`. We require $pc \sqsubseteq l_a$ and $n \succcurlyeq l_a$ at this point in the code, because node n learns about the future accesses to the object.

4.2.3 Interaction with Object-Oriented Features

In a simple imperative language with mutable cells on the heap and language level references, the label checking rules in the previous section would be enough to prevent read channels. However, the various features of a general purpose object-oriented language offered by Fabric requires further extensions to the type system.

Specification. Access labels are only specified on fields. Access labels need to have a statically known value so that they can be compared with the *pc* label at compile time. Also, the access label of an object cannot depend on the state of the object. This is because access labels help prevent insecure object fetches and if computing the access label would require fetching the object, the purpose of access labels would be defeated. Thus, access labels cannot have dynamic label/principal components such as label/principal variables and fields. However, access labels can contain label/principal class level parameters, since they are both runtime representable and known for an object prior to fetching it. Access labels also need to be runtime representable so that the constraint $n \succcurlyeq l_a$ (equivalent to $l_a \sqsubseteq \{\top \rightarrow n\}$ where n is dynamically known) can be established at all constructor call sites. Thus, access labels cannot have components such as {`this`} and arg labels such as {`x`}, which are not runtime representable. A field whose access label is not explicitly specified is assigned a default access label equal to the field's update label.

Multiple fields/methods. In a class with multiple fields, the access label for the class needs to be computed from the access labels on individual fields, which can all be different from each other. The language allows access labels to be specified on a per-field basis for greater expressiveness. The access label on a

field can thus be specified based on the sensitivity of the program points where that particular field is accessed. The access label for the class is conservatively approximated as the join of the individual field access labels and the confidentiality components of the method begin labels. Only public and package visible members (fields and methods) need to be included in computation of the access label, since accessing private and protected members can be done only after the object has already been fetched and a separate *access* is not necessary.

If a field's access label (explicit or default) has dynamic label or arg label components, an error is thrown and the programmer is expected to rewrite the label by replacing disallowed components with appropriate bounds. Often, this involves the use of label parameters, which are allowed within access labels. Method begin labels, however, can legally contain dynamic and arg label components. To be able to include them into the access label, we require programmer supplied bounds for them expressed within the where clauses of the corresponding method, as shown in the following example:

```
1  class D {
2          int{}@{Chuck→} x;
3          int{}@{} y;
4          void m{*l1;l1;l2}(label l1, label l2)
5          where         l1 <= {Alice→},
6                            {l1} <= {Bob→},
7                            {l2} <= {Bob→}
8          {
...
29         }
30 }
```

The access label for class `D` is `{Chuck→}⊔{}` joined with a bound on `{*l1;l1;l2}` computed using the where clauses. The bound needs to be written using constant and parameter labels, which is `{Alice→;Bob→}` in this case. The access label for `D` is therefore `{Chuck→;Alice→;Bob→}`. The integrity component of all access labels is assigned the dummy value of {⊥ ←}.

In a full system, we can imagine achieving greater precision by boxing each of the individual fields into an object of a new class and including a reference to this object in the original class. Each new class would have a single field with the same access label as the field it boxes, and all the references to instances of these classes can be given the same access label, computed by taking the meet of all the original access labels. This scheme, called class splitting, can actually be performed automatically by the compiler as an extra translation phase. Class splitting would provide more flexibility – for instance in the example above we can place `x` and `y` on different stores, instead of finding a store that can host both

fields. At field accesses, we would need to compare the *pc* label with the access label of the particular field, instead of the class access label, allowing more local reasoning. This allows us to tighten the begin labels of methods that do not access all the fields of its class, to a less restrictive value. It is useful to tighten the begin label when it does not affect where the method can be called from, since it increases the number of possible workers that can execute the method remotely. Class splitting is also motivated by more precision in reasoning of update labels in classes containing fields with different update labels. We leave the design and implementation details of class splitting to future work.

An Alternate Type System. An alternate type system with access labels could associate them with individual object references instead of associating them with all instances of a particular static type. Thus, we could declare access labels on local variables as follows:

```
1  class C {
2          C{}@{} p;
...
12         static void m(Store store) {
13                 C{}@{Alice→} o = new C@store(...);
14                 int y = o.f;
15                 p = o;
...
26         }
27 }
```

The compiler would need to prove on line 13 that $\{\texttt{Alice}\rightarrow\} \sqsubseteq \{\top \rightarrow \texttt{store}\}$. Subsequently, if o is dereferenced as on line 14, the compiler would simply need

to check $pc \sqsubseteq \{\texttt{Alice} \rightarrow\}$. If `o` is copied into another variable/field `p`, we need to check that the access label on `p` is at most as restrictive as the access label on `o`. Thus the assignment on line 15 is secure since $\{\} \sqsubseteq \{\texttt{Alice} \rightarrow\}$. Similar to our type system, if the access label contains dynamic labels or arg labels, the compiler would need the programmer to specify constant upper bound labels for them. Although, superficially, this scheme seems to not require elaborate `where` clauses on every method of a class and thus have a lesser annotation burden, it is easy to see that it results in annotation burden elsewhere. `where` clauses are still required for bounding access labels containing dynamic labels and arg labels. Object references that are passed around as method arguments all need to have an extra access label annotation. This scheme also requires checking $pc \sqsubseteq l_a$ at method calls in addition to checking that the pc is less restrictive than the begin label. More importantly, there are two reasons why we reject this scheme in favour of our earlier proposed one. Firstly, the reasoning associated with accesses to a particular object are spread out over all the individual accesses which could be anywhere in the system. In contrast, in the earlier scheme, the reasoning associated with the access of an object are well contained within the class definition of that object, making it easier to maintain. Secondly, our earlier scheme would be more compatible with a future version of the Fabric compiler that has an automatic class splitting pass. In the alternate scheme the access label becomes a property of the object reference, and accesses cannot be fine tuned depending on the particular field/method accessed. This results in lesser expressiveness.

Inheritance. If a class A inherits another class B, we compute the access label of A by taking the join of field access labels and method begin labels of both A and B. In general, the access label of a class is computed by taking the

join of access labels and begin labels of all members of this class and all its superclasses until we reach `fabric.lang.Object`. Similar to any other class that has no members, `Object` would have an access label of $\{\bot \rightarrow\}$. This would mean that these objects can be placed on any store, but can be accessed only from a public context. This is not restrictive for `Object` since no object reference is ever accessed (a down cast or an instanceof also counts as an access) at the level of `Object`. In general, it is rare for "empty" classes to be accessed via a down cast or an instanceof and thus will not be restrictive for them. In those cases where an access to an empty class needs to happen in a secret context, the programmer can always introduce a dummy field in the class and assign it an appropriate access label. Computing access labels this way ensures that if $A <: B$ then $l_a^A \sqsupseteq l_a^B$.

Overriding. In Jif and earlier version of Fabric, an overriding method needs to have a begin label at least as restrictive as the one in the parent class. For instance, let us define a class `C` that subclasses `D` and overrides the method `D.m`, which is defined as follows:

```
1  void m{*l1}(label{} l1, label{} l2)
2  where   l1 <= {Alice→},
3  {
...
24 }
```

Let us also say that the begin label of `C.m` is more restrictive than the begin label of `D.m`. Since the access label computation requires constant bounds on the begin label of `C.m`, we can imagine defining `C.m` with `where` clauses as follows:

```
1  void m{*l1;*l2;*lbl}(label{} l1, label{} l2)
2  where   l1 <= {Alice→},
3          l2 <= L,
4          lbl <= {Chuck→}
5  {
...
26 }
```

where `L` is a label parameter of `C` and `lbl` is a final label field in `C`. This definition of `C.m` would actually be unsound, since the `where` clauses of an overriding method need to be weaker than (or implied by) the `where` clauses of the overridden method. This can be fixed by using class-level `where` clauses for final label fields and breaking up `l2 <= L` into `l2 <= {Bob}` and `{Bob}<= L` where `l2 <= {Bob}` is a condition that holds true at all call sites of `D.m`. The resulting code is shown in Figure 4.2. Note that the `where` clauses of `D.m` imply the `where` clause of `C.m`. Class-level where clauses can name entities that are in scope at the class level, such as final label/principal fields, class parameters and static constants. Class-level where clauses are checked as post-conditions of the constructors of the class, and are assumed to be true for all instances of the class. The class level `where` clause of the inherited class `C` needs to be stronger than (needs to imply) the class level `where` clause of the parent class `D`.

Interfaces and Abstract Classes. Similar to classes, an interface level access label is computed by taking the join of confidentiality projections of method begin labels. Constant bounds for dynamic labels and arg labels need to be stated for interface methods similar to classes, which might require interface level `where` clauses in some cases. If an interface `I` extends interfaces `J` and `K`,

```
1  class D {
2          void m{*l1}(label{} l1, label{} l2)
3          where   l1 <= {Alice→},
4                  l2 <= {Bob→}
5          {
...
26         }
27 }
28
29 class C[label L] extends D
30 where   {Bob→} <= L,
31         lbl <= {Chuck→}
32 {
33         final label lbl;
34         C() {
35                 lbl = ...
...
46         }
47         void m{*l1;*l2;*lbl}(label{} l1, label{} l2)
48         where   l2 <= L
49         {
...
70         }
71 }
```

Figure 4.2: Access Labels and Method Overriding

the rules for overriding are similar to those for classes. A class `C` that implements an interface `I` needs to ensure proper rules of overriding as well. For instance, the implementation `C.m` cannot have stronger `where` clauses than `I.m`. In general, the type of an implemented method `C.m` needs to be a subtype of the signature `I.m`. Also, the class level `where` clauses of `C` need to imply the interface level `where` clauses of `I`. Since interfaces do not have constructors, interface level `where` clauses are never enforced directly. Rather, they serve as constraints on the class level `where` clauses of the classes that implement it.

Similar to classes, the class level access label for an abstract class is com-

puted by taking the join of field access labels and method begin labels, after replacing dynamic and arg labels with their bounds. Similar to interfaces, class level `where` clauses are not enforced directly but serve as constraints for class level `where` clauses of inherited classes. The `where` clauses on a class `C` that inherits an abstract class `B` need to imply the `where` clauses on `B`.

Casts and Instanceofs. Both a cast and an instanceof require fetching the dynamic type (class object) of the instance and therefore counts as an access. At the program point that the cast/instanceof is performed, say, to a class C, the compiler checks $pc \sqsubseteq l_a^C$. At runtime, prior to fetching the object instance, the worker executing the code checks if $l_a^C \sqsubseteq \{\top \rightarrow \texttt{o.store}\}$ where `o` is the runtime reference to the object. This check can be performed without fetching `o` since the store on which this object is located needs to be encoded within the reference. Also, the cast/instanceof is translated into runtime code in a way that retains the value of l_a^C, possible since access labels are runtime representable. The object is fetched only if the runtime check succeeds.

4.2.4 Interaction with Mobile Code

Access labels also interact with provider-bounded label checking. Recall that the compiler ensures the initial *pc* of methods contain at least as much confidentiality as the label of the code. Therefore, the access label of objects used by confidential code must be at least as high as the confidentiality of the code.

Mobile Fabric also encounters a new kind of read channel that did not exist in the original Fabric system: class object read channels. Fetching an object may require fetching its class, so the class object must be stored on a node that is

trusted to enforce the object's access label. To satisfy this requirement without unnecessary restrictiveness, we can ensure that when an object is created on a node, its class object is stored at a suitably trusted node. Since the node storing the object itself must be such a node, the class object can be replicated onto the same node as the object if necessary. Since class objects are immutable, their replication is harmless in Fabric.

4.2.5 Runtime Mechanisms

Access labels also introduce a new dynamic check. When a worker fetches an object, the access label bounds how much information is leaked to the object's store. However, if the reference to the object is provided by an adversary, there is no guarantee that the store is trusted to learn that information. Therefore, before the fetch is performed, the worker must check dynamically that the store can enforce the access label.

A second potential source of read channels arises in distributed computations spanning multiple worker nodes. In Fabric, a cached object is owned by a single worker in a multi-worker distributed transaction. This worker is contacted when the object is accessed. If the worker is not trusted to enforce the object's access label, this access creates an insecure read channel.

A solution to this problem is to modify Fabric's *writer map*, which securely maintains the mapping between oids and the workers that currently own them. If a node is not trusted to enforce the access label, it must give up ownership at control transfers. Instead of putting current ownership information into the writer map, the node inserts the object update directly into the map. If a node

were to misbehave by claiming ownership incorrectly, other nodes will refuse to attempt to access the object. This fatal error is a termination channel, which is ignored.

4.3 Automatic Elimination of Read Channels

Experience suggests that programming against a type system with access labels can easily get tricky and/or cumbersome. For instance, in the FriendMap example from Mobile Fabric [2] (referring to Figure 1) which uses the access labels type system from Section 4.2.2, Alice's client needs to access parts of a map from MapServ, an untrusted third-party map service. It does this so that it can post Alice's friend's locations on Snapp, a social network. Since none of the users of the social network trust the map service with their location, simply accessing specific parts of the map corresponding to the friend's locations would leak that information to the map service via a read channel. To make this work, Alice's client first creates a "bounding box" of the friend's locations and then creates a local copy of the map containing all the map parts within the bounding box. This leaks minimal information to MapServ. However, creating such local copies and making sure that the copies are made correctly and that ensuring that all the objects that need to be copied are actually copied can be unnecessarily burdensome. The rest of this chapter investigates automatic methods based on program-analysis to help the programmer address such read channel related issues.

In the automated scheme, programmers write programs without concern for read channels. Objects are still annotated with access labels, restricting the hosts

on which they can be placed, and thus restricting the hosts that can potentially learn about accesses to the objects. An object with a high access label can be placed only on a high host. An object with a low access label can be placed anywhere. However, access labels in the type system do not restrict where the objects can be accessed from. Program analysis and transformation methods are used to automatically eliminate any read channels the program might contain.

With the previous type system with access labels, many details needed to be specified by the programmer. In contrast, the programming model offered by automatic read channel elimination is more expressive because:

- It is more concise. It copies data structures automatically without the programmer having to explicitly do the copying himself.
- It provides a guarantee that all relevant objects are copied, which is not available in the manual copying case. In the manual copying case with a simple type system, it is awkward, annoying and burdensome to do the copying and having to check that the copies are made correctly.

We investigate automatic elimination of read channels by considering a simplified source language. The simplified language contains all the features of the Fabric language that are essential for exploring the technique: labeled types, records and arrays, loops, function calls and recursive types. It does not contain more complex features such as object-oriented features, exceptions, etc. This simplification allows us to focus on the interesting aspects of automatic read channel elimination. The source language syntax, type system and semantics are presented in Section 4.3.1. We then discuss an automatic program transformation technique based on abstract interpretation, to transform a given pro-

gram in this source language to a program that is guaranteed to not have read channel vulnerabilities. The abstract interpretation is presented in Section 4.3.2 and the discussion on the transformation technique follows.

In Section 4.3.3, we present an alternate interpretation for the same source program that eliminates the read channel. It does this by running the abstract interpretation on each block of sensitive code, before executing that code normally. The abstract interpretation computes the set of possible objects that the sensitive code might need, independent of secret information. This makes it safe to prefetch exactly these objects into a cache, and then use the cache while executing the block of sensitive code. Since the alternate interpretation is a purposeful hybrid of the regular interpretation (denotational semantics) and the abstract interpretation, we call it the *interleaved interpretation*.

In Section 4.3.4, we discuss the implementation of the three interpreters and the performance overhead of the approach. A real system would use the interleaved semantics to transform the source program into a program in a suitable target language. The performance results from the implementation of the interleaved interpretation is only suggestive and is used to highlight the relative overhead rather than the absolute overhead. A transformed program would perform significantly better, since there would be no interpretation overhead. Using the interleaved semantics to perform an actual program transformation is left to future work.

4.3.1 Source Language

The source language is an extension of the simple while-language with higher level language features such as arrays and records and information flow labels. Arrays and records are also associated with access labels, similar to the access labels in Section 4.2.2. The difference is that these access labels restrict only the stores on which the objects can be created. They do not restrict accesses to the object. Accesses to objects will be rendered secure by our program transformation technique. Let us call the source language `IMPPAR` (IMP with procedures, arrays and records). We now present the grammar, type system and denotational semantics for `IMPPAR`. Denotational semantics is used instead of the standard operational semantics so that the abstract interpretation in the following section can build upon the denotational semantics in this section.

Grammar

The grammar shown in Figure 4.3 is self-explanatory. Most language constructs are fairly standard. The construct *se* stands for "simple expression", defined as either a value or a local variable. The idea is to define a (non-exhaustive) subset of expressions which will semantically never require a remote fetch. Restricting certain expressions such as method arguments, initializers, etc. to be simple expressions helps simplify some of the presentation related to read channels.

Type System

The typing rules for `IMPPAR` are shown in Figures 4.4 (typing rules for expressions) and 4.5 (typing rules for statements). The type system is similar to the

Variables	x, y	$\in$	$\mathcal{V}$
Names	f, g	$\in$	$\mathcal{F}$
Binary Operators	$\oplus$	$\in$	$\{+, -, \div, \times, \wedge, \vee, >, <, \geq, \leq, ==, !=\}$
Unary Operators	$\ominus$	$\in$	$\{-, !\}$
Confidentiality Labels	l	$\in$	$\mathcal{L} = \{\top, \bot\}$
PC/Begin/Access Labels	k	$\in$	$\mathcal{L}$
Hosts	h	$\in$	$\mathcal{H}$
Types	α_l, τ_l	::=	$\texttt{int}_l \mid \texttt{boolean}_l \mid \omega_l^k$
Heap Types	ω_l^k	::=	$\texttt{array}_l(k, \tau_{l'}) \mid$
			$\mu\alpha.\texttt{record}_l(k, f_1 : \tau_{1l_1}, f_2 : \tau_{2l_2}, ..., f_m : \tau_{ml_m})$
Function Body	fb	::=	$\tau_{1l_1}\ x_1, \tau_{2l_2}\ x_2, ..., \tau_{ml_m}\ x_m\ ;\ s\ ;\ \texttt{return}\ x$
Function Declarations	d^i	::=	$\tau_l^i\ g^i\{k^i\}@h^i(\tau^i_{1l^i_1}\ x_1, \tau^i_{2l^i_2}\ x_2, ..., \tau^i_{ml^i_{m_i}}\ x_{m_i})\ \{fb^i\}$
Simple Expressions	se	::=	$n \mid true \mid false \mid \texttt{null} \mid x$
Expressions	e	::=	$se \mid \ominus e \mid e_1 \oplus e_2 \mid (e) \mid x.f \mid x[y] \mid$
			$length(x) \mid g(se_1, se_2, ..., se_m) \mid$
			$\texttt{new}\ \tau_l\ se \mid \texttt{new}\ \tau_l\ \{f_1 = se_1, f_2 = se_2, ..., f_m = se_m\}$
Statements	s	::=	$skip \mid x = e \mid x.f = e \mid$
			$x[y] = e \mid s_1; s_2 \mid \texttt{if}\ x\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid$
			$\texttt{while}\ x\ \texttt{do}\ s \mid g(se_1, se_2, ..., se_m)$
Program	P	::=	$d^1\ d^2\ ...\ d^p\ fb$

Figure 4.3: Grammar for the IMPPAR language

Fabric type system based on access labels (Section 4.2.2). The important difference is that the typing rules here only restrict the stores on which an object can be created on – they don't restrict where these objects can be accessed from (i.e. only the second of the two static checks at the end of Section 4.2.2 is applied). This allows the programmer greater choice and expressiveness while accessing objects. This type system is also simpler than the Fabric access labels type system, since it is for a simpler language.

$L(h)$: Label associated with host h
$\sqsubseteq$: Information flow ordering on labels
Typing Judgement: $\Gamma; h; pc \vdash e : \tau_l$

$$\frac{}{\Gamma; h; pc \vdash n : \texttt{int}_\bot} \quad \frac{}{\Gamma; h; pc \vdash true : \texttt{boolean}_\bot} \quad \frac{}{\Gamma; h; pc \vdash false : \texttt{boolean}_\bot}$$

$$\frac{}{\Gamma; h; pc \vdash \texttt{null} : \omega^k_\bot} \quad \frac{x \in \Gamma}{\Gamma; h; pc \vdash x : \Gamma(x)} \quad \frac{\Gamma; h; pc \vdash e : \texttt{int}_l}{\Gamma; h; pc \vdash -e : \texttt{int}_l} \quad \frac{\Gamma; h; pc \vdash e : \texttt{boolean}_l}{\Gamma; h; pc \vdash\ !e : \texttt{boolean}_l}$$

$$\frac{\Gamma; h; pc \vdash e_1 : \texttt{int}_{l_1} \qquad \Gamma; h; pc \vdash e_2 : \texttt{int}_{l_2} \qquad \oplus \in \{+, -, \div, \times\}}{\Gamma; h; pc \vdash e_1 \oplus e_2 : \texttt{int}_{l_1 \sqcup l_2}}$$

$$\frac{\Gamma; h; pc \vdash e_1 : \texttt{int}_{l_1} \qquad \Gamma; h; pc \vdash e_2 : \texttt{int}_{l_2} \qquad \oplus \in \{>, <, \geq, \leq, ==, !=\}}{\Gamma; h; pc \vdash e_1 \oplus e_2 : \texttt{boolean}_{l_1 \sqcup l_2}}$$

$$\frac{\Gamma; h; pc \vdash e_1 : \texttt{boolean}_{l_1} \qquad \Gamma; h; pc \vdash e_2 : \texttt{boolean}_{l_2} \qquad \oplus \in \{\wedge, \vee\}}{\Gamma; h; pc \vdash e_1 \oplus e_2 : \texttt{boolean}_{l_1 \sqcup l_2}}$$

$$\frac{\Gamma; h; pc \vdash e_1 : \omega^k_{l_1} \qquad \Gamma; h; pc \vdash e_2 : \omega^k_{l_2} \qquad \oplus \in \{==, !=\}}{\Gamma; h; pc \vdash e_1 \oplus e_2 : \texttt{boolean}_{l_1 \sqcup l_2}}$$

$$\frac{\Gamma; h; pc \vdash e : \tau_l}{\Gamma; h; pc \vdash (e) : \tau_l} \quad \frac{\begin{array}{c}\Gamma; h; pc \vdash x : \mu\alpha.\texttt{record}_l(k, ..., f : \tau_{l'}, ...) \\ \tau'_{l'} = \tau\{\mu\alpha.\texttt{record}(k, ..., f : \tau_{l'}, ...)/\alpha\}\end{array}}{\Gamma; h; pc \vdash x.f : \tau'_{l \sqcup l'}}$$

$$\frac{\Gamma; h; pc \vdash x : \texttt{array}_{l_1}(k, \tau_l) \qquad \Gamma; h; pc \vdash y : \texttt{int}_{l_2}}{\Gamma; h; pc \vdash x[y] : \tau_{l \sqcup l_1 \sqcup l_2}} \quad \frac{\Gamma; h; pc \vdash x : \texttt{array}_l(k, \tau_{l'})}{\Gamma; h; pc \vdash length(x) : \texttt{int}_{l \sqcup l'}}$$

$$\frac{\Gamma(g) = \tau_l\ \{k\}@h'(\tau_{1l_1}, \tau_{2l_2}, ..., \tau_{ml_m}) \qquad \Gamma; h; pc \vdash se_i : \tau_{il'_i} \qquad pc \sqcup l'_i \sqsubseteq l_i \qquad pc \sqsubseteq k}{\Gamma; h; pc \vdash g(se_1, se_2, ..., se_m) : \tau_l}$$

$$\frac{\Gamma; h; pc \vdash se : \texttt{int}_{l_1} \qquad \tau_l = \texttt{array}_l(k, \tau_{l_2}) \qquad pc \sqcup l_1 \sqsubseteq l_2 \qquad l \sqsubseteq L(h) \qquad k \sqsubseteq L(h)}{\Gamma; h; pc \vdash \texttt{new}\ \tau_l\ se : \tau_l}$$

$$\frac{\begin{array}{c}\Gamma; h; pc \vdash se_i : \tau_{il'_i}\{\mu\alpha.\texttt{record}(k, f_1 : \tau_{1l_1}, ..., f_m : \tau_{ml_m})/\alpha\} \\ \tau_l = \mu\alpha.\texttt{record}_l(k, f_1 : \tau_{1l_1}, ..., f_m : \tau_{ml_m}) \qquad pc \sqcup l'_i \sqsubseteq l_i \qquad l_i \sqsubseteq L(h) \qquad k \sqsubseteq L(h)\end{array}}{\Gamma; h; pc \vdash \texttt{new}\ \tau_l\ \{f_1 = se_1, f_2 = se_2, ..., f_m = se_m\} : \tau_l}$$

$$\frac{\Gamma' = \Gamma, x_1 : \tau_{1l_1}, x_2 : \tau_{2l_2}, ..., x_m : \tau_{ml_m} \qquad \Gamma'; h; pc \vdash s \qquad \Gamma'(x) = \tau_l}{\Gamma; h; pc \vdash \tau_{1l_1}\ x_1, \tau_{2l_2}\ x_2, ..., \tau_{ml_m}\ x_m\ ;\ s\ ;\ \texttt{return}\ x : \tau_l}$$

Figure 4.4: Typing rules for expressions in the IMPPAR language

Typing Judgement: $\Gamma; h; pc \vdash s$

$$\frac{}{\Gamma; h; pc \vdash skip} \qquad \frac{\Gamma; h; pc \vdash e : \omega^k_{l'} \qquad \Gamma(x) = \omega^k_l \qquad pc \sqcup l' \sqsubseteq l}{\Gamma; h; pc \vdash x = e}$$

$$\frac{\Gamma; h; pc \vdash e : \texttt{int}_{l'} \qquad \Gamma(x) = \texttt{int}_l \qquad pc \sqcup l' \sqsubseteq l}{\Gamma; h; pc \vdash x = e}$$

$$\frac{\Gamma; h; pc \vdash e : \texttt{boolean}_{l'} \qquad \Gamma(x) = \texttt{boolean}_l \qquad pc \sqcup l' \sqsubseteq l}{\Gamma; h; pc \vdash x = e}$$

$$\frac{\begin{array}{c}\Gamma; h; pc \vdash x : \ \mu\alpha.\texttt{record}_{l_1}(k, ..., f : \tau_l, ...) \\ \Gamma; h; pc \vdash e : \tau_{l_2}\{\mu\alpha.\texttt{record}(k, ..., f : \tau_l, ...)/\alpha\} \qquad pc \sqcup l_1 \sqcup l_2 \sqsubseteq l\end{array}}{\Gamma; h; pc \vdash x.f = e}$$

$$\frac{\begin{array}{c}\Gamma; h; pc \vdash x : \texttt{array}_{l_1}(k, \tau_l) \\ \Gamma; h; pc \vdash y : \texttt{int}_{l_2} \qquad \Gamma; h; pc \vdash e : \tau_{l_3} \qquad pc \sqcup l_1 \sqcup l_2 \sqcup l_3 \sqsubseteq l\end{array}}{\Gamma; h; pc \vdash x[y] = e}$$

$$\frac{\Gamma; h; pc \vdash s_1 \qquad \Gamma; h; pc \vdash s_2}{\Gamma; h; pc \vdash s_1; s_2} \qquad \frac{\Gamma; h; pc \vdash x : \texttt{boolean}_l \qquad \Gamma; h; pc \sqcup l \vdash s_1 \qquad \Gamma; h; pc \sqcup l \vdash s_2}{\Gamma; h; pc \vdash \texttt{if } x \texttt{ then } s_1 \texttt{ else } s_2}$$

$$\frac{\Gamma; h; pc \vdash x : \texttt{boolean}_l \qquad \Gamma; h; pc \sqcup l \vdash s}{\Gamma; h; pc \vdash \texttt{while } x \texttt{ do } s} \qquad \frac{\Gamma; h; pc \vdash g(se_1, se_2, ..., se_m) : \tau_l}{\Gamma; h; pc \vdash g(se_1, se_2, ..., se_m)}$$

$$\frac{x_1 : \tau_{1l_1}, x_2 : \tau_{2l_2}, ..., x_m : \tau_{ml_m}; h; k \vdash fb : \tau_l \qquad k \sqsubseteq L(h) \qquad l_i \sqsubseteq L(h)}{\vdash \tau_l \ g\{k\}@h(\tau_{1l_1} \ x_1, \tau_{2l_2} \ x_2, ..., \tau_{ml_m} \ x_m) \ \{fb\}}$$

$$\frac{\vdash d^i \qquad \vdash fb : \tau_l}{\vdash d^1 \ d^2 \ ... \ d^m \ fb}$$

Figure 4.5: Typing rules for statements in the IMPPAR language

Denotational Semantics

The denotational semantics is defined only for well-typed programs, indicated by defining the semantic rules over the typing judgement of the corresponding expression or statement. The translation function $\mathcal{E}$ for typed expressions takes in the variable environment ρ, the heap σ, the function environment ϕ and the host h and returns a tuple of a value and the updated heap. The value is an element of the domain D_τ which is the translation of the type τ of the expression. The translation function C for statements takes in the same input, but returns a tuple of the updated variable environment and the updated heap. To simplify presentation, the semantic rules for certain expressions and statements elide the typing judgement. In these rules, the type information is either not relevant or is obvious from context. The abstract interpretation in the next section will build upon the denotational semantics presented here.

Preliminaries:

Locations $\ell\ (\in Loc)\ ::=\ (\omega, n, h)\ \mid\ NULL$

$h \in \mathcal{H}$

$\mathcal{E}[\![\Gamma; h; pc \vdash e : \tau]\!]_{\rho\sigma\phi h} \in (D_\tau \times \Sigma)_\perp$

$C[\![\Gamma; h; pc \vdash s]\!]_{\rho\sigma\phi h} \in (P \times \Sigma)_\perp$

where

$D_\tau = \mathcal{T}[\![\tau]\!]$

$\rho \in P = Var \to Value : \rho(x) \in \mathcal{T}[\![\Gamma(x)]\!] \wedge (\omega(\rho(x)) = \Gamma(x), \texttt{if}\ \rho(x) \in Loc)$

$\sigma \in \Sigma = Loc \to Object : \sigma(\ell) \in \mathbb{Z} \times (\mathbb{Z} \to \mathcal{T}[\![\tau]\!]),\ \texttt{if}\ \omega(\ell) = \texttt{array}(k, \tau_l)$

$\sigma(\ell) \in String \to Value \wedge \sigma(\ell)(\text{“}f_i\text{”}) \in \mathcal{T}[\![\tau_i\{\omega(\ell)/\alpha\}]\!]),$

$\texttt{if}\ \omega(\ell) = \mu\alpha.\texttt{record}(k, f_1 : \tau_{1l_1}, f_2 : \tau_{2l_2}, ..., f_m : \tau_{ml_m})$

$\mathcal{T}[\![\texttt{int}]\!] = \mathbb{Z}, \mathcal{T}[\![\texttt{boolean}]\!] = \mathbb{B} = \{T, F\}, \mathcal{T}[\![\omega]\!] = Loc$

$ZERO(\texttt{int}) = 0, ZERO(\texttt{boolean}) = F, ZERO(\omega) = NULL$

$\phi \in \mathcal{F} = (P \times \Sigma \to (D_{\tau^1} \times \Sigma)_\perp)\ \times\ (P \times \Sigma \to (D_{\tau^2} \times \Sigma)_\perp)\ \times\ ...\ \times\ (P \times \Sigma \to (D_{\tau^p} \times \Sigma)_\perp)$

$\phi = \texttt{fix}\ \lambda F \in \mathcal{F}.(\lambda\rho \in P.\lambda\sigma \in \Sigma.\mathcal{E}[\![fb^1]\!]_{\rho\sigma F h^1},$

$\vdots$

$\lambda\rho \in P.\lambda\sigma \in \Sigma.\mathcal{E}[\![fb^p]\!]_{\rho\sigma F h^p})$

$Value = \mathbb{Z} + \mathbb{B} + Loc$

$Object = Arr + Rec$

$Arr = \mathbb{Z} \times (\mathbb{Z} \to Value)$

$Rec = String \to Value$

Given $f : D \to E_\perp,$ define $f^* : D_\perp \to E_\perp = \lambda d : D_\perp.\texttt{if}\ d = \lfloor d' \rfloor\ \texttt{then}\ f(d')\ \texttt{else}\ \perp_E$

Define $\texttt{let}\ \lfloor x \rfloor = e_1\ \texttt{in}\ e_2 = (\lambda x.e_2)^* e_1$

Use $\lfloor v, \sigma \rfloor$ and $\lfloor (v, \sigma) \rfloor$ interchangeably

Denotational Semantics for Expressions

$$
\begin{aligned}
\mathcal{E}[\![\Gamma; h; pc \vdash n : \texttt{int}_{\perp}]\!]_{\rho\sigma\phi h} &= \lfloor n, \sigma \rfloor \\
\mathcal{E}[\![\Gamma; h; pc \vdash true : \texttt{boolean}_{\perp}]\!]_{\rho\sigma\phi h} &= \lfloor T, \sigma \rfloor \\
\mathcal{E}[\![\Gamma; h; pc \vdash false : \texttt{boolean}_{\perp}]\!]_{\rho\sigma\phi h} &= \lfloor F, \sigma \rfloor \\
\mathcal{E}[\![\Gamma; h; pc \vdash NULL : \omega^{k}_{\perp}]\!]_{\rho\sigma\phi h} &= \lfloor NULL, \sigma \rfloor \\
\mathcal{E}[\![\Gamma; h; pc \vdash x : \Gamma(x)]\!]_{\rho\sigma\phi h} &= \lfloor \rho(x), \sigma \rfloor \\
\mathcal{E}[\![\Gamma; h; pc \vdash x.f : \tau_{l \sqcup l'}]\!]_{\rho\sigma\phi h} &= \texttt{let } \lfloor \ell, \sigma \rfloor = \\
&\quad \mathcal{E}[\![\Gamma; h; pc \vdash x : \mu\alpha.\texttt{record}_{l}(k, ..., f : \tau_{l'}, ...)]\!]_{\rho\sigma\phi h} \texttt{ in} \\
&\quad \texttt{if } \ell = NULL \texttt{ then } \perp \texttt{ else } \lfloor \sigma(\ell)(\text{“}f\text{”}), \sigma \rfloor \\
\mathcal{E}[\![\Gamma; h; pc \vdash x[y] : \tau_{l \sqcup l_1 \sqcup l_2}]\!]_{\rho\sigma\phi h} &= \texttt{let } \lfloor \ell, \sigma \rfloor = \\
&\quad \mathcal{E}[\![\Gamma; h; pc \vdash x : \texttt{array}_{l_1}(k, \tau_l)]\!]_{\rho\sigma\phi h} \texttt{ in} \\
&\quad \texttt{if } \ell = NULL \texttt{ then } \perp \texttt{ else} \\
&\quad \texttt{let } \lfloor i, \sigma \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash y : \texttt{int}_{l_2}]\!]_{\rho\sigma\phi h} \texttt{ in} \\
&\quad \texttt{let } (n, a) = \sigma(\ell) \texttt{ in} \\
&\quad \texttt{if } i \geq n \texttt{ then } \perp \texttt{ else } \lfloor a(i), \sigma \rfloor \\
\mathcal{E}[\![\Gamma; h; pc \vdash length(x) : \texttt{int}_{l \sqcup l'}]\!]_{\rho\sigma\phi h} &= \texttt{let } \lfloor \ell, \sigma \rfloor = \\
&\quad \mathcal{E}[\![\Gamma; h; pc \vdash x : \texttt{array}_{l_1}(k, \tau_l)]\!]_{\rho\sigma\phi h} \texttt{ in} \\
&\quad \texttt{if } \ell = NULL \texttt{ then } \perp \texttt{ else} \\
&\quad \texttt{let } (n, a) = \sigma(\ell) \texttt{ in } \lfloor n, \sigma \rfloor \\
\mathcal{E}[\![\Gamma; h; pc \vdash e_1 \oplus e_2 : \tau_l]\!]_{\rho\sigma\phi h} &= \texttt{let } \lfloor v_1, \sigma_1 \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash e_1 : \tau_{1 l_1}]\!]_{\rho\sigma\phi h} \texttt{ in} \\
&\quad \texttt{let } \lfloor v_2, \sigma_2 \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash e_2 : \tau_{2 l_2}]\!]_{\rho\sigma_1\phi h} \texttt{ in} \\
&\quad \texttt{in if } v_1 \oplus v_2 = \text{undefined } \texttt{then } \perp \\
&\quad \texttt{else } \lfloor v_1 \oplus v_2, \sigma_2 \rfloor
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\ominus e]\!]_{\rho\sigma\phi h} &= \texttt{let}\ \lfloor v, \sigma_1 \rfloor = \mathcal{E}[\![e]\!]_{\rho\sigma\phi h}\ \texttt{in} \\
&\quad \texttt{if}\ \ominus v = \text{undefined}\ \texttt{then}\ \bot\ \texttt{else}\ \lfloor \ominus v, \sigma_1 \rfloor \\
\mathcal{E}[\![g^i(se_1, se_2, ..., se_m)]\!]_{\rho\sigma\phi h'} &= \texttt{let}\ \lfloor v_1, \sigma \rfloor = \mathcal{E}[\![se_1]\!]_{\rho\sigma\phi h'}\ \texttt{in} \\
&\quad \texttt{let}\ \lfloor v_2, \sigma \rfloor = \mathcal{E}[\![se_2]\!]_{\rho\sigma\phi h'}\ \texttt{in} \\
&\quad \vdots \\
&\quad \texttt{let}\ \lfloor v_m, \sigma \rfloor = \mathcal{E}[\![se_m]\!]_{\rho\sigma\phi h'}\ \texttt{in} \\
&\quad \texttt{let}\ \rho' = [x_1 \mapsto v_1, x_2 \mapsto v_2, ..., x_m \mapsto v_m]\ \texttt{in} \\
&\quad (\pi_i \phi)(\rho', \sigma) \\
\mathcal{E}[\![fb]\!]_{\rho\sigma\phi h} &= \texttt{let}\ \rho' = \\
&\quad \rho[x_1 \mapsto ZERO(\tau_1), x_2 \mapsto ZERO(\tau_2), ..., x_m \mapsto ZERO(\tau_m)] \\
&\quad \texttt{in let}\ \lfloor \rho'', \sigma' \rfloor = C[\![s]\!]_{\rho'\sigma\phi h} \\
&\quad \texttt{in}\ \lfloor \rho''(x), \sigma' \rfloor \\
&\quad \text{where}\ fb = \tau_{1 l_1}\ x_1, \tau_{2 l_2}\ x_2, ..., \tau_{m l_m}\ x_m\ ;\ s\ ;\ \texttt{return}\ x \\
\mathcal{E}[\![\texttt{new}\ \tau\ se]\!]_{\rho\sigma\phi h} &= \texttt{let}\ \lfloor n, \sigma \rfloor = \mathcal{E}[\![se]\!]_{\rho\sigma\phi h}\ \texttt{in} \\
&\quad \texttt{let}\ z = ZERO(\tau)\ \texttt{in} \\
&\quad \texttt{let}\ \ell = \ell_{\tau,h}^{fresh}\ \texttt{in} \\
&\quad \lfloor \ell, \sigma[\ell \mapsto (n, [1 \mapsto z, 2 \mapsto z, ..., n \mapsto z])] \rfloor \\
&\quad \text{where}\ \tau = \texttt{array}_l(k, \tau_{l'})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\texttt{new}\ \tau\ \{f_1 = se_1, f_2 = se_2, ..., f_m = se_m\}]\!]_{\rho\sigma\phi h} \quad = \quad & \texttt{let}\ \lfloor v_1, \sigma \rfloor = \mathcal{E}[\![se_1]\!]_{\rho\sigma\phi h}\ \texttt{in} \\
& \texttt{let}\ \lfloor v_2, \sigma \rfloor = \mathcal{E}[\![se_2]\!]_{\rho\sigma\phi h}\ \texttt{in} \\
& \vdots \\
& \texttt{let}\ \lfloor v_n, \sigma \rfloor = \mathcal{E}[\![se_n]\!]_{\rho\sigma\phi h}\ \texttt{in} \\
& \texttt{let}\ \ell = \ell_{\tau,h}^{fresh}\ \texttt{in} \\
& \lfloor \ell, \sigma[\ell \mapsto [\text{“}f_1\text{”} \mapsto v_1, ..., \text{“}f_n\text{”} \mapsto v_n]] \rfloor
\end{aligned}
$$

where $\tau = \mu\alpha.\texttt{record}_l(k, f_1 : \tau_{1l_1}, ..., f_n : \tau_{nl_n})$

Denotational Semantics for Statements

$$
\begin{aligned}
C[\![skip]\!]_{\rho\sigma\phi h} \quad = \quad & \lfloor \rho, \sigma \rfloor \\
C[\![x = e]\!]_{\rho\sigma\phi h} \quad = \quad & \texttt{let}\ \lfloor v, \sigma' \rfloor = \mathcal{E}[\![e]\!]_{\rho\sigma\phi h}\ \texttt{in} \\
& \lfloor \rho[x \mapsto v], \sigma' \rfloor \\
C[\![\Gamma; h; pc \vdash x.f = e]\!]_{\rho\sigma\phi h} \quad = \quad & \texttt{let}\ \lfloor v, \sigma' \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash e : \tau_{l_2}]\!]_{\rho\sigma\phi h}\ \texttt{in} \\
& \texttt{let}\ \lfloor \ell, \sigma' \rfloor = \\
& \mathcal{E}[\![\Gamma; h; pc \vdash x : \mu\alpha.\texttt{record}_{l_1}(k, ..., f : \tau_l, ...)]\!]_{\rho\sigma\phi' h}\ \texttt{in} \\
& \texttt{if}\ \ell = NULL\ \texttt{then}\ \bot\ \texttt{else} \\
& \lfloor \rho, \sigma'[\ell \mapsto \sigma'(\ell)[\text{“}f\text{”} \mapsto v]] \rfloor
\end{aligned}
$$

$$
\begin{aligned}
C[\![\Gamma; h; pc \vdash x[y] = e]\!]_{\rho\sigma\phi h} &= \texttt{let } \lfloor v, \sigma' \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash e]\!]_{\rho\sigma\phi h} \texttt{ in} \\
&\quad \texttt{let } \lfloor i, \sigma' \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash y : \texttt{int}_{l_2}]\!]_{\rho\sigma\phi' h} \texttt{ in} \\
&\quad \texttt{let } \lfloor \ell, \sigma' \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash x : \texttt{array}_{l_1}(k, \tau_l)]\!]_{\rho\sigma\phi' h} \texttt{ in} \\
&\quad \texttt{if } \ell = \mathit{NULL} \texttt{ then } \bot \texttt{ else} \\
&\quad \texttt{let } (n, a) = \sigma'(\ell) \texttt{ in} \\
&\quad \texttt{if } i \geq n \texttt{ then } \bot \texttt{ else} \\
&\quad \lfloor \rho, \sigma'[\ell \mapsto (n, a[i \mapsto v])] \rfloor \\
C[\![s_1; s_2]\!]_{\rho\sigma\phi h} &= \texttt{let } \lfloor \rho', \sigma' \rfloor = C[\![s_1]\!]_{\rho\sigma\phi h} \texttt{ in } C[\![s_2]\!]_{\rho'\sigma'\phi h} \\
C[\![\texttt{if } x \texttt{ then } s_1 \texttt{ else } s_2]\!]_{\rho\sigma\phi h} &= \texttt{let } \lfloor v, \sigma \rfloor = \mathcal{E}[\![x]\!]_{\rho\sigma\phi h} \texttt{ in} \\
&\quad \texttt{if } v = \mathit{true} \texttt{ then } C[\![s_1]\!]_{\rho\sigma\phi h} \\
&\quad \texttt{else } C[\![s_2]\!]_{\rho\sigma\phi h} \\
C[\![\texttt{while } x \texttt{ do } s]\!]_{\rho\sigma\phi h} &= \texttt{let } f = (\texttt{fix } \lambda w : P \times \Sigma \to (P \times \Sigma)_\bot. \\
&\quad \lambda\rho' : P. \lambda\sigma' : \Sigma. \\
&\quad \texttt{let } \lfloor v, \sigma' \rfloor = \mathcal{E}[\![x]\!]_{\rho'\sigma'\phi h} \texttt{ in} \\
&\quad \texttt{let } \lfloor \rho'', \sigma'' \rfloor = C[\![s]\!]_{\rho'\sigma'\phi h} \texttt{ in} \\
&\quad \texttt{if } v = \mathit{true} \texttt{ then } w(\rho'', \sigma'') \texttt{ else } (\rho', \sigma')) \texttt{ in} \\
&\quad f(\rho, \sigma) \\
C[\![g(se_1, se_2, ..., se_m)]\!]_{\rho\sigma\phi h} &= \texttt{let } \lfloor v, \sigma' \rfloor = \mathcal{E}[\![g(se_1, se_2, ..., se_m)]\!]_{\rho\sigma\phi h} \texttt{ in} \\
&\quad \lfloor \rho, \sigma' \rfloor
\end{aligned}
$$

4.3.2 Abstract Interpretation

A correct IMPPAR program may still have read channel vulnerabilities. We now present a methodology for automatically eliminating read channel vulnerabilities in a given IMPPAR program. We first present, in this section, an abstract interpretation that computes for a given IMPPAR program segment, the set of all

objects that are dereferenced within it and have a low access label. Referring to Figure 4.6, the left-hand side shows the contract for the regular denotational semantics of a program segment (statement(s) or expression). The denotational semantics takes the variable environment and the heap (ρ and σ) as input and gives the modified heap and either the modified variable environment (if it is a statement) or the value v (if it is an expression) as output. We elide the function environment and the host (ϕ and h) inputs (refer Section 4.3.1) in the diagrams for clarity.

The right-hand side of Figure 4.6 shows how the contract for the abstract interpretation is based on that of the denotational semantics. Instead of the variable environment and heap, their abstract versions ($\hat{\rho}$ and $\hat{\sigma}$) are taken as input. An additional abstract cache ($\hat{c}$) is taken as input, which keeps track of the *references* to objects that are accessed in a way that leaks information through a read channel. The actual contents of the object are not stored in $\hat{c}$.

Next, we formalize these ideas in our presentation of the abstract interpretation. We lay the groundwork and define some notation in the Preliminaries. Then, we define the abstract interpretation for expressions and statements. At the end of this subsection, we discuss how we compute the fixed point for loops and recursive functions. In Section 4.3.3, we show how the abstract interpretation can be interleaved with the original `IMPPAR` semantics to yield an interpretation of an `IMPPAR` program such that data is prefetched into a cache before it is read within a secret context. Given enough cache, the program executes without any read channels.

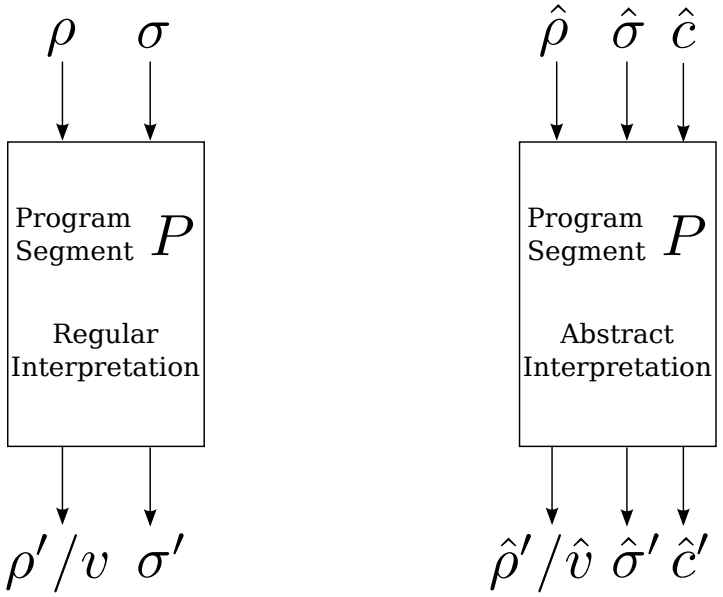

Figure 4.6: Regular Interpretation (Denotational Semantics) and Abstract Interpretation

Abstract Interpretation

Preliminaries. (extending from denotational semantics)

$\mathcal{E}[\![\Gamma; h; pc \vdash e : \tau]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \in \hat{D}_\tau \times \hat{\Sigma} \times \hat{C}$

$C[\![\Gamma; h; pc \vdash s]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \in \hat{P} \times \hat{\Sigma} \times \hat{C}$ where

$D_\tau = \mathcal{T}[\![\tau]\!]$ and $\hat{D}_\tau = \mathcal{P}(D_\tau)$

$\hat{\rho} \in \hat{P} = Var \to \widehat{Value} : \hat{\rho}(x) \in \mathcal{P}(\mathcal{T}[\![\Gamma(x)]\!]) \wedge (\omega(\ell \in \hat{\rho}(x)) = \Gamma(x), \texttt{if}\ \hat{\rho}(x) \in \mathcal{P}(Loc))$

$\hat{\sigma} \in \hat{\Sigma} = Loc \to \widehat{Object} : \hat{\sigma}(\ell) \in \mathcal{P}(\mathbb{Z}) \times (\mathbb{Z} \to \mathcal{P}(\mathcal{T}[\![\tau]\!])),\ \texttt{if}\ \omega(\ell) = \texttt{array}(k, \tau_l)$

$\hat{\sigma}(\ell) \in String \to \widehat{Value} \wedge \hat{\sigma}(\ell)(\text{“}f_i\text{”}) \in \mathcal{P}(\mathcal{T}[\![\tau_i\{\omega(\ell)/\alpha\}]\!])),$

$\texttt{if}\ \omega(\ell) = \mu\alpha.\texttt{record}(k, f_1 : \tau_{1l_1}, f_2 : \tau_{2l_2}, ..., f_m : \tau_{ml_m})$

$\hat{c} \in \hat{C} = \mathcal{P}(Loc + Loc \times \mathbb{Z})$

$\phi \in \mathcal{F} = (\hat{P} \times \hat{\Sigma} \times \hat{C} \to \hat{D}_{\tau^1} \times \hat{\Sigma} \times \hat{C}) \times ... \times (\hat{P} \times \hat{\Sigma} \times \hat{C} \to \hat{D}_{\tau^p} \times \hat{\Sigma} \times \hat{C})$

$$\phi = \texttt{fix}\ \lambda F \in \mathcal{F}.(\lambda\hat{\rho} \in \hat{P}.\lambda\hat{\sigma} \in \hat{\Sigma}.\lambda\hat{c} \in \hat{C}.\mathcal{E}[\![fb^1]\!]_{\hat{\rho}\hat{\sigma}\hat{c}Fh^1},$$

$$\vdots$$

$$\lambda\hat{\rho} \in \hat{P}.\lambda\hat{\sigma} \in \hat{\Sigma}.\lambda\hat{c} \in \hat{C}.\mathcal{E}[\![fb^p]\!]_{\hat{\rho}\hat{\sigma}\hat{c}Fh^p})$$

$$\widehat{Value} = \mathcal{P}(\mathbb{Z}) + \mathcal{P}(\mathbb{B}) + \mathcal{P}(Loc)$$

$$\widehat{Object} = \widehat{Arr} + \widehat{Rec}$$

$$\widehat{Arr} = \mathcal{P}(\mathbb{Z}) \times (\mathbb{Z} \to \widehat{Value})$$

$$\widehat{Rec} = String \to \widehat{Value}$$

$$\hat{\ell} \in \mathcal{P}(Loc) = \widehat{Loc}$$

$$\hat{\oplus} : \widehat{Value} \times \widehat{Value} \to \widehat{Value} = \lambda V_1 \in \widehat{Value}.\lambda V_2 \in \widehat{Value}.$$

$$\{v_1 \oplus v_2 \mid v_1 \in V_1 \wedge v_2 \in V_2 \wedge v_1 \neq \text{undefined} \wedge v_2 \neq \text{undefined}\}$$

$\hat{\ominus}, \hat{\cup}$: analogous to $\hat{\oplus}$

$$\hat{.} : \widehat{Loc} \times String \to \widehat{Value} = \lambda\hat{\ell} \in \widehat{Loc}.\lambda f \in String.\{\ell.f \mid \ell \in \hat{\ell} \wedge \ell \neq NULL\}$$

$$[\hat{\ }] : \widehat{Loc} \times \mathcal{P}(\mathbb{Z}) \to \widehat{Value} = \lambda\hat{\ell} \in \widehat{Loc}.\lambda I \in \mathcal{P}(\mathbb{Z}).\{\ell[i] \mid \ell \in \hat{\ell} \wedge i \in I \wedge \ell \neq NULL\}$$

Abstract Interpretation for Expressions

$$\begin{aligned}
\mathcal{E}[\![\Gamma; h; pc \vdash n : \texttt{int}_\perp]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= (\{n\}, \hat{\sigma}, \hat{c}) \\
\mathcal{E}[\![\Gamma; h; pc \vdash true : \texttt{boolean}_\perp]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= (\{T\}, \hat{\sigma}, \hat{c}) \\
\mathcal{E}[\![\Gamma; h; pc \vdash false : \texttt{boolean}_\perp]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= (\{F\}, \hat{\sigma}, \hat{c}) \\
\mathcal{E}[\![\Gamma; h; pc \vdash NULL : \omega^k_\perp]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= (\{NULL\}, \hat{\sigma}, \hat{c}) \\
\mathcal{E}[\![\Gamma; h; pc \vdash x : \Gamma(x)]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= (\hat{\rho}(x), \hat{\sigma}, \hat{c})
\end{aligned}$$

$$
\begin{array}{rcl}
\mathcal{E}[\![\Gamma; h; pc \vdash x.f : \tau_{l \sqcup l'}]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} & = & \texttt{let } (\hat{\ell}, \hat{\sigma}, \hat{c}) = \\
& & \mathcal{E}[\![\Gamma; h; pc \vdash x : \mu\alpha.\texttt{record}_l(k, ..., f : \tau_{l'}, ...)]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \texttt{ in} \\
& & \texttt{let } \hat{\ell}' = \{\ell \in \hat{\ell} \mid \ell \neq NULL\} \texttt{ in} \\
& & \texttt{let } \hat{c}' = \texttt{if } pc \sqcup l = \top \wedge k = \bot \\
& & \qquad \texttt{then } \hat{c} \cup \hat{\ell}' \texttt{ else } \hat{c} \texttt{ in} \\
& & (\cup_{\ell \in \hat{\ell}'} \hat{\sigma}(\ell)(\text{“}f\text{”}), \hat{\sigma}, \hat{c}') \\
\mathcal{E}[\![\Gamma; h; pc \vdash x[y] : \tau_{l \sqcup l_1 \sqcup l_2}]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} & = & \texttt{let } (\hat{\ell}, \hat{\sigma}, \hat{c}) = \\
& & \mathcal{E}[\![\Gamma; h; pc \vdash x : \texttt{array}_{l_1}(k, \tau_l)]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \texttt{ in} \\
& & \texttt{let } \hat{\ell}' = \{\ell \in \hat{\ell} \mid \ell \neq NULL\} \texttt{ in} \\
& & \texttt{let } (\hat{i}, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![\Gamma; h; pc \vdash y : \texttt{int}_{l_2}]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \texttt{ in} \\
& & \texttt{let } \hat{c}' = \texttt{if } pc \sqcup l_1 \sqcup l_2 = \top \wedge k = \bot \\
& & \texttt{then } \hat{c} \hat{\cup} (\hat{\ell}', \hat{i}) \texttt{ else } \hat{c} \texttt{ in} \\
& & (\cup_{\ell \in \hat{\ell}', i \in \hat{i}} (\pi_2\ \hat{\sigma}(\ell))(i), \hat{\sigma}, \hat{c}') \\
\mathcal{E}[\![\Gamma; h; pc \vdash \mathit{length}(x) : \texttt{int}_{l \sqcup l'}]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} & = & \texttt{let } (\hat{\ell}, \hat{\sigma}, \hat{c}) = \\
& & \mathcal{E}[\![\Gamma; h; pc \vdash x : \texttt{array}_l(k, \tau_{l'})]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \texttt{ in} \\
& & \texttt{let } \hat{\ell}' = \{\ell \in \hat{\ell} \mid \ell \neq NULL\} \texttt{ in} \\
& & \texttt{let } \hat{c}' = \texttt{if } pc \sqcup l \sqcup l' = \top \wedge k = \bot \\
& & \texttt{then } \hat{c} \cup \hat{\ell}' \texttt{ else } \hat{c} \texttt{ in} \\
& & (\cup_{\ell \in \hat{\ell}', i \in \hat{i}} \pi_1\ \hat{\sigma}(\ell), \hat{\sigma}, \hat{c}')
\end{array}
$$

$$
\begin{aligned}
\mathcal{E}[\![\Gamma; h; pc \vdash e_1 \oplus e_2 : \tau_l]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= \texttt{let } (\hat{v}_1, \hat{\sigma}_1, \hat{c}_1) = \mathcal{E}[\![\Gamma; h; pc \vdash e_1 : \tau_{1l_1}]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \\
&\quad \texttt{in let } (\hat{v}_2, \hat{\sigma}_2, \hat{c}_2) = \mathcal{E}[\![\Gamma; h; pc \vdash e_2 : \tau_{2l_2}]\!]_{\hat{\rho}\hat{\sigma}_1\hat{c}_1\phi h} \\
&\quad \texttt{in } (\hat{v}_1 \hat{\oplus} \hat{v}_2, \hat{\sigma}_2, \hat{c}_2) \\
\mathcal{E}[\![\ominus e]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= \texttt{let } (\hat{v}, \hat{\sigma}_1, \hat{c}_1) = \mathcal{E}[\![e]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \texttt{ in} \\
&\quad (\hat{\ominus}\hat{v}, \hat{\sigma}_1, \hat{c}_1) \\
\mathcal{E}[\![g^i(se_1, se_2, ..., se_m)]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h'} &= \texttt{let } (\hat{v}_1, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![se_1]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h'} \\
&\quad \texttt{in let } (\hat{v}_2, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![se_2]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h'} \\
&\quad \vdots \\
&\quad \texttt{in let } (\hat{v}_m, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![se_m]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h'} \\
&\quad \texttt{in let } \hat{\rho}' = [x_1 \mapsto \hat{v}_1, x_2 \mapsto \hat{v}_2, ..., x_m \mapsto \hat{v}_m] \\
&\quad \texttt{in } (\pi_i \phi)(\hat{\rho}', \hat{\sigma}, \hat{c}) \\
\mathcal{E}[\![fb]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= \texttt{let } \hat{\rho}' = \hat{\rho}[x_1 \mapsto \{ZERO(\tau_1)\}, ..., x_m \mapsto \{ZERO(\tau_m)\}] \\
&\quad \texttt{in let } (\hat{\rho}'', \hat{\sigma}', \hat{c}') = C[\![s]\!]_{\hat{\rho}'\hat{\sigma}\hat{c}\phi h} \\
&\quad \texttt{in } (\hat{\rho}''(x), \hat{\sigma}', \hat{c}') \\
&\quad \text{where } fb = \tau_{1l_1}\ x_1, \tau_{2l_2}\ x_2, ..., \tau_{ml_m}\ x_m\ ;\ s\ ;\ \texttt{return } x \\
\mathcal{E}[\![\texttt{new } \tau\ se]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= \texttt{let } (\hat{n}, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![se]\!]_{\hat{\rho}\hat{\sigma}\hat{c}} \texttt{ in} \\
&\quad \texttt{let } z = ZERO(\tau) \texttt{ in} \\
&\quad \texttt{let } \ell = \ell_{\tau,h}^{fresh} \texttt{ in} \\
&\quad (\{\ell\}, \hat{\sigma}[\ell \mapsto (\hat{n}, [1 \mapsto \{z\}, 2 \mapsto \{z\}, ..., n \mapsto \{z\}])], \hat{c}) \\
&\quad \text{where } \tau = \texttt{array}_l(k, \tau_{l'}), n = \max(n | n \in \hat{n})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\texttt{new}\ \tau\ \{f_1 = se_1, f_2 = se_2, ..., f_m = se_m\}]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \quad = \quad & \texttt{let}\ (\hat{v}_1, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![se_1]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h}\ \texttt{in} \\
& \texttt{let}\ (\hat{v}_2, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![se_2]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h}\ \texttt{in} \\
& \vdots \\
& \texttt{let}\ (\hat{v}_n, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![se_n]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h}\ \texttt{in} \\
& \texttt{let}\ \ell = \ell^{fresh}_{\tau,h}\ \texttt{in} \\
& (\{\ell\}, \hat{\sigma}[\ell \mapsto [\text{“}f_1\text{”} \mapsto \hat{v}_1, ..., \text{“}f_n\text{”} \mapsto \hat{v}_n]], \hat{c})
\end{aligned}
$$

where $\tau = \texttt{record}_l(k, f_1 : \tau_{1l_1}, ..., f_n : \tau_{nl_n})$

Abstract Interpretation for Statements

$$
\begin{aligned}
C[\![skip]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \quad = \quad & (\hat{\rho}, \hat{\sigma}, \hat{c}) \\
C[\![x = e]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \quad = \quad & \texttt{let}\ (\hat{v}, \hat{\sigma}', \hat{c}') = \mathcal{E}[\![e]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h}\ \texttt{in} \\
& (\hat{\rho}[x \mapsto \hat{v}], \hat{\sigma}', \hat{c}') \\
C[\![\Gamma; h; pc \vdash x.f = e]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \quad = \quad & \texttt{let}\ (\hat{v}, \hat{\sigma}', \hat{c}') = \mathcal{E}[\![e]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h}\ \texttt{in} \\
& \texttt{let}\ (\hat{\ell}, \hat{\sigma}', \hat{c}') = \\
& \mathcal{E}[\![\Gamma; h; pc \vdash x : \mu\alpha.\texttt{record}_l(k, ..., f : \tau_{l'}, ...)]\!]_{\hat{\rho}\hat{\sigma}'\hat{c}'\phi h}\ \texttt{in} \\
& \texttt{let}\ \{\ell_1, \ell_2, ..., \ell_n\}\ \text{s.t.}\ \ell_i \in \hat{\ell} \wedge \ell_i \neq NULL\ \texttt{in} \\
& (\hat{\rho}, \hat{\sigma}'[\ell_1 \mapsto \hat{\sigma}'(\ell_1)[\text{“}f\text{”} \mapsto \hat{v}]][\ell_2 \mapsto \hat{\sigma}'(\ell_2)[\text{“}f\text{”} \mapsto \hat{v}]] \\
& ...[\ell_n \mapsto \hat{\sigma}'(\ell_n)[\text{“}f\text{”} \mapsto \hat{v}]], \hat{c}')
\end{aligned}
$$

$$
\begin{aligned}
C[\![x[y] = e]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= \texttt{let } (\hat{v}, \hat{\sigma}', \hat{c}') = \mathcal{E}[\![e]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \texttt{ in} \\
&\quad \texttt{let } (\hat{\ell}, \hat{\sigma}', \hat{c}') = \\
&\quad \mathcal{E}[\![\Gamma; h; pc \vdash x : \texttt{array}_{l_1}(k, \tau_l)]\!]_{\hat{\rho}\hat{\sigma}'\hat{c}'\phi h} \texttt{ in} \\
&\quad \texttt{let } (\hat{i}, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![\Gamma; h; pc \vdash y : \texttt{int}_{l_2}]\!]_{\hat{\rho}\hat{\sigma}'\hat{c}'\phi h} \texttt{ in} \\
&\quad \texttt{let } \{\ell_1, \ell_2, ..., \ell_n\} \text{ s.t. } \ell_i \in \hat{\ell} \wedge \ell_i \neq NULL \texttt{ in} \\
&\quad \texttt{let } (\hat{n}_i, \hat{a}_i) = \hat{\sigma}'(\ell_i) \texttt{ in} \\
&\quad \texttt{let } \{j_{i1}, j_{i2}, ..., j_{im}\} \text{ s.t. } j_{ik} \in \hat{i} \wedge j_{ik} < max(\hat{n}_i) \texttt{ in} \\
&\quad \texttt{let } \hat{a}_i' = \hat{a}_i[j_{i1} \mapsto \hat{v}][j_{i2} \mapsto \hat{v}]...[j_{im} \mapsto \hat{v}] \texttt{ in} \\
&\quad (\hat{\rho}, \hat{\sigma}'[\ell_i \mapsto (\hat{n}_i, \hat{a}_i')], \hat{c}') \\
C[\![s_1; s_2]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= \texttt{let } (\hat{\rho}', \hat{\sigma}', \hat{c}') = C[\![s_1]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \texttt{ in } C[\![s_2]\!]_{\hat{\rho}'\hat{\sigma}'\hat{c}'\phi h} \\
C[\![\Gamma; h; pc \vdash \texttt{if } x \texttt{ then } s_1 \texttt{ else } s_2]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= \texttt{let } (\hat{v}, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![\Gamma; h; pc \vdash x : \texttt{boolean}_l]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \texttt{ in} \\
&\quad \texttt{if } \hat{v} == \{true\} \texttt{ then } C[\![s_1]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \\
&\quad \texttt{else if } \hat{v} == \{false\} \texttt{ then } C[\![s_2]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \\
&\quad \texttt{else } C[\![s_1]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \sqcup C[\![s_2]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \\
C[\![\Gamma; h; pc \vdash \texttt{while } x \texttt{ do } s]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= \texttt{let } f = (\texttt{fix } \lambda w \in \hat{P} \times \hat{\Sigma} \times \hat{C} \to \hat{P} \times \hat{\Sigma} \times \hat{C}. \\
&\quad \lambda\hat{\rho}' \in \hat{P}.\lambda\hat{\sigma}' \in \hat{\Sigma}.\lambda\hat{c}' \in \hat{C} \\
&\quad \texttt{let } (\hat{v}, \hat{\sigma}', \hat{c}') = \mathcal{E}[\![\Gamma; h; pc \vdash x : \texttt{boolean}_l]\!]_{\hat{\rho}'\hat{\sigma}'\hat{c}'\phi h} \texttt{ in} \\
&\quad \texttt{let } (\hat{\rho}'', \hat{\sigma}'', \hat{c}'') = C[\![s]\!]_{\hat{\rho}'\hat{\sigma}'\hat{c}'\phi h} \texttt{ in} \\
&\quad \texttt{if } v == \{true\} \texttt{ then } w(\hat{\rho}'', \hat{\sigma}'', \hat{c}'') \texttt{ else} \\
&\quad \texttt{if } v == \{false\} \texttt{ then } (\hat{\rho}', \hat{\sigma}', \hat{c}') \texttt{ else} \\
&\quad w(\hat{\rho}'', \hat{\sigma}'', \hat{c}'') \sqcup (\hat{\rho}', \hat{\sigma}', \hat{c}')) \texttt{ in} \\
&\quad f(\hat{\rho}, \hat{\sigma}, \hat{c}) \\
C[\![g(se_1, se_2, ..., se_m)]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} &= \texttt{let } (\hat{v}, \hat{\sigma}', \hat{c}') = \mathcal{E}[\![g(se_1, se_2, ..., se_m)]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \texttt{ in} \\
&\quad (\hat{\rho}, \hat{\sigma}', \hat{c}')
\end{aligned}
$$

Computing the Fixpoint

The two program constructs for which the denotation is not obviously computable are mutually recursive functions and the `while` loop.

Computing the denotation of the `while` loop.

Let $G : L \to L$ where $L = \hat{P} \times \hat{\Sigma} \times \hat{C} \to \hat{P} \times \hat{\Sigma} \times \hat{C}$

be the function whose fixpoint gives us the denotation of the `while` loop. The least upper bound of the following iterative sequence should give us the fixpoint:

$G(\perp_L), G^2(\perp_L), G^3(\perp_L), ...$

where $\perp_L = \lambda\hat{\rho} \in \hat{P}.\lambda\hat{\sigma} \in \hat{\Sigma}.\lambda\hat{c} \in \hat{C}.(\hat{\rho}, \hat{\sigma}, \hat{c})$

However, the sequence might not converge. To make it converge, we can use a widening operator $\nabla : \hat{P} \times \hat{\Sigma} \times \hat{C} \times \hat{P} \times \hat{\Sigma} \times \hat{C} \to \hat{P} \times \hat{\Sigma} \times \hat{C}$ to guarantee that the following sequence will converge to a conservative overapproximation of the fixpoint:

$G(\perp_L),\ G(\perp_L)\nabla G^2(\perp_L),\ G(\perp_L)\nabla G^2(\perp_L)\nabla G^3(\perp_L), ...$

One way of obtaining such a widening operator is to first define a Galois connection between $\widehat{Value}$ and $\widehat{FValue}$ (say, 'Finite Value'), where $\widehat{FValue}$ is coarser than $\widehat{Value}$ and is of finite height. Using α and γ from the Galois connection, the widening operator is given by:

$\hat{v}_1 \nabla_{\widehat{Value}} \hat{v}_2 = \gamma(\alpha(\hat{v}_1) \sqcup \alpha(\hat{v}_2))$

Similarly, we need to define $\widehat{FArr}$, $\widehat{FC}$ and the corresponding $\nabla_{\widehat{Arr}}, \nabla_{\hat{C}}$. We can

then define ∇ above as follows:

$$
\begin{array}{l}
(\hat{\rho}_1, \hat{\sigma}_1, \hat{c}_1)\nabla(\hat{\rho}_2, \hat{\sigma}_2, \hat{c}_2) = \\
\quad (\lambda x : Var.\hat{\rho}_1(x)\nabla_{\widehat{Value}}\hat{\rho}_2(x), \\
\quad \lambda \ell : Loc.\texttt{let}\ \hat{o}_1, \hat{o}_2 = \hat{\sigma}_1(\ell), \hat{\sigma}_2(\ell)\ \texttt{in} \\
\qquad \texttt{case}\ \hat{o}_1, \hat{o}_2\ \texttt{of} \\
\qquad\quad in_1(\hat{a}_1), in_1(\hat{a}_2).\hat{a}_1\nabla_{\widehat{Arr}}\hat{a}_2 \\
\qquad \mid\ in_2(\hat{r}_1), in_2(\hat{r}_2).\lambda f : String.\hat{r}_1(f)\nabla_{\widehat{Value}}\hat{r}_2(f), \\
\quad \hat{c}_1\nabla_{\hat{C}}\hat{c}_2)
\end{array}
$$

We now define $\widehat{FValue}, \widehat{FArr}, \widehat{FC}$ and the corresponding αs and γs.

$$\widehat{FValue} = \mathcal{P}_N(Loc) \times \{PLAINSET, REACHABLE\} + \Omega + \mathcal{P}(\mathbb{B}) + \mathcal{P}_N(\mathbb{Z}) + \mathbb{Z}' \times \mathbb{Z}'$$

$$\alpha : \widehat{Value} \to \widehat{FValue}, \gamma : \widehat{FValue} \to \widehat{Value}$$

Intuitively, we modify the lattice $\widehat{Value}$ as follows, to make it finite in height. We replace $\mathcal{P}(Loc)$ with three possibilities. $\mathcal{P}_N(Loc)$ is the powerset of Loc with N as the upper limit on the size of the sets of elements from Loc. Sets of locations with more than N elements can be represented by a set of seed locations from which the other locations are reachable. Both cases are represented by a set of locations – they are distinguished by using a tag: *PLAINSET* for the former and *REACHABLE* for the latter. If the size of the *REACHABLE* set also grows beyond N, the entire set is just represented by the type of the locations, which are all required to be the same. Ω is the set of all reference types.

The powerset of booleans is already finite and need not be replaced. The powerset of integers $\mathcal{P}(\mathbb{Z})$ is replaced by two possibilities. Again, $\mathcal{P}_N(\mathbb{Z})$ is the powerset of integers with a maximum set size of N. Larger sets are represented

by intervals, which are pairs of elements from $\mathbb{Z}' = \mathbb{Z} \cup \{-\infty, +\infty\}$.

$$\alpha(\hat{v}) = \begin{cases} (\hat{v}, PLAINSET) & \hat{v} \in \mathcal{P}(Loc) \wedge |\hat{v}| \leq N \\ (\hat{\ell}, REACHABLE) & \hat{v} \in \mathcal{P}(Loc) \wedge |\hat{v}| > N \wedge \hat{v} \subseteq \texttt{reachable} - \texttt{from}(\hat{\ell}) \wedge |\hat{\ell}| \leq N \\ \tau & \hat{v} \in \mathcal{P}(Loc) \wedge \forall \ell \in \hat{v}.\omega(\ell) = \tau \wedge |\hat{\ell}| > N \\ \hat{v} & \hat{v} \in \mathcal{P}(\mathbb{B}) \\ \hat{v} & \hat{v} \in \mathcal{P}(\mathbb{Z}) \wedge |\hat{v}| \leq N \\ (\texttt{min}(\hat{v}), \texttt{max}(\hat{v})) & \hat{v} \in \mathcal{P}(\mathbb{Z}) \wedge |\hat{v}| > N \\ (-\infty, +\infty) & \hat{v} = \mathbb{Z} \end{cases}$$

$$\gamma(\hat{v}_F) = \begin{cases} \hat{v} & \hat{v}_F = (\hat{v}, PLAINSET) \\ \texttt{reachable} - \texttt{from}(\hat{\ell}) & \hat{v}_F = (\hat{\ell}, REACHABLE) \\ \{\ell | \omega(\ell) = \tau\} & \hat{v}_F = \tau \\ \hat{v}_F & \hat{v}_F \in \mathcal{P}(\mathbb{B}) \\ \hat{v}_F & \hat{v}_F \in \mathcal{P}(\mathbb{Z}) \\ \{n | n \geq n_1 \wedge n \leq n_2\} & \hat{v}_F = (n_1, n_2) \\ \mathbb{Z} & \hat{v}_F = (-\infty, +\infty) \end{cases}$$

$$\widehat{FArr} = (\mathcal{P}_N(\mathbb{Z}) + \mathbb{Z}^+_M \cup \{+\infty\}) \times (\mathbb{Z}_N \to \widehat{Value} + \widehat{Value})$$

The first element in this pair is the coarse abstraction of $\mathcal{P}(\mathbb{Z})$: It is either a set of possible integers of maximum size n, or a single integer (could be ∞ too) representing the set of all integers up to that integer.

The second element is the coarse abstraction of $\mathbb{Z} \to \widehat{Value}$: It is either a map of integers to abstract values, with n being the maximum element in the the domain of the map. Or it could be a simple abstract value representing the set of all possible values, regardless of index.

$\alpha : \widehat{Arr} \to \widehat{FArr}, \gamma : \widehat{FArr} \to \widehat{Arr}$

$$\alpha((\hat{n}, \hat{a})) = \begin{cases} (\hat{n}, \hat{a}) & \max(\hat{n}) \leq N \\ (\hat{n}, \cup_{i<\max(\hat{n})} \hat{a}(i)) & \max(\hat{n}) > N \wedge |\hat{n}| \leq N \\ (\max(\hat{n}), \cup_{i<\max(\hat{n})} \hat{a}(i)) & \max(\hat{n}) \leq M \wedge |\hat{n}| > N \\ (+\infty, \cup_{i<\max(\hat{n})} \hat{a}(i)) & \max(\hat{n}) > M \wedge |\hat{n}| > N \end{cases}$$

$$\gamma(\hat{a}_F) = \begin{cases} (\hat{n}, \hat{a}) & \hat{a}_F = (\hat{n}, \hat{a}) \\ (\hat{n}, [0 \mapsto \hat{v}, ..., j \mapsto \hat{v}]) & \hat{a}_F = (\hat{n}, \hat{v}) \wedge j = \max(\hat{n}) - 1 \\ (\{i | 0 \leq i < j\}, [0 \mapsto \hat{v}, ..., j \mapsto \hat{v}]) & \hat{a}_F = (j, \hat{v}) \\ (\mathbb{Z}, \lambda i \in \mathbb{Z}.\hat{v}) & \hat{a}_F = (+\infty, \hat{v}) \end{cases}$$

$\widehat{FC} = \mathcal{P}_N(Loc + Loc \times (\mathcal{P}_N(\mathbb{Z}) + \mathbb{Z}^+_M \cup \{+\infty\})) + \mathcal{P}(\Omega)$

$\hat{C}' = \mathcal{P}(Loc + Loc \times \mathcal{P}(\mathbb{Z}))$ (easy to define $\alpha : \hat{C} \to \hat{C}'$ and $\gamma : \hat{C}' \to \hat{C}$)

$\alpha : \hat{C}' \to \widehat{FC}, \gamma : \widehat{FC} \to \hat{C}'$

$\alpha(\hat{c} = \{\ell_1, \ell_2, ..., \ell_k, (\ell_{k+1}, \hat{n}_1), ..., (\ell_{k+m}, \hat{n}_m\}) =$

$$\begin{cases} \hat{c} & m \leq N \wedge \forall i.|\hat{n}_i| \leq N \\ \{\ell_1, \ell_2, ..., \ell_k, (\ell_{k+1}, \hat{n}_1), ..., (\ell_{k+j}, \max(\hat{n}_j)), ..., (\ell_{k+m}, \hat{n}_m\} & m \leq N \wedge \forall i \neq j.|\hat{n}_i| \leq N \wedge \forall j.N < |\hat{n}_j| \leq M \\ \{\ell_1, \ell_2, ..., \ell_k, (\ell_{k+1}, \hat{n}_1), ..., (\ell_{k+j}, +\infty), ..., (\ell_{k+m}, \hat{n}_m\} & m \leq N \wedge \forall i \neq j.|\hat{n}_i| \leq N \wedge \forall j.M < |\hat{n}_j| \\ \hat{\omega} = \{\omega_1, \omega_2, ...\} & m > N \wedge \forall i \leq m.\omega(\ell_i) \in \hat{\omega} \end{cases}$$

$$\gamma(\hat{c}_F) = \begin{cases} \hat{c}_F & \hat{c}_F = \{\ell_1, \ell_2, ..., \ell_k, (\ell_{k+1}, \hat{n}_1), ..., (\ell_{k+m}, \hat{n}_m\} \\ \hat{c}_F - (\ell, j) \cup (\ell, \{i | 0 \leq i \leq j\}) & (\ell, j) \in \hat{c}_F \\ \hat{c}_F - (\ell, +\infty) \cup (\ell, \mathbb{Z}) & (\ell, +\infty) \in \hat{c}_F \\ \{\ell | \omega(\ell) \in \hat{c}_F\} & \hat{c}_F = \{\omega_1, \omega_2, ...\} \end{cases}$$

Computing the function environment.

Again, let $G : L \to L$ where $L = (\hat{P} \times \hat{\Sigma} \times \hat{C} \to \hat{D}_{\tau^1} \times \hat{\Sigma} \times \hat{C}) \times \ldots \times (\hat{P} \times \hat{\Sigma} \times \hat{C} \to \hat{D}_{\tau^p} \times \hat{\Sigma} \times \hat{C})$.

be the function whose fixpoint gives us the function environment. The least upper bound of the following iterative sequence should give us the fixpoint:

$$G(\perp_L), G^2(\perp_L), G^3(\perp_L), \ldots$$

where $\perp_L = (\lambda\hat{\rho} \in \hat{P}.\lambda\hat{\sigma} \in \hat{\Sigma}.\lambda\hat{c} \in \hat{C}.(\{\top_{\tau_1}\}, \hat{\sigma}, \hat{c}),$

$$\vdots$$

$$\lambda\hat{\rho} \in \hat{P}.\lambda\hat{\sigma} \in \hat{\Sigma}.\lambda\hat{c} \in \hat{C}.(\{\top_{\tau_p}\}, \hat{\sigma}, \hat{c}))$$

A widening operator $\nabla : L \times L \to L$ is one that guarantees that the following sequence will converge to an approximation of the fixpoint:

$$G(\perp_L),\ G(\perp_L)\nabla G^2(\perp_L),\ G(\perp_L)\nabla G^2(\perp_L)\nabla G^3(\perp_L), \ldots$$

We use a simple widening operator that limits the call depth to a given parameter k. Mathematically,

$$f\ \nabla G^n(\perp_L) = \begin{cases} G^n(\perp_L) & n < k \\ G^k(\perp_L) & n \geq k \end{cases}$$

We can imagine more sophisticated widening operators that gradually coarsen the results of the functions at successive call depth parameters, instead of a sudden coarsening to $\top$. We leave this to future work.

4.3.3 Interleaved Semantics

We extend the denotational semantics of IMPPAR by adding a concrete cache and interleaving the abstract interpretation to compute the set of objects that will be prefetched into this cache. Figure 4.7 shows a simple strawman interleaving scheme. First the abstract variable environment and the abstract heap are generated from the regular environment and heap by replacing concrete values with abstract values (e.g. a high integer value is replaced with the set of all integers). The abstract interpretation is then applied to the entire program to yield $\hat{c}$, the set of all objects whose accesses can cause a read channel leak. Given these object references, the objects are prefetched into a concrete cache c, using the concrete heap σ. The regular denotational semantics is modified to take in as input the concrete cache as well and use that instead of the heap to fetch the objects that would otherwise cause a read channel leak. The exact set of these objects is identified in the same manner that the abstract interpretation identifies them (i.e. objects with a low access label accessed in a high pc context).

The strawman scheme uses much more cache than necessary. If the two interpretations are interleaved at a finer granularity, cache elements can be evicted as appropriate. Figure 4.8 shows how the finer grained interleaving works. For most program segments, the two interpretations are effectively run side by side without interacting with each other. The only interaction happens when a sensitive program segment (ΔP in the figure) is encountered. A sensitive program segment is one where the pc label transitions from low to high. This can happen with conditionals, loops, dereferences and function calls. For these statements, the abstract interpretation is first executed, using the up to date versions of the abstract environment and heap, but a fresh abstract cache. The resulting ab-

stract cache and the up to date concrete heap are used to compute the concrete cache c so that the regular interpretation would be safe to run. The up to date concrete cache c' can also be used by the prefetcher to prefetch only those objects in $\hat{c}$ that are not already in c', thus saving some network bandwidth. This is secure since the contents of the concrete cache are always low information. In this scheme c replaces c', effectively evicting all objects from c', except those that will be reused in c. This is also secure since only low information is involved. More aggressive cache reuse is possible by retaining more elements from c' and also threading the cache through the entire computation – this is left to future work.

We now formalize these notions by presenting the interleaved semantics as a hybrid of the abstract and regular semantics. The semantics of most program constructs can be expressed as a simple combination of the two semantics as shown in the preliminaries. The semantics for the other program constructs (those involved in sensitive program segments and those that use the cache) follow.

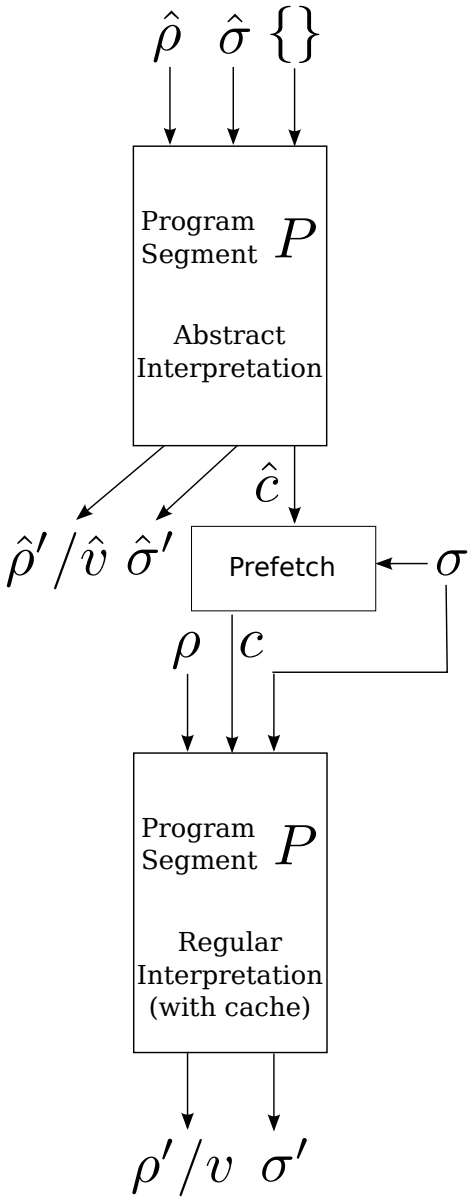

Figure 4.7: A Strawman Interleaved Interpretation

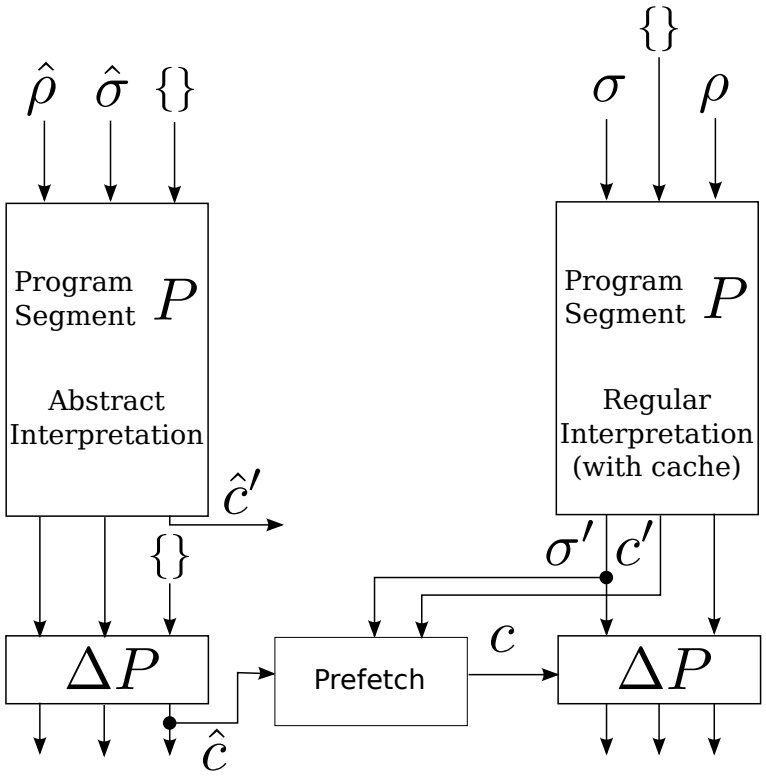

Figure 4.8: A Cache Optimizing Interleaved Interpretation

Preliminaries. (extending from denotational and abstract semantics)

$\mathcal{E}[\![\Gamma; h; pc \vdash e : \tau]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \in (\hat{\Sigma} \times D_\tau \times \Sigma)_\perp$

$C[\![\Gamma; h; pc \vdash s]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \in (\hat{P} \times \hat{\Sigma} \times P \times \Sigma)_\perp$ where

$c \in \Sigma$

$\mathcal{E}[\![\Gamma; h; pc \vdash e : \tau]\!]_{\rho\sigma\phi h} \in (D_\tau \times \Sigma)_\perp$

$C[\![\Gamma; h; pc \vdash s]\!]_{\rho\sigma\phi h} \in (P \times \Sigma)_\perp$

$\mathcal{E}[\![\Gamma; h; pc \vdash e : \tau]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \in \hat{D}_\tau \times \hat{\Sigma} \times \hat{C}$

$C[\![\Gamma; h; pc \vdash s]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h} \in \hat{P} \times \hat{\Sigma} \times \hat{C}$

$\phi \in \mathcal{F} = (\hat{P} \times \hat{\Sigma} \times P \times \Sigma \times \Sigma \to (\hat{\Sigma} \times D_{\tau^1} \times \Sigma)_\perp) \times (\hat{P} \times \hat{\Sigma} \times P \times \Sigma \to (\hat{\Sigma} \times D_{\tau^2} \times \Sigma)_\perp) \times \ldots$

$\times (\hat{P} \times \hat{\Sigma} \times P \times \Sigma \to (\hat{\Sigma} \times D_{\tau^p} \times \Sigma)_\perp)$

$\phi = \texttt{fix}\ \lambda F \in \mathcal{F}.(\lambda\hat{\rho} \in \hat{P}.\lambda\hat{\sigma} \in \hat{\Sigma}.\lambda\rho \in P.\lambda\sigma \in \Sigma.\lambda c \in \Sigma.\mathcal{E}[\![fb^1]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma cFh^1},$

$\vdots$

$\lambda\hat{\rho} \in \hat{P}.\lambda\hat{\sigma} \in \hat{\Sigma}.\lambda\rho \in P.\lambda\sigma \in \Sigma.\lambda c \in \Sigma.\mathcal{E}[\![fb^p]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma cFh^p})$

If $f_e : P \times \Sigma \times \mathcal{H} \to (D_\tau \times \Sigma)_\perp$ and $g_e : \hat{P} \times \hat{\Sigma} \times \hat{C} \times \mathcal{H} \to \hat{D}_\tau \times \hat{\Sigma} \times \hat{C}$

are the denotations for regular semantics of expressions and abstract semantics of expressions respectively, then we can define the denotation

$d_e : \hat{P} \times \hat{\Sigma} \times P \times \Sigma \times \Sigma \times \mathcal{H} \to (\hat{\Sigma} \times D_\tau \times \Sigma \times \Sigma)_\perp$ for the interleaved semantics as follows:

(except for the constructs defined right after, which make use of the cache,

directly or by passing it to subterms)

$d_e = \lambda\hat{\rho}\hat{\sigma}\rho\sigma ch.\texttt{let}\ (\hat{v}, \hat{\sigma}', \hat{c}) = g_e(\hat{\rho}, \hat{\sigma}, \{\})\ \texttt{in}\ \texttt{let}\ (v, \sigma') = f_e(\rho, \sigma)\ \texttt{in}\ (\hat{\sigma}', v, \sigma')$

Similarly,

$d_s = \lambda\hat{\rho}\hat{\sigma}\rho\sigma ch.\texttt{let}\ (\hat{\rho}', \hat{\sigma}', \hat{c}) = g_s(\hat{\rho}, \hat{\sigma}, \{\})\ \texttt{in}\ \texttt{let}\ (\rho', \sigma') = f_s(\rho, \sigma)\ \texttt{in}\ (\hat{\rho}', \hat{\sigma}', \rho', \sigma')$

Interleaved semantics for expressions

$$
\begin{aligned}
\mathcal{E}[\![\Gamma; h; pc \vdash x.f : \tau_{l \sqcup l'}]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \quad = \quad & \texttt{let}\ \lfloor \ell, \sigma \rfloor = \\
& \mathcal{E}[\![\Gamma; h; pc \vdash x : \mu\alpha.\texttt{record}_l(k, ..., f : \tau_{l'}, ...)]\!]_{\rho\sigma\phi h}\ \texttt{in} \\
& \texttt{if}\ \ell = \mathit{NULL}\ \texttt{then}\ \perp\ \texttt{else} \\
& \texttt{let}\ r = \texttt{if}\ pc \sqcup l = \top \wedge k = \perp \\
& \qquad \texttt{then}\ (\texttt{if}\ pc = \perp\ \texttt{then} \\
& \qquad\quad \texttt{prefetch}(\hat{\rho}(x), c, \sigma)(\ell)\ \texttt{else}\ c(\ell)) \\
& \qquad \texttt{else}\ \sigma(\ell)\ \texttt{in}\ \lfloor \hat{\sigma}, r(\text{“}f\text{”}), \sigma \rfloor
\end{aligned}
$$

$$
\begin{array}{rcl}
\mathcal{E}[\![\Gamma; h; pc \vdash x[y] : \tau_{l \sqcup l_1 \sqcup l_2}]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} & = & \texttt{let } \lfloor \ell, \sigma \rfloor = \\
& & \mathcal{E}[\![\Gamma; h; pc \vdash x : \texttt{array}_{l_1}(k, \tau_l)]\!]_{\rho\sigma\phi h} \texttt{ in} \\
& & \texttt{if } \ell = \mathit{NULL} \texttt{ then } \bot \texttt{ else} \\
& & \texttt{let } \lfloor i, \sigma \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash y : \texttt{int}_{l_2}]\!]_{\rho\sigma\phi h} \texttt{ in} \\
& & \texttt{let } (n, a) = \texttt{if } pc \sqcup l_1 \sqcup l_2 = \top \wedge k = \bot \\
& & \qquad \texttt{then if } pc = \bot \texttt{ then} \\
& & \qquad\quad \texttt{prefetch}(\hat{\rho}(x) \times \hat{\rho}(y), c, \sigma)(\ell) \\
& & \qquad\quad \texttt{else } c(\ell) \\
& & \qquad \texttt{else } \sigma(\ell) \texttt{ in} \\
& & \texttt{if } i \geq n \texttt{ then } \bot \texttt{ else } \lfloor \hat{\sigma}, a(i), \sigma \rfloor \\
\mathcal{E}[\![\Gamma; h; pc \vdash \mathit{length}(x) : \texttt{int}_{l \sqcup l'}]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} & = & \texttt{let } \lfloor \ell, \sigma \rfloor = \\
& & \mathcal{E}[\![\Gamma; h; pc \vdash x : \texttt{array}_{l_1}(k, \tau_l)]\!]_{\rho\sigma\phi h} \texttt{ in} \\
& & \texttt{if } \ell = \mathit{NULL} \texttt{ then } \bot \texttt{ else} \\
& & \texttt{let } (n, a) = \texttt{if } pc \sqcup l_1 = \top \wedge k = \bot \texttt{ then} \\
& & \qquad \texttt{if } pc = \bot \texttt{ then} \\
& & \qquad\quad \texttt{prefetch}(\hat{\rho}(x), c, \sigma)(\ell) \\
& & \qquad \texttt{else } c(\ell) \\
& & \texttt{else } \sigma(\ell) \texttt{ in } \lfloor \hat{\sigma}, n, \sigma \rfloor
\end{array}
$$

$$
\begin{aligned}
\mathcal{E}[\![\Gamma; h; pc \vdash e_1 \oplus e_2 : \tau_l]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \;=\;& \texttt{let } \lfloor \hat{\sigma}_1, v_1, \sigma_1 \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash e_1 : \tau_{1 l_1}]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \texttt{ in} \\
& \texttt{let } \lfloor \hat{\sigma}_2, v_2, \sigma_2 \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash e_2 : \tau_{2 l_2}]\!]_{\hat{\rho}\hat{\sigma}_1\rho\sigma_1 c\phi h} \texttt{ in} \\
& \texttt{in if } v_1 \oplus v_2 = \text{undefined } \texttt{then } \bot \\
& \texttt{else } \lfloor \hat{\sigma}_2, v_1 \oplus v_2, \sigma_2 \rfloor \\
\mathcal{E}[\![\ominus e]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \;=\;& \texttt{let } \lfloor \hat{\sigma}_1, v, \sigma_1 \rfloor = \mathcal{E}[\![e]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \texttt{ in} \\
& \texttt{if } \ominus v = \text{undefined } \texttt{then } \bot \texttt{ else } \lfloor \hat{\sigma}_1, \ominus v, \sigma_1 \rfloor \\
\mathcal{E}[\![\Gamma; h; pc \vdash g^i(se_1, se_2, ..., se_m) : \tau_l]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h'} \;=\;& \texttt{let } (\hat{v}, \hat{\sigma}', \hat{c}) = \mathcal{E}[\![g^i(se_1, se_2, ..., se_m)]\!]_{\hat{\rho}\hat{\sigma}\{\}\phi h} \texttt{ in} \\
& \texttt{let } c' = \texttt{if } k = \top \wedge pc = \bot \\
& \qquad\qquad \texttt{then prefetch}(\hat{c}, c, \sigma) \texttt{ else } c \texttt{ in} \\
& \texttt{let } \lfloor v_1, \sigma \rfloor = \mathcal{E}[\![se_1]\!]_{\rho\sigma\phi h'} \texttt{ in} \\
& \texttt{let } \lfloor v_2, \sigma \rfloor = \mathcal{E}[\![se_2]\!]_{\rho\sigma\phi h'} \texttt{ in} \\
& \vdots \\
& \texttt{let } \lfloor v_m, \sigma \rfloor = \mathcal{E}[\![se_m]\!]_{\rho\sigma\phi h'} \texttt{ in} \\
& \texttt{let } \rho' = [x_1 \mapsto v_1, x_2 \mapsto v_2, ..., x_m \mapsto v_m] \texttt{ in} \\
& \texttt{let } (\hat{v}_1, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![se_1]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h'} \\
& \texttt{in let } (\hat{v}_2, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![se_2]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h'} \\
& \vdots \\
& \texttt{in let } (\hat{v}_m, \hat{\sigma}, \hat{c}) = \mathcal{E}[\![se_m]\!]_{\hat{\rho}\hat{\sigma}\hat{c}\phi h'} \\
& \texttt{in let } \hat{\rho}' = [x_1 \mapsto \hat{v}_1, x_2 \mapsto \hat{v}_2, ..., x_m \mapsto \hat{v}_m] \\
& \texttt{in } (\pi_i \phi)(\hat{\rho}', \hat{\sigma}, \rho', \sigma, c')
\end{aligned}
$$

$$
\begin{array}{rcl}
\mathcal{E}[\![fb]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} & = & \texttt{let}\ \rho' = \\
 & & \rho[x_1 \mapsto ZERO(\tau_1), ..., x_m \mapsto ZERO(\tau_m)] \\
 & & \texttt{in let}\ \hat{\rho}' = \\
 & & \hat{\rho}[x_1 \mapsto \{ZERO(\tau_1)\}, ..., x_m \mapsto \{ZERO(\tau_m)\}] \\
 & & \texttt{in let}\ \lfloor \hat{\rho}'', \hat{\sigma}', \rho'', \sigma' \rfloor = C[\![s]\!]_{\hat{\rho}'\hat{\sigma}\rho'\sigma c\phi h} \\
 & & \texttt{in}\ \lfloor \hat{\sigma}', \rho''(x), \sigma' \rfloor \\
\text{where } fb & = & \tau_{1 l_1}\ x_1, \tau_{2 l_2}\ x_2, ..., \tau_{m l_m}\ x_m\ ;\ s\ ;\ \texttt{return}\ x
\end{array}
$$

Interleaved semantics for statements

$$
\begin{array}{rcl}
C[\![x = e]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} & = & \texttt{let}\ (\hat{v}, \hat{\sigma}', \hat{c}) = \mathcal{E}[\![e]\!]_{\hat{\rho}\hat{\sigma}\{\}\phi h}\ \texttt{in} \\
 & & \texttt{let}\ \lfloor \hat{\sigma}', v, \sigma' \rfloor = \mathcal{E}[\![e]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h}\ \texttt{in} \\
 & & \lfloor \hat{\rho}[x \mapsto \hat{v}], \hat{\sigma}', \rho[x \mapsto v], \sigma' \rfloor \\
C[\![\Gamma; h; pc \vdash x.f = e]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} & = & \texttt{let}\ (\hat{\rho}', \hat{\sigma}', \hat{c}') = C[\![x.f = e]\!]_{\hat{\rho}\hat{\sigma}\{\}\phi h}\ \texttt{in} \\
 & & \texttt{let}\ \lfloor \hat{\sigma}'', v, \sigma' \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash e : \tau_{l_2}]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h}\ \texttt{in} \\
 & & \texttt{let}\ \lfloor \ell, \sigma' \rfloor = \\
 & & \mathcal{E}[\![\Gamma; h; pc \vdash x : \mu\alpha.\texttt{record}_{l_1}(k, ..., f : \tau_l, ...)]\!]_{\rho\sigma'\phi h}\ \texttt{in} \\
 & & \texttt{if}\ \ell = NULL\ \texttt{then}\ \bot\ \texttt{else} \\
 & & \lfloor \hat{\rho}', \hat{\sigma}', \rho, \sigma'[\ell \mapsto \sigma'(\ell)[\text{“}f\text{”} \mapsto v]] \rfloor \\
C[\![g(se_1, se_2, ..., se_m)]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} & = & \texttt{let}\ \lfloor \hat{\sigma}', v, \sigma' \rfloor = \mathcal{E}[\![g(se_1, se_2, ..., se_m)]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h}\ \texttt{in} \\
 & & \lfloor \hat{\rho}, \hat{\sigma}', \rho, \sigma' \rfloor
\end{array}
$$

$$
\begin{aligned}
C[\![\Gamma; h; pc \vdash x[y] = e]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \;=\;& \texttt{let } (\hat{\rho}', \hat{\sigma}', \hat{c}') = C[\![x[y] = e]\!]_{\hat{\rho}\hat{\sigma}\{\}\phi h} \texttt{ in} \\
& \texttt{let } \lfloor \hat{\sigma}'', v, \sigma' \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash e]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \texttt{ in} \\
& \texttt{let } \lfloor i, \sigma' \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash y : \texttt{int}_{l_2}]\!]_{\rho\sigma'\phi h} \texttt{ in} \\
& \texttt{let } \lfloor \ell, \sigma' \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash x : \texttt{array}_{l_1}(k, \tau_l)]\!]_{\rho\sigma'\phi h} \texttt{ in} \\
& \texttt{if } \ell = \mathit{NULL} \texttt{ then } \bot \texttt{ else} \\
& \texttt{let } (n, a) = \sigma'(\ell) \texttt{ in} \\
& \texttt{if } i \geq n \texttt{ then } \bot \texttt{ else} \\
& \lfloor \hat{\rho}', \hat{\sigma}', \rho, \sigma'[\ell \mapsto (n, a[i \mapsto v])] \rfloor \\
C[\![s_1; s_2]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \;=\;& \texttt{let } \lfloor \hat{\rho}', \hat{\sigma}', \rho', \sigma' \rfloor = C[\![s_1]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \texttt{ in } C[\![s_2]\!]_{\hat{\rho}'\hat{\sigma}'\rho'\sigma' c\phi h} \\
C[\![\Gamma; h; pc \vdash \texttt{if } x \texttt{ then } s_1 \texttt{ else } s_2]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \;=\;& \texttt{let } (\hat{\rho}', \hat{\sigma}', \hat{c}) = C[\![\texttt{if } x \texttt{ then } s_1 \texttt{ else } s_2]\!]_{\hat{\rho}\hat{\sigma}\{\}\phi h} \texttt{ in} \\
& \texttt{let } c' = \texttt{if } l = \top \wedge pc = \bot \\
& \qquad \texttt{then prefetch}(\hat{c}, c, \sigma) \texttt{ else } c \texttt{ in} \\
& \texttt{let } \lfloor v, \sigma \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash x : \tau_l]\!]_{\rho\sigma\phi h} \texttt{ in} \\
& \texttt{let } (\hat{\rho}'', \hat{\sigma}'', \rho', \sigma') = \texttt{if } v = \mathit{true} \texttt{ then } C[\![s_1]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c'\phi h} \\
& \texttt{else } C[\![s_2]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c'\phi h} \texttt{ in } (\hat{\rho}', \hat{\sigma}', \rho', \sigma') \\
C[\![\Gamma; h; pc \vdash \texttt{while } x \texttt{ do } s]\!]_{\hat{\rho}\hat{\sigma}\rho\sigma c\phi h} \;=\;& \texttt{let } (\hat{\rho}_0, \hat{\sigma}_0, \hat{c}) = C[\![\texttt{while } x \texttt{ do } s]\!]_{\hat{\rho}\hat{\sigma}\{\}\phi h} \texttt{ in} \\
& \texttt{let } c' = \texttt{if } l = \top \wedge pc = \bot \\
& \qquad \texttt{then prefetch}(\hat{c}, c, \sigma) \texttt{ else } c \texttt{ in} \\
& \texttt{let } f = (\texttt{fix } \lambda w : \hat{P} \times \hat{\Sigma} \times P \times \Sigma \to (\hat{P} \times \hat{\Sigma} \times P \times \Sigma)_\bot. \\
& \lambda\hat{\rho}' : \hat{P}.\lambda\hat{\sigma}' : \hat{\Sigma}.\lambda\rho' : P.\lambda\sigma' : \Sigma. \\
& \texttt{let } \lfloor v, \sigma' \rfloor = \mathcal{E}[\![\Gamma; h; pc \vdash x : \tau_l]\!]_{\rho'\sigma'\phi h} \texttt{ in} \\
& \texttt{let } \lfloor \hat{\rho}'', \hat{\sigma}'', \rho'', \sigma'' \rfloor = C[\![s]\!]_{\hat{\rho}'\hat{\sigma}'\rho'\sigma' c'\phi h} \texttt{ in} \\
& \texttt{if } v = \mathit{true} \texttt{ then } w(\hat{\rho}'', \hat{\sigma}'', \rho'', \sigma'') \\
& \qquad \texttt{else } (\hat{\rho}', \hat{\sigma}', \rho', \sigma')) \texttt{ in} \\
& \texttt{let } (\hat{\rho}''', \hat{\sigma}''', \rho''', \sigma''') = f(\hat{\rho}, \hat{\sigma}, \rho, \sigma) \texttt{ in} \\
& (\hat{\rho}_0, \hat{\sigma}_0, \rho''', \sigma''')
\end{aligned}
$$

4.3.4 Evaluation

To evaluate the efficacy of automatic elimination of read channels, we implement the denotational semantics, abstract interpretation and interleaved interpretation and measure the overhead along various dimensions. The interpreters are implemented in Java, totaling 8165 lines of code.

The implementation is tested using three example `IMPPAR` programs. The first is a shop application distributed between the client and the server, implemented in 62 lines of `IMPPAR` code. The client adds items to a shopping cart data structure, maintained locally. It then makes a remote call to the server, which uses secret information to compute a discount that can be offered to the shopper. In the process, a read channel is introduced as the server tries to access the shopping cart on the client. The solution is to prefetch the shopping cart on the server, before beginning the discount computation.

In the second example, implemented in 56 lines of `IMPPAR` code, the server searches for a secret in a list (with 10 elements) stored on the client. This program has a read channel too, since the loop that compares the secret with elements in the list exits as soon as the secret is found. Since the program is public information, the client can learn the secret based on which element in the list was accessed last. The solution is to prefetch the entire list on the server before entering the loop.

The third example, 143 lines of `IMPPAR` code, computes the intersection of two relatively large lists (10 elements each), one on the client and the other on the server. The list on the server has some elements that are secret. The intersection computation happens on the server, and fetches elements from the client

list as necessary. The program first sorts the two lists before computing the intersection. As a result, only one client list element needs to be prefetched at a time, in order to eliminate the read channel.

The tests were run on a 64 bit, 8-core (Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz with hyperthreading) x86 computer, with 8GB of RAM, running Ubuntu Linux version 14.04. Table 4.1 shows the computation time overhead of the abstract and the interleaved interpretation. Similarly, table 4.2 shows the memory overhead of the two interpreters. Table 4.3 shows that an optimal amount of cache was used, suggesting that the just-in-time abstract interpretation is precise.

There are a few ways in which the performance overhead reported here can be significantly reduced. One is to improve on the precision/accuracy of the abstract interpretation so that the computed set of objects to be cached is as close as possible to the minimal set of objects that need to be cached for security. Although for our examples the abstract interpretation is precise enough, greater precision would become necessary for larger example programs. Precision and accuracy can be improved by using more sophisticated/fine-grained lattices for the abstract interpretation. The run-time of the abstract interpretation can also potentially be reduced by caching intermediate results. Since the expressions and statements are evaluated over many possible values, these values are likely to overlap when the evaluation happens in a loop, making the cache useful.

Also, as mentioned in Section 4.3.3, the performance of the interleaved interpretation can be enhanced by aggressive cache reuse and maintaining a single cache threaded through the entire computation. This would significantly reduce the number of fetch requests required in both the Shop and Set Intersec-

tion examples (Table 4.3). More interestingly, the performance of the interleaved interpretation can be further enhanced by using the cache for *all* remote objects, instead of caching only for eliminating read channels. When used as such, care has to be taken that covert channels are not introduced, since now the cache would contain both high and low information. Having separate noninterfering caches for high access label and low access label objects is an option. However, better performance and cache consistency is achieved by using a common cache. For instance, it is safe to store a low access label object in a high access label cache when there is extra space available there. For this performance optimization, eviction of the object must be done carefully so as to not leak information to users of the low access label cache. One way to do this would be to maintain two copies each in one of the caches and maintaining consistency by letting information flow only in one direction. A better way to maintain consistency would be to have a common cache with only a single copy of the object with a label corresponding to the virtual cache(s) it is part of. Storing objects this way also allows *deduplication*, i.e. merging equal or similar objects to save cache space. The labels on the cache objects would become useful while evicting from a deduplicated cache.

The cache labels can also be used to keep track of partial information on whether the object will be accessed in the future. For instance, if a progam analysis reveals that a particular object will not be accessed in a low context, the low label on it can be erased. Of course, an object with no labels can be evicted altogether.

A labeled cache is also useful in aiding information flow tracking that cannot be done statically. For instance, static information flow analyses allow copying

Example	Regular Interpretation	Abstract Interpretation		Interleaved Interpretation	
		Time	Slowdown	Time	Slowdown
Shop	7 ms	13 ms	1.8 x	24 ms	3.4 x
Secret Search	7 ms	41 ms	5.8 x	74 ms	10.5 x
Set Intersection	8 ms	40 ms	5.0 x	109 ms	13.6 x

Table 4.1: Time usage/overhead of the abstract/interleaved interpretations

Example	Regular Interpretation	Abstract Interpretation		Interleaved Interpretation	
		Used	Blowup	Used	Blowup
Shop	1.9 MB	1.9 MB	1.0 x	2.6 MB	1.3 x
Secret Search	1.9 MB	2.6 MB	1.3 x	3.9 MB	2.0 x
Set Intersection	1.9 MB	3.3 MB	1.6 x	27.0 MB	13.6 x

Table 4.2: Memory usage/overhead of the abstract/interleaved interpretations

a low value into a high variable but most of them forget the fact that it was a low value that was copied. A labeled cache can keep track of this fact by saving both the labels on the cached value.

In addition, it is possible to run the abstract interpretation and the regular interpretation in parallel so that the cache is ready when the regular interpretation needs it (optionally parallelizing the abstract interpretation as well for even better performance). Implementing and evaluating these performance optimization ideas is left to future work.

Example	Num. Fetch Requests	Cache used	Minimum Cache needed
Shop	2	1 cache line	1 cache line
Secret Search	10	10 cache lines	10 cache lines
Set Intersection	80	1 cache line	1 cache lines

Table 4.3: Cache space usage/overhead of interleaved interpretation

CHAPTER 5

CONCLUSION

We are increasingly witnessing applications and services that are inherently distributed in nature. The term 'cloud computing' is often used to describe this trend. Such a trend brings up the problem of information security. Safeguarding the privacy and integrity of valuable information from untrusted entities has always been known to be a difficult problem. The trend towards distribution makes the problem even harder.

Information flow control has been shown to provide strong enforcement of confidentiality and integrity of sensitive data. The goal of this work is to show how information flow control can be applied to constructing distributed systems securely.

Web applications are the simplest and most popular instantiations of distributed systems. Chapter 3 demonstrates how Jif can be extended to support development of secure web servlets, resulting in a system called SIF and a newer version of the Jif language: Jif 3.0. At compile time, applications are checked to see if they respect the confidentiality and integrity of information held on the server: confidential information is not released inappropriately to clients, and low-integrity information from clients is not used in high-integrity contexts. SIF tracks information flow both within the handling of a single request, and over multiple requests. Jif 3.0 makes building secure web applications possible by adding sophisticated dynamic mechanisms for access control, authentication, delegation, and principal management, and shows how to integrate these features securely with language-based, largely static, information-flow control.

Chapter 3 shows how the above idea can be extended to include more tiers in the web application, such as persistence and client side code. Both client-side code and server side persistence are indispensable in today's web applications. Section 3.1 discusses SIF-Fabric, which allows tracking information flow through a persistence tier. A distributed airline reservation system is implemented in SIF-Fabric, demonstrating that real web applications can be developed securely, using persistent objects and atomic transactions and that the language of SIF-Fabric is expressive for this purpose. Moreover, in comparison to a similar implementation using a relational database for persistence, the SIF-Fabric implementation is more concise and enforces true end-to-end information flow security.

Section 3.2 in the same chapter presents Swift, which shows how information flow can be tracked through client-side code running as JavaScript. We show that Swift automatically takes care of some awkward tasks: partitioning application functionality across the client–server boundary, and designing protocols for exchanging information. As a result, it is possible to build web applications with active client-side computation, that enforce security by construction, resulting in greater security assurance. Swift satisfies three important goals: enforcement of information security; a dynamic, responsive user interface; and a uniform, general-purpose programming model. No prior system delivers these capabilities. Because web applications are being used for so many important purposes by so many users, better methods are needed for building them securely. Swift appears to be a promising solution to this important problem.

Chapter 4 discusses the problem of read channels, which often arise when building a distributed system with security concerns. The problem is first dis-

cussed in the context of Fabric [44], and a type system is presented which disallows programs with read channels. The experience of the difficulty of programming against this type system led us to design an automatic program transformation that eliminates read channels in a given program by prefetching necessary objects into a cache before entering a sensitive block of code. The set of objects is computed via an abstract interpretation. Preliminary results suggest moderate to high computation overhead and moderate memory overhead. However, compiling the insecure source program down to a secure target program will significantly reduce computation overhead, since the overhead of interpretation will not exist. The automatic approach thus seems promising.

APPENDIX A

DOWNGRADING IN CASE STUDIES

These tables describe the case studies' functional downgrades.

CDIS application

Description	Category
Error composing message. If an error is made when composing a message (e.g., leaving Subject field empty), the user is sent back to message composition. Downgrading this information flow reveals very little about the message data.	Application
Message approval. When a reviewer approves a message, he downgrades his confidentiality restriction. Once all reviewers have approved the message, the recipient may view it.	Application
Database access. Access to the database is done with the authority of the principal `CDISApp`. There are 11 functional downgrades for database accesses, releasing info from `CDISApp` to the user.	Access control
Delegation to `CDISRoot`. All users delegate authority to a root user for the CDIS application, `CDISRoot`, to perform operations that affect all users. This delegation requires user endorsement.	Application

User library

Description	Category
Unsuccessful login. When user enters a password on the login page, he learns if the password was correct. If incorrect, the user is returned to the login page with an error message. This information release about the password is acceptable.	Application
Successful login. When the user logs in successfully, he learns that the password was correct. This information flow is secure.	Application
Delegation to session principal. When the user logs in, he delegates authority to the session principal, using a closure. The decision to authorize the delegation closure must be declassified.	Application
Delegation to session principal. Delegating authority from a newly logged in user to the session principal requires the trust of the user, and thus an endorsement.	Application
Retrieving users from the database. When selecting one or more users, info must be retrieved from the database, and returned to the caller of the Select User(s) page. This transfer requires a total of 3 functional downgrades during user selection.	Access control
Error selecting user(s). A user making an error on the Select User(s) page (e.g., no user id entered) is returned to the Select User(s) page. As this page is a reusable component, its label is set conservatively. A declassification is needed for the error message, from the conservative label to the actual label used for a given page invocation.	Imprecision

Calendar application

Description	Category
Update session state with date to display. The display date must be trusted by the session principal. The date input by the user is trusted by the user, but must be endorsed by the session principal before it's stored in session state.	Access control
Update session state with which user's calendar to display. Similarly, the user selects a user's calendar to display. This downgrade ensures that the session principal authority is required to update session state.	Access control
Fresh id for new event. A new event requires a fresh unique id. The unique id may act as a covert channel, revealing info about the order in which events are created. Since ids are generated randomly, downgrading the fresh id is secure.	Application
Update and retrieve info from database. When info needs to be updated in the database (e.g., edit an event) or retrieved (e.g., fetch user details, or events) information must be transferred between the current user and the application principal `CalApp`. There are 10 such functional downgrades, for different database accesses.	Access control
Go to View/Edit Event page. An event's name is displayed as a hyperlink to the View Event or Edit Event page (depending on user's permissions). Since the link contains the event's name, the info gained by invoking View/Edit Event action is at least as restrictive as the event detail's label. This reveals little about *which* event is being viewed/edited.	Application
Error editing event. The user who makes an error editing an event (e.g., end time before start) is sent back to the Edit Event page. Like the "Go to View/Edit Event" downgrade, this reveals little about the data input.	Application
Changing attendees or viewers of an event. When the user edits an event and changes the attendees or viewers of an event, the labels to enforce on the event time and details change. This requires a downgrade.	Application
Delegation to `CalRoot`. All users delegate their authority to a root user for the calendar application, `CalRoot`, whose authority is needed to perform operations that affect all users. This requires an endorsement from each user.	Application

BIBLIOGRAPHY

[1] Martín Abadi. Secrecy by Typing in Security Protocols. In *Proc. Theoretical Aspects of Computer Software: Third International Conference*, September 1997.

[2] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing Mobile Code Securely With Information Flow Control. In *Proc. IEEE Symposium on Security and Privacy*, pages 191–205, May 2012.

[3] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *ESORICS*, pages 333–348, October 2008.

[4] Aslan Askarov and Andrei Sabelfeld. Security-Typed Languages for Implementation of Cryptographic Protocols: A Case Study. In *Proc. 10th European Symposium on Research in Computer Security (ESORICS)*, number 3679 in Lecture Notes in Computer Science. Springer-Verlag, September 2005.

[5] M. Backes, A. Kate, M. Maffei, and K. Pecina. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 257–271, 2012.

[6] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, April 2003. ISBN 0321136160.

[7] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[8] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as DTIC AD-A023 588.

[9] Hans Bergsten. *JavaServer Pages*. O'Reilly & Associates, Inc., 3rd edition, 2003.

[10] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure Web Applications via Automatic Partitioning. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, October 2007.

[11] Stephen Chong and Andrew C. Myers. Decentralized Robustness. In *Proc. 19th IEEE Computer Security Foundations Workshop*, pages 242–253, July 2006.

[12] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proc. 16th USENIX Security Symp.*, August 2007.

[13] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, November 1998.

[14] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative Analysis of the Leakage of Confidential Data. *Electronic Notes in Theoretical Computer Science*, 59(3):238 – 251, 2002.

[15] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *IEEE Symp. on Computer Security Foundations*, pages 51–65, June 2008.

[16] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *Proc. 5th International Symposium on Formal Methods for Components and Objects*, November 2006.

[17] Thomas A. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[18] Danny Coward and Yutaka Yoshida. Java Servlet Specification, Version 2.4, November 2003. JSR-000154.

[19] Mads Dam and Pablo Giambiagi. Confidentiality for Mobile Code: The Case of a Simple Payment Protocol. In *Proceedings of the 13th IEEE workshop on Computer Security Foundations*, CSFW '00, pages 233–, Washington, DC, USA, 2000. IEEE Computer Society.

[20] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, San Diego, CA, August 2014. USENIX Association.

[21] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Comm. of the ACM*, 19(5):236–243, 1976.

[22] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[23] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[24] U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithm—A Computational Investigation. *Methods and Models of Operations Research (ZOR)*, 33:383–403, 1989.

[25] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate Non-Interference. In *Proc. 15th IEEE Computer Security Foundations Workshop*, pages 1–15, June 2002.

[26] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. In *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, October 2005.

[27] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 4th edition, 2002.

[28] T. Garfinkel, B. Bfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine Based Platform for Trusted Computing. 2003.

[29] Joseph A. Goguen and Jose Meseguer. Security Policies and Security Models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[30] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996.

[31] Google Web Toolkit. `http://code.google.com/webtoolkit/`.

[32] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks. In *Proc. International Conference on Automated Software Engineering (ASE'05)*, pages 174–183, November 2005.

[33] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. From Languages to Systems: Understanding Practical Application Development in

Security-typed Languages. Technical Report NAS-TR-0035-2006, Penn. State Univ., April 2006.

[34] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proc. 13th International World Wide Web Conference (WWW'04)*, pages 40–52, May 2004.

[35] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proc. IEEE Symposium on Security and Privacy*, pages 258–263, May 2006.

[36] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, pages 364–, Washington, DC, USA, 1997. IEEE Computer Society.

[37] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, pages 165–182, October 1991. *Operating System Review*, 253(5).

[38] Butler W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, October 1973.

[39] Lap-chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proc. 22st Annual Computer Security Applications Conference (ACSAC 2006)*, December 2006.

[40] Peeter Laud and Varmo Vene. A type system for computationally secure information flow. In *Proceedings of the 15th International Symposium on Fundamentals of Computational Theory*, pages 365–377, 2005.

[41] Gary T. Leavens, Clyde Ruby, K. Rustan M. Leino, Erik Poll, and Bart Jacobs. JML (poster session): Notations and tools supporting detailed design in Java. In *Addendum 15th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA '00, pages 105–106, New York, NY, USA, 2000. ACM.

[42] Peng Li and Steve Zdancewic. Practical information-flow control in web-based information systems. In *Proc. 18th IEEE Computer Security Foundations Workshop*, pages 2–15, 2005.

[43] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, Owen Arden, Danfeng Zhang, and Andrew C. Myers. Fabric 0.1. Software release, `http://www.cs.cornell.edu/projects/fabric`, September 2010.

[44] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. 22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, 2009.

[45] V. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proc. USENIX Annual Technical Conference*, pages 271–286, August 2005.

[46] Jacob R. Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 199–214, Berkeley, CA, USA, 2013. USENIX Association.

[47] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[48] Gavin Lowe. Quantifying information flow. In *Computer Security Foundations Workshop, 2002. Proceedings. 15th IEEE*, pages 18–31.

[49] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. A High-Performance Oblivious RAM Controller on the Convey HC-2ex Heterogeneous Computing Platform. In *Proceedings of the 3rd Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, CARL'13, December 2013.

[50] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 311–324, New York, NY, USA, 2013. ACM.

[51] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In *Proc. 13th Usenix Security Symposium*, pages 287–302, August 2004.

[52] Pratyusa K. Manadhata and Jeannette M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.

[53] Kevin S. McCurley. Odds and ends from cryptology and computational number theory. In Carl Pomerance, editor, *Cryptology and computational number theory*, volume 42 of *Proceedings of Symposia in Applied Mathematics, AMS Short Course Lecture Notes, Boulder/CO (USA)*, pages 145–166, 1990.

[54] Catherine Meadows and Ira S. Moskowitz. Covert channels - a context-based view. In *Proceedings of the First International Workshop on Information Hiding*, pages 73–93, London, UK, 1996. Springer-Verlag.

[55] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proc. 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, volume 25 of *SIGOPS Operating System Review*, pages 75–84, New York, NY, USA, April 1991. ACM.

[56] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.

[57] Andrew C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachusetts Institute of Technology, Cambridge, MA, January 1999. Ph.D. thesis.

[58] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, 1997.

[59] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[60] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *Proc. 17th IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004.

[61] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.

[62] A. Nguyen-Tuong, S. Guarneri, D. Greene, and D. Evans. Automatically hardening web applications using precise tainting. In *Proc. 20th International Information Security Conference*, pages 372–382, May 2005.

[63] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: an extensible compiler framework for Java. In *Proc. 12th International Conference on Compiler Construction*, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag. LNCS 2622.

[64] Dag Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. *Topics in Cryptology–CT-RSA 2006*, January 2006.

[65] PHP: hypertext processor. `http://www.php.net`.

[66] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, CRYPTO'10, pages 502–519, Berlin, Heidelberg, 2010. Springer-Verlag.

[67] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, April 1979.

[68] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 46–57, 2000.

[69] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[70] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, 1972.

[71] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. 16th ACM Conf. on Computer and Communications Security (CCS)*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[72] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[73] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proc. 18th IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[74] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[75] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2001. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

[76] M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the Web 2.0. In *Proc. 1st Dynamic Languages Symposium*, pages 975–985, October 2006.

[77] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, pages 1–16, October 2005.

[78] Elaine Shi, T.-H.Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with O((logN)3) Worst-Case Cost. In DongHoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214. Springer Berlin Heidelberg, 2011.

[79] Geoffrey Smith and Rafael Alpízar. Secure information flow with random assignment and encryption. In *FMSE '06: Proceedings of the fourth ACM workshop on Formal methods in security*, pages 33–44, 2006.

[80] Guy L. Steele, Jr. RABBIT: A compiler for Scheme. Technical Report AITR-474, MIT AI Laboratory, Cambridge, MA, May 1978.

[81] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 253–267. IEEE, 2013.

[82] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 253–267, Washington, DC, USA, 2013. IEEE Computer Society.

[83] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. *CoRR*, abs/1106.3652, 2011.

[84] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[85] Sun Microsystems. *Java Language Specification*, version 1.0 beta edition, October 1995. Available at `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`.

[86] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *Proc. 19th IEEE Computer Security Foundations Workshop*, pages 202–216, July 2006.

[87] Symantec Internet security threat report, volume IX. Symantec Corporation, March 2006.

[88] Symantec Internet security threat report, volume X. Symantec Corporation, September 2006.

[89] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. The Pragmatic Programmers, 2nd edition, 2004. ISBN 0-974-51405-5.

[90] Trusted Computing Group. *TCG TPM Specification Version 1.2 Revision 94*, March 2006.

[91] Stephen Tse and Steve Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. 14th European Symposium on Programming*, 2005.

[92] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proc. 7th International Joint Conference on the Theory and Practice of Software Development*, pages 607–621, 1997.

[93] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL)*, pages 268–276, Boston, MA, January 2000.

[94] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[95] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. 15th USENIX Security Symp.*, pages 179–192, July 2006.

[96] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proc. 15th USENIX Security Symp.*, pages 121–136, August 2006.

[97] Wei Xu, V.N. Venkatakrishnan, R. Sekar, and I.V. Ramakrishnan. A framework for building privacy-conscious composite web services. In *4th IEEE International Conference on Web Services (ICWS'06)*, September 2006.

[98] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *Proc. 16th International World Wide Web Conference (WWW'07)*, pages 341–350, 2007.

[99] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, and Johannes Gehrke. Hilda: A high-level language for data-driven web applications. In *Proc. 22nd International Conference on Data Engineering (ICDE'06)*, pages 32–43, Washington, DC, USA, 2006. IEEE Computer Society.

[100] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *Proc. 34th ACM Symposium on Principles of Programming Languages (POPL)*, pages 237–249, January 2007.

[101] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.

[102] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. Technical Report 2001–1846, Computer Science Dept., Cornell University, 2001.

[103] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

[104] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proc. 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[105] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2012. To appear.

[106] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 236–250, May 2003.

[107] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd Workshop on Formal Aspects in Security and Trust, IFIP TC1 WG1.7*. Springer, August 2004.

[108] Lantian Zheng and Andrew C. Myers. Making distributed computation trustworthy by construction: Technical report. Technical Report 2006–2040, Cornell University Computing and Information Science, 2006.