THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

The Applicability of Information Flow to Secure Java Development

by
Joseph Spearritt

School of Information Technology and Electrical Engineering,
The University of Queensland.

Submitted for the degree of
Bachelor of Engineering
in the field of Software Engineering.

November 2017

2/15 Dansie Street
Greenslopes, QLD 4120
Tel. 0422 655 246

November 5, 2017

Prof Michael Bruenig
Head of School
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, QLD 4072

Dear Professor Bruenig,

In accordance with the requirements of the degree of Bachelor of Engineering in the division of Software Engineering, I present the following thesis entitled "The Applicability of Information Flow to Secure Java Development". This work was performed under the supervision of Dr Larissa Meinicke & Prof Ian Hayes.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,

Joseph Spearritt.

Acknowledgements

I would like to acknowledge the work and the dedication of my supervisors, Dr Larissa Meinicke and Prof Ian Hayes, in keeping this project on track, and in shaping its final form over our many weekly meetings. I would also like to acknowledge Dr Raghavendra K. R. for his advice throughout the project and for his role in finding this thesis's ultimate direction – something which took several iterations and significant discussion. I'd like to thank Aidan Goldthorpe for offering advice and a second opinion to me at several points throughout the year, even while working on a thesis project of his own, and finally I'd like to thank my peers, my friends and my family for their support over the lifetime of the project.

Abstract

Mechanisms commonly used to control the confidentiality of information within computer programs frequently fail to capture the potential paths by which information may 'leak'. Information Flow security provides a framework for reasoning about the confidentiality properties of a program, and *security typed* programming languages can provide provable guarantees that an application respects a desired confidentiality policy. In this thesis, two such security typed extensions to the Java programming language, Java Information Flow (JIF) and Paragon, are employed to develop three case study applications which showcase common real-world security properties. The languages are then evaluated and compared in terms of the confidentiality guarantees they provide, and the burden they place upon the application developer.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

The vast majority of computer applications and systems used today interact with confidential information, whether in the form of sensitive government or commercial documents, or in the form of customer data. Users rely on these applications to correctly enforce their intended confidentiality policies and thereby keep that data secure. In practice these applications frequently fail to correctly control confidentiality, often as a result of programmer error or of edge cases in usage that were not considered in the program's design.

1.1 Simple, Common Confidentiality Violations

A common class of edge case which typifies this problem is the *account enumeration vulnerability* [1], which frequently appears on even high-traffic, professionally developed websites. An account enumeration vulnerability arises from an often subtle difference in the information presented to a user when their login attempt fails for different reasons. Often, failing to log in will present a message of the form "The username you entered was incorrect", or "The password you entered was incorrect". The intention is that the user should learn no information from failing to log in, except that the credentials they used were incorrect.

However, the messages presented above *do* give the user more information: if they get the message "The password you entered was incorrect", then they know that the username they entered is in fact a registered username, and if they receive the other message, they know it is not. An attacker can use this to try many possible usernames (i.e. to enumerate the valid accounts), making it significantly easier for them to gain unauthorised access to these accounts.

The effects of account enumeration vulnerabilities can be mitigated by limiting how frequently a single visitor may attempt to log into the application, but the vulnerability itself can be easily addressed by ensuring that the error message presented when a username is invalid is the same as that presented when a password is invalid. Yet, this vulnerability is consistently and repeatedly found on websites.

A more specific example of these information edge cases leading to confidentiality violations is a bug found in a well-known system for conference management [2][3]. In this system, papers are submitted to a conference and then either accepted, or rejected by a conference panel. Once accepted, papers are assigned to a particular conference session. Authors of a paper should be able to see which papers they have submitted, but they should not be able to learn whether or not their papers have been accepted before the conference is finalised.

The view which shows the papers an author has submitted contains a column for the session number of a paper, linked to the system's database. If the paper does not have a session yet, this column states "`(not assigned yet)`". But if the paper has been rejected then it no longer exists in the relevant part of the database, and so has no value for this column. An author viewing their papers can determine that any which state "`(not assigned yet)`" under the session allocation column have been accepted, and those which lack any information in that column have been rejected, and thereby the author can learn the status of the paper even when it is intended to be 'pending'.

1.2 Statically-Enforced Information Flow

Confidentiality is most often controlled through the implementation of *access control* mechanisms, which perform runtime checks to determine whether data may be accessed by a particular user or at a particular point in the program's execution. These mechanisms are widely used, but they struggle to protect confidentiality in cases similar to the examples presented above. If the programmer did not realise that conference paper session allocation status could reveal the paper's acceptance status, why would they insert a runtime access check?

The *information flow* mechanisms described in the following sections aim to provide confidentiality guarantees which, for the most part, do not require runtime checks; instead, a static analysis of the program is performed which determines the possible ways information can flow through it. This is checked against a stated policy, and only programs which *provably* enforce their policy correctly will be accepted by the checker. In practice, this is implemented through the use of a *security typing* system, and a violation of the confidentiality policy corresponds to a type-checking compile failure.

1.3 Comparing Security Typed Languages

Two extensions to the Java programming language which implement security typing, Java Information Flow (JIF) [4] and Paragon [5], are examined and compared in the sections below, with the aim of determining whether these languages provide useful and meaningful advantages in maintaining confidentiality from the perspective of an application developer, and what the disadvantages and limitations of these languages are both in terms of failing to produce useful confidentiality guarantees and in terms of the additional burden they place upon a programmer attempting to develop an application.

First, in Chapter 2, the background theory and the literature on information flow, and security typing specifically, is discussed. The policy models and syntax of the JIF and Paragon languages are explained and examined in Chapter 3.

The comparison of the languages is then performed through the process of developing and analysing three case study applications, presented in Chapter 4, which showcase particular security properties: an implementation of the game 'Battleships', a conference management system, and a calendar scheduler application. Finally, Chapter 5 draws conclusions and provides reflection on the project and the final comparison.

Chapter 2

Background & Theory

2.1 Overview

This chapter presents the background theory for information flow security and security typing mechanisms, with reference to the supporting literature. Specifically, it provides context for access control and the model of Mandatory Access Control, the theory for which underpins most security typing implementations, as well as for the widely used Java security model. Information flow security, and its expression through the notion of *non-interference* are explained along with key concepts around the declassification of information. Finally, security typing itself is introduced, as are the policy models for security typing from the literature.

2.2 Security Context

Information security in practice revolves around three key goals colloquially known as the 'CIA Triad' [6]:

- Ensuring information is appropriately *Confidential* (the focus of this thesis)
  - Example: a secret password being leaked is a confidentiality violation
- Ensuring information has *Integrity*
  - Example: a database being accessed and records falsified is an integrity violation
- Ensuring information is *Available*
  - Example: a Denial of Service (DoS) attack causing a website to crash is an availability violation

Most language-based security features focus on the confidentiality and integrity of information; that is, ensuring that secret information remains secret, and that information which needs to be trusted is in fact trustworthy.

Here, systems which aim to ensure confidentiality are examined, though similar mechanisms may also be used to control integrity.

2.3 Access Control

The most common mechanism used in computer systems to ensure confidentiality is *access control*: associating some 'permissions' with users of a system and restricting the actions they are able to perform based on these permissions. The principle of limiting user access is most frequently applied through one of two approaches: *Discretionary* Access Control, or *Mandatory* Access Control.

2.3.1 Discretionary Access Control

Discretionary Access Control (DAC) is the basis of most operating system-level permissions systems, including those in Unix-like operating systems [7], and those implemented in modern versions of Windows.

Central to DAC is the *Access Matrix* [8] – a two-dimensional matrix indexed by *subjects* (users or groups) and by *objects* (documents, files or other information

to be accessed). The values in the matrix determine whether a given subject has permissions to access a particular object, and this is enforced with a run-time check: when the user attempts to access an object they lack permissions for, the system will respond to the failure (often by presenting the user with an error message).

With any non-trivial set of users and documents, the size of the Access Matrix will be prohibitively large, and so it is most commonly implemented using Access Control Lists (ACLs), which are equivalent to columns of the Access Matrix. An ACL is stored alongside each object, indicating which subjects may access it [8].

2.3.2 Limitations of Discretionary Access Control

Access control mechanisms attempt to control confidentiality by inserting checks at boundary points where information flows between subjects.

Consider a scenario in which a user Alice has a diary, which she wishes to keep secret. Alice allows Betty to read her diary, but does not allow a third user, Mal, to read it. Using DAC, an Access Control List for the diary would list Alice and Betty, but not Mal. Then, if Mal attempts to read the diary the access check will fail, preserving confidentiality.

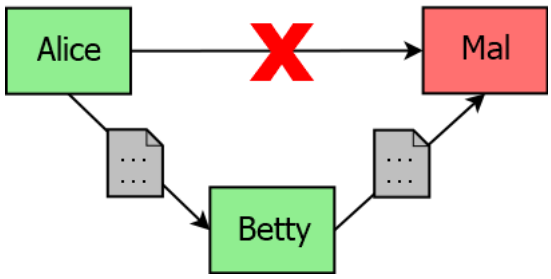

Figure 2.1: Circumventing an Access Control Check

However, suppose Betty now attempts to read Alice's diary. The access check allows this, since it is within policy. Betty reads the diary and makes a copy of it, which she transmits to Mal, as shown in Figure 2.1. Since Betty's copy may have whatever policy she desires, the existing access policy on the diary does not prevent Mal from reading the copy.

The root cause of this failure is that the access control policy merely performs boundary checks, but does not model the *propagation* of information through the system.

2.3.3 Mandatory Access Control

Mandatory Access Control (MAC) systems are most commonly associated with the military, and are generally used in situations where confidentiality is of primary importance. In a MAC system, all data has a *classification*, and users operate with a *clearance* [8].

In the simplest case, the set of classifications is just an ordered list – for instance, Unclassified $<$ Classified $<$ Secret $<$ Top Secret (represented in Figure 2.2 (a)). A user with 'Secret' clearance, then, cannot access or modify 'Top Secret' documents.

More flexible access control policies may be implemented under the Bell-LaPadula lattice model [9], which is at the heart of most modern MAC systems. Under this model, classifications form a *lattice* – a set partially ordered by a flow relation (written $\succeq$) where, for every possible pair of elements, a 'least upper bound' (or 'join', written $\sqcup$) and 'greatest lower bound' (or 'meet', written $\sqcap$) may be found.

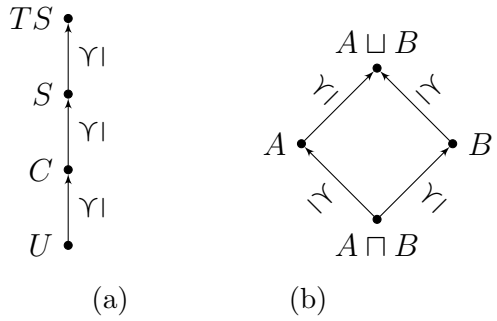

Figure 2.2: Example Bell-LaPadula Model Lattices

In the MAC context, for policies in a lattice structure a user at clearance A can access information at classification B if $A \succeq B$ (which may be read as "B is less restrictive than A"). Given any two classifications A and B, one can find the least restrictive classification which enforces both, $A \sqcup B$, and the most restrictive classification which users at either clearance A or B can read, $A \sqcap B$.

The Bell-LaPadula model may be applied to an access control problem concerning users (or 'subjects') and classified documents (or 'objects') via two rules [10]:

Simple Security Property – 'No Read Up' A subject with clearance C_s may access an object with classification C_o if and only if $C_s \succeq C_o$

*** Property – 'No Write Down'** A subject with clearance C_s may only modify an object with classification C_o if $C_o \succeq C_s$

The Simple Security Property prevents users with insufficient clearance from accessing highly classified data. But the * Property is also important: it states that a user with high clearance cannot modify data at a low classification. For the scenario presented in 2.3.2, this would prevent Betty from creating a copy of the document and sending it to Mal, as Betty may not copy the document to a lower classification. In effect, this introduces a control on the *propagation* of that information.

Lattice-based Mandatory Access Control may be applied to computer systems, just as to Alice's diary or to physical documents. Such systems generally use a 'high water mark' approach [11], allowing programs to begin at low clearance and only moving to higher clearance as needed.

2.4 The Java Security Model

The Java programming language was designed in the context of emerging internet technologies, with the explict use case of users downloading and running 'applets' from web pages [12]. As such, security was a core consideration in Java's design – the original design goals document [13] specifies that "applications written in the Java programming language are secure from intrusion by unauthorized code."

Java is type safe and memory safe, which reduces the possibility of programmer error leading to exploitable flaws in an application. Java's application sandbox takes a three-pronged approach to security in the context of applets which may be downloaded from a remote server, having a Bytecode Verifier, a Class Loader, and a Security Manager [14].

The Bytecode Verifier and the Class Loader concern the loading of new classes into a running Java Virtual Machine (JVM). The Bytecode Verifier analyses the class to ensure that it maintains the class format, does not perform illegal casts, and that it obeys Java's typing rules more generally (e.g. that private methods may not be accessed outside of a class, and that final methods may not be overridden) [15]. The

Class Loader performs the actual loading of a class into the JVM; the 'primordial' Class Loader [14] loads the Java API classes, and other classes may be loaded by a user-specified `ClassLoader` instance.

2.4.1 The Security Manager & Stack Inspection

The final element of the sandbox model is the `SecurityManager` class. When enabled, an instance of `SecurityManager` runs in the JVM, and performs runtime access control: when a potentially sensitive action is undertaken, the Security Manager checks the permissions of the calling class based on a system policy, usually specified by an external policy file. This represents a *discretionary* access control model as per the definition in 2.3.1. If the permission check fails, a `SecurityException` is thrown [16]. A check is typically of the form [17]:

```
1 SecurityManager sm = System.getSecurityManager();
2 Object context = sm == null ? null : sm.getSecurityContext();
3 if (sm != null)
4     sm.checkPermission(new FilePermission("someFile.txt", "read"), context);
```

Permission checks are performed using stack inspection [18]: every frame on the call stack below the sensitive operation is examined, and if *any* frame does not have the required permissions, the check fails. Java provides the `doPrivileged` construct to bypass full stack inspection [18] where this functionality is desired.

2.5 Information Flow Security

Access control mechanisms limit the *permissions* of a subject attempting to access data. Many programs and operating systems make use of Discretionary Access Control. Such access check based mechanisms do not restrict the way in which information *propagates*, leading to circumvention as described in 2.3.2.

The Bell-LaPadula Lattice Model used in MAC may be used to control the *flow* of information rather than merely check for access. The * Property as presented in 2.3.3 provides a primitive form of this: it prevents even those with high clearance from 'leaking' high classification data to a low level, thereby controlling its flow rather than simply performing an access check.

2.6 Formal Non-interference

The traditional MAC model, including controls against information leaks (such as via the * Property), may be implemented in a real system via runtime-enforced access control checks. However, an important question remains: how does the program's author know whether it is implemented *correctly*? Many security issues are the result of unintentional programmer error, and so it is useful to approach information flow security from the perspective of program verification.

This goal can be more clearly defined through the notion of *non-interference*. A program is said to be non-interfering if, given any two program executions which are identical in terms of their low confidentiality input, the results and behaviour of the executions are indistinguishable in terms of their low confidentiality output [19]. Equivalently, a program is non-interfering if there is no *dependency* of low confidentiality data on high confidentiality data [20].

For example, consider a function $f : L \times H \rightarrow L \times H$, where L is the set of integers at low confidentiality and H is the set of integers at high confidentiality, such that $f(l, h) = \langle \frac{l+h}{2}, \frac{l+h}{2} \rangle$. Then f is a function that returns the average of its low and high inputs in both its low and high outputs. This function is interfering: the value of the low output depends on the value of the high input.

Consider instead $g : L \times H \rightarrow L \times H$, where $g(l, h) = \langle l + 5, \frac{l+h}{2} \rangle$; g's high output is the same as that of f, but its low output simply adds 5 to the low input l. This function is *non-interfering*: the low output has no dependency upon the high input.

For programs written in languages without referential transparency [21], like Java, it is difficult to accurately capture a program's entire output as a function of its input – to do so would require accounting for the starting and final states of memory, and for any side effects the program incurs (such as printing the value of a variable to the screen, or to a file). Instead, non-interference checking for these languages considers the flow of information through the system, through assignment to variables and through calls to methods which incur side effects. So, for instance, the following statement with variables *low* and *high* would be invalid:

$$low := high$$

Clearly, assigning the value of a high confidentiality variable to one with low confidentiality violates the policy, and in a non-interference sense allows an attacker with low confidentiality privileges to learn the high confidentiality value.

2.6.1 Implicit Flows

Denning and Denning [22] categorise the flows of information within a program into *explicit* and *implicit* flows. An explicit flow $x \rightarrow y$ is one such as an assignment $y := x$, where the value assigned to a variable y is directly dependent on the value of variable x. An implicit flow $x \rightarrow y$ is then any arbitrary flow $z \rightarrow y$ where z may not be directly related to x, but whether or not the statement is executed depends on the value of x.

A simple example of an implicit flow causing an information leak is one where execution branches based on the value of a high confidentiality variable:

if $high = 5$ **then**
    $low := 1$
else
    $low := 0$
end if

Here, someone with access to the low variable knows that low will only have the value 1 if $high$ has the value 5. After this code executes, they can see that $low = 0$, and therefore $high \neq 5$. They do not know the actual value of $high$ but they have still learned some information about it.

For a program to be non-interfering, a passive attacker must not be able to distinguish *any* information about high confidentiality data; hence implicit flows which allow an attacker to learn high confidentiality information must be prevented [19].

To handle implicit flow of confidentiality via the flow of control, the confidentiality of the execution *context* must be considered. The above snippet's conditional statement represents a *high context* where assignments to low confidentiality variables is impermissible.

2.6.2 Covert Channels & Attacker Model

Consider the following program:

```
low := 0
if high = 5 then
    for i := 1 to 1000000000 do
        skip
    end for
else
    skip
end if
```

By the conventional definitions of information flow, considering both explicit and implicit flows, this program is non-interfering: the value of *low* does not in any way depend on the value of *high*. Yet, someone observing this programs output would be able to learn information about the value of *high*: the program will clearly take significantly longer to execute in the case where $high = 5$.

A program can only be considered secure *with respect to its environment* [19], and the standard construction of non-interference does not consider the timing of a program as a channel for the transmission of information; hence, this kind of information leak is considered a *covert channel*.

Sabelfeld and Myers [19] address a number of possible covert channels which impact on information flow analysis, including timing channels like the above, termination-sensitivity (i.e. whether a program terminates at all), system power usage, and probabilistic channels for programs with stochastic behaviour. Notions of non-interference which examine these channels may be constructed, though doing so will further restrict the potential acceptable programs which may be written.

Hence, proving a program's non-interference under the model presented thus far is not a panacaea: a program's environment and attacker model must still be considered. In some contexts, 'timing-sensitive' non-interference may be necessary, but enforcing that a program will never differ in execution time based on high confidentiality information has quite obvious drawbacks in the performance of a valid program and in the kinds of programs which may be written.

2.7 Enforcement & Security Typing

2.7.1 Enforcement Mechanisms

Enforcement of access control mechanisms is typically done at runtime, as with the Java security model (described in 2.4), by performing checks whenever a potentially sensitive operation is performed. While some implementations of information flow security are also based on runtime checks [23][24][3], treating information flow as a problem of program verification allows for a guarantee of non-interference with no runtime overhead.

2.7.2 Security Typing

In "Certification of programs for secure information flow", Denning and Denning [22] provide a mechanism to verify a program as correctly respecting a lattice-model policy using type checking. All variables are assigned a classification, and the analyser ensures that the policy is not broken under any execution path via a set of type semantics. A language which makes use of type checking-based flow verification is considered a *security typed* language.

The nature of the type semantics used, and hence the complexity of the typing ruleset, depends on the policy model being enforced. Denning and Denning [22] enforce a Bell-LaPadula lattice model policy, and so variable assignments from a variable with a lower security type to a higher security type would fail to compile, as would modification of low confidentiality variables inside a high conditional.

2.7.3 Undecidability in the General Case

The type checking approach to proving non-interference is necessarily conservative: there will always be some programs which are non-interfering which will nonetheless be rejected by the static analysis. This goes beyond being a limitation of current models: determining a condition which is both necessary and sufficient for a program to be non-interfering is an undecidable problem [22][25] – it may be reduced to the halting problem [19].

In practice, systems for statically analysing information flow attempt to ensure that they reject as few *useful* valid programs as is feasible.

2.8 Declassification

A program which is provably non-interfering provides useful guarantees about its behaviour: in the military context for which MAC was designed, it ensures that a program running in a high confidentiality environment cannot declassify information to a lower level. However, in most contexts, requiring strict non-interference prevents useful and necessary programs from being written.

One example of this is a 'password checker' program. The true value of the password is confidential, and the user must provide a correct guess in order to log in. Clearly the operation of the program depends on the value of the high confidentiality password, and a user attempting to log in learns some information about it – even if rejected, the user has learned that the true password is *not* equal to their guess.

A password checker is interfering by design: it has to give feedback on whether the user's guess matches the true password. But this does not violate the desired security properties, since the search space for possible passwords is large, and the user may only guess one at a time.

Rejecting *all* interfering programs makes it very difficult to write even simple programs like a password checker. Hence, security typed languages provide *declassification* mechanisms to allow interference, but only when a programmer explicitly declares it.

2.8.1 Dimensions of Declassification

Mechanisms for declassification may be categorised by the controls they aim to implement. Sabelfeld and Sands [26] provide four dimensions of declassification: *what*, *who*, *where* and *when*.

A *selective* declassification mechanism controls *what* information or *how much* information may be released. An *ownership-based* declassification mechanism regulates *who* may release information by applying access control; this is generally accompanied by a policy model which encodes ownership, such as the Decentralised Label Model discussed in 2.9.2. Models of intransitive non-interference [27] place controls on *where* information may be declassified, requiring that information only be released through some form of sanitising function. Finally, mechanisms may control *when* information may be released – whether relative to other events in the program, or in absolute terms.

2.9 Policy Models for Security Typing

2.9.1 The Denning Lattice Model

Modern implementations of security typing derive primarily from the model proposed by Denning and Denning [22], previously discussed in 2.6. This model assumes that classifications are determined by a central authority; in a security typing implementation, this is the programmer. The lattice model makes it possible to implement both simplistic totally ordered policies and policies with more complex lattice structures. It also allows for the combination of existing policies within the lattice using the join and meet operators.

Selective Declassification Mechanisms

A simple approach to declassification within a standard lattice model is to include a 'declassify' syntax, wherein the classification of a variable may be changed to one of lower confidentiality [28]. This allows for any interfering program to be written (and hence allows the programmer to break the guarantees that the type system would provide), but it requires them to explicitly declare what information they wish to declassify.

Volpano and Smith [29] propose a declassification construct which guarantees information may only flow against the stated policy in worse-than-polynomial runtime complexity. Using this mechanism, the password checker example would be valid, since finding the actual value of the password requires an exponential time brute force checking of all possibilities. Being more restrictive, this approach is more difficult to apply to real world programs.

2.9.2 The Decentralised Label Model

The Decentralised Label Model (DLM) proposed by Myers and Liskov [30] extends the Bell-LaPadula lattice model by removing the need for a central authority which determines all classifications.

Instead, under the DLM subjects (or *principals*) within the system may define their own policies on data, specifying which other subjects *they believe* the data should be able to flow to. These policies still form a lattice, and so the type checker enforces the join of all subjects' policies on the data. This can be used to model

ownership of data, and even systems where subjects do not trust each other to respect confidentiality.

The DLM is discussed in more depth in Chapter 3 as it is the policy framework used by the JIF programming language.

Selective Declassification Mechanism

As with Denning model systems, 'declassify' syntax may be used with the DLM to provide a control for *what* information may be released. However, the DLM also allows for controls on *who* may declassify information: an additional access control check may be used so that a given subject may only declassify policies under their own authority, preventing them from reducing the confidentiality of data which does not 'belong' to them.

2.9.3 The Flow Lock Policy Model

The Denning model uses MAC classification labels, and the DLM extends this with a decentralised model of information ownership. Both models control declassification with the introduction of explicit constructs which declare that an information downgrade is taking place. Broberg and Sands [31] propose an alternative form of policy specification using predicate logic.

Under this model, policies are statements of which subjects may access the data. Policies may specify individual subjects (e.g. a policy stating "Alice may read this information"), or may universally quantify over subjects belonging to some class (e.g. "All subjects who are System Administrators may access this information").

This model is discussed in more depth in Chapter 3 as it forms the basis of policy definition in the Paragon programming language [32].

Dynamic Policy

This model's eponymous 'Flow Lock' system allows it to represent dynamic flow policy – that is, policy which cannot be represented in a static lattice structure. This presents an alternate approach to the declassification problem: policy based on "what" information is declassified (as per the Denning model or the DLM) is viewed as a special case of dynamic policy, as are other models based on "when" or "where" declassification occurs.

A Flow Lock is a runtime boolean value that may be in either an 'open' or 'closed' state. The lock may then guard a flow policy, allowing for policies of the form "Alice may read this information *if* the lock `IsUnlocked` is open". Although locks are runtime values, their values throughout the program are still approximated statically through the type checker. Since locks may change state at runtime, they can be used to not only control *what* information is declassified, but also *when* information is declassified; such time-variant policies cannot be easily encoded in the DLM or in Denning model security type systems.

The Flow Lock concept was extended further in a second paper [33] to include *parametrised* locks, called 'Paralocks'. Parametrised locks allow for policies such as "Alice may read this information if the lock `IsUnlocked(Alice)` is open".

Unary (single argument) locks can then easily represent policies with dynamic *roles*, like "Any employee may access this information if they are currently a manager", built using a quantification over employees and an `IsManager(Employee)` lock. Binary locks, on the other hand, can represent relationships between subjects.

The semantics of Paralocks provide a generalisation of existing policy mechanisms: it is possible to encode the DLM, and hence any policy the DLM can encode, using Paralocks [33].

Chapter 3

JIF and Paragon

The Java Information Flow (JIF) and Paragon programming languages are 'mostly static' information flow (IF) extensions to the Java language which use type systems to enforce information flow constraints.

As articulated by Broberg, Delft, and Sands in "Paragon for Practical Programming with Information-Flow Control" [32], JIF may be considered a 'second-generation Information Flow language'. Its Decentralised Label Model allows for more flexible and more useful policies to be expressed than the Mandatory Access Control lattice model that had been central to most prior work [22], and its implementation as an extension of the popular Java language make it comparatively practical to work with.

Under this taxonomy, Paragon is a 'third-generation Information Flow language', with policy definition built upon first order logic. This abstraction combined with the Paralock construct broadens the scope of what security requirements can be expressed.

3.1 JIF

3.1.1 Relation to Plain Java

JFlow, a Java language extension implementing the Decentralised Label Model (DLM) via type checking was first proposed in "JFlow: Practical Mostly-Static Information Flow Control" [34]. It included DLM-based type labels with support for polymorphism and type inference, while integrating with Java's Object-Oriented programming model – including language features like inheritance and exceptions which introduce potential difficulties for information flow checking.

JIF implements a superset of the JFlow functionality, having added a number of additional features over its development [4]. Since JIF's 1.0.0 version released in 2003 its design is based on Java 1.4, and so it lacks support for a number of more modern Java features.

In general, working with JIF has a number of differences when compared to Java. Some notable ones include:

- JIF places DLM labels on fields, method signatures and potentially local variables
- The special `declassify` statement (and its integrity equivalent, `endorse`) may be used within the code
- *All* exceptions are checked in JIF (including runtime exceptions such as `NullPointerException`)
- When running a JIF program, the JIF runtime must be on the classpath
- Interacting with (non-JIF) Java classes requires writing a JIF interface indicating labels on the class's methods
- Console I/O using stdin / stdout is performed through JIF's runtime
- JIF lacks support for the angle bracket generics syntax introduced in more recent versions of Java

3.1.2 Policy Model

The *decentralised* nature of the DLM is its primary innovation over previous models. Mandatory Access Control requires that each unit of information – such as a variable, or method return value in Java – has a label universally agreed upon

by all users of the system. The DLM instead models 'principals' in the system (which may represent individual users, groups of users or roles), which do not need to trust each other.

The model uses an *acts-for* relation between principals – if principal p acts for principal q, then any action taken by p is assumed to be authorised by q [35]. Principals may also be composed conjunctively as $p\&q$ or disjunctively as p, q. The composite principal $p\&q$ acts for both p and q, while p and q each act for composite principal p, q.

In practice, each principal in JIF is represented by a Java class implementing the `jif.lang.Principal` interface, allowing the programmer to define new principals and specify which other principals they act for.

Confidentiality labels in JIF consist of terms of the form `{o->r1,r2}`, where `o` is the principal expressing ownership over the policy. When evaluated, these labels produce a *reader set* of principals that the owner believes may read the data – in this case, `r1` and `r2` and itself, `o`. The reader sets produced form a lattice, and so labels may be combined via the *join* (least upper bound – "`;`") and *meet* (greatest lower bound – "`meet`") operators, which produce the intersection and union of the constituent labels' reader sets, respectively.

Most commonly, the join operator is used to allow multiple principals to provide differing policies on data. This produces a label of the form:

```
1 {o1->r1,r2; o2->r2,o1,r3}
```

The join of each policy produces a reader set which is the *intersection* of both – so in this case, only `r2` and `o1` (or principals which act for either) may read the information. Note that neither `o1` nor `o2` is the 'true' owner of the data: each is simply expressing what they believe the policy to be.

In addition to producing new labels via the join and meet operators, there exists a top principal (written $*$), which is a principal for which no other principals act, and a bottom principal (written `_`), for which all other principals act. A reader set consisting of the top principal allows *nobody* to read the information, and a reader set consisting of the bottom principal allows *anyone* to read the information.

In practice, a label term of the form `{Alice->*}` indicates that Alice believes that only she should be in the reader set; a label term `{Alice->_}` indicates that Alice believes that everyone should be in the reader set.

The top label `{*->*}` represents the highest possible confidentiality (essentially allowing no principal to read it), while the bottom label `{_->_}` represents the lowest confidentiality; an empty label `{}` is equivalent to the bottom label.

3.1.3 Implicit Flows

Since information may leak to a lower confidentiality implicitly via flow of control, JIF has a program counter or pc label which represents the confidentiality status of the current block of code. For instance, within a branch of an `if` statement with a conditional dependent on a high confidentiality value, the pc label will have high confidentiality.

3.1.4 Label Annotations

The below shows JIF labels on a field (as in `{P->_}` above), indicating that field's confidentiality.

```
1 public String{P->_} myField;
2 public int{A->*} method{B->*}(String{C->*} param) : {D->*}
```

JIF also places labels on method signatures, for their return type (as in `{A->*}` above) and formal parameter types (as in `{C->*}`); a formal parameter's label represents an *upper bound* on the allowed label of its actual argument. The begin label (as in `{B->*}`) states the *required* pc label for calling code, while the end label (as in `{D->*}`) indicates what the pc label will be once the code completes.

Local variables may be annotated with labels, but JIF uses type inference to attempt to determine the correct label for local variables automatically. If labels are omitted on fields or method signatures, a default label is applied: for fields and method end labels this is the bottom or empty label `{}`, while for formal method arguments and method begin labels it is the top label `{*->*}`. For return labels, the default label is the join of all parameter labels and the end label.

3.1.5 Declassification

It is allowable for the label of a value or variable in JIF to change *if* that re-labelling results in a label which is strictly higher confidentiality with respect to the DLM lattice. Hence, restricting readers or joining an additional policy to a label is always acceptable, as is adding a new reader `r` if there already exists a reader in the policy that `r` acts for. For instance, the following is valid:

```
1 int{Alice->Bob,Charles} x = 1;
2 int{Alice->Bob} y = x;
3 int{Alice->Bob,Charles; Bob->*} z = x;
```

However, most useful programs must leak some information to lower confidentiality in order to perform useful action. JIF provides the declassification construct in order to enable this, while still providing some controls: a user must provide *authority* to a method in order to declassify data within it, and they may only declassify policy terms which they own.

```
1 public static void declassifyInt{Alice->*}() where authority(Alice) {
2     int{Alice->*} high;
3     int low{Alice->_} low;
4
5     low = declassify(high, {Alice->*} to {Alice->_});
6 }
```

In the above, the method operates with Alice's authority, and is therefore able to allow the information stored in `high` (which only Alice can read) to move to `low` (which anyone can read).

3.1.6 Sample Class: Diary

The class in Listing 3.1 implements the policy scenario from 2.3.2. Alice has ownership over the diary, and allows Betty to read it. The program will not let Betty propagate access to Mal: a program which did would not compile.

JIF's type system ensures that explicit and implict flows are valid, as in the `readDiary` and `isTheSecret5` methods respectively. It includes the `declassifyDiary` method; only by an explicit call to this method can Mal gain access.

```
1  public class Diary authority (Alice) {
2      private int{Alice->Betty} diaryEntry;
3
4      public Diary() {
5          diaryEntry = 5;
6      }
7
8      public int{Alice->Betty} readDiary() {
9          return diaryEntry;
10     }
11
12     public boolean{Alice->Betty} isTheSecret5() {
13         boolean result;
14         if (diaryEntry == 5) {
15             return true;
16         } else {
17             return false;
18         }
19     }
20
21     public int{Alice->Betty,Mal}
22     declassifyDiary{Alice->Betty,Mal}()
23     where authority(Alice) {
24         int result =
25             declassify(diaryEntry,
26                    {Alice->Betty} to {Alice->Betty,Mal});
27         return result;
28     }
29 }
```

Listing 3.1: JIF Diary Implementation

Compilation Failure in JIF

Consider the modified (and abridged) version of the previous class in Listing 3.2:

```
1  public class DiaryError authority (Alice) {
2      private int{Alice->Betty} diaryEntry;
3
4      public DiaryError() {
5          diaryEntry = 5;
6      }
7
8      public int{Alice->Betty,Mal} readDiary() {
9          return diaryEntry;
10     }
11     ...
12 }
```

Listing 3.2: Erroneous JIF Diary Implementation

The only change from the original `Diary` class is the return label of the `readDiary` method, allowing Mal to read the result of the method call and violate the information flow policy. This causes a compiler error, as per Listing 3.3.

```
1  /home/students/s4359044/thesis/example_printouts/diary/jif/Diary.jif:10:
2      Unsatisfiable constraint
3          general constraint:
4                  rv <= Lrv
5          in this context:
6                  {Alice->Betty; _<-_ ; this ; caller_pc} <= {Alice->Betty,Mal; _<-_;
7      caller_pc}
8          cannot satisfy equation:
9                  {Alice->Betty} <= {Alice->Betty,Mal; _<-_ ; caller_pc}
10         in environment:
11                 {this} <= {caller_pc}
12                 []
13
14     This method may return a value with a more restrictive label than the
15     declared return value label. The declared return type of this method is
16     int{AliceBetty,Mal}. As such, values returned by this method can have a
17     label of at most Lrv.
18                 return diaryEntry;
19                 ^-----------------^
20 1 error.
```

Listing 3.3: Erroneous JIF Diary Implementation: Compiler Message

3.1.7 Integrity Controls

From version 3.0.0, JIF includes a set of integrity controls in addition to the confidentiality controls discussed here. These controls are beyond the scope of this thesis and so will not be examined in-depth. In brief, the integrity lattice is much like the confidentiality lattice, except rather than being ordered by how high the confidentiality is, integrity labels are ordered by how *trusted* they are, and information cannot flow from untrusted to trusted context.

3.1.8 Security Type Polymorphism

Since all data must be labelled (or have a label determined implicitly via type inference), writing implementations of standard data types presents a challenge – it is desirable to avoid having to rewrite the `ArrayList` class for every possible security label the elements of the list may have. To deal with this, JIF introduces security type generics. Since JIF's syntax was designed prior to Java's adoption of generics, square brackets are used to denote type parameters (as opposed to angle brackets).

JIF provides two different kinds of generics: *label* generics and *principal* generics.

A class with a label generic, of the form `TestClass[label L]` may be parametrised by any label. The label type parameter may be used anywhere within the class (e.g. as labels on fields, method return values or begin/end labels). Using this, a data type like `ArrayList[Label L]` may be written such that it can be instantiated with any confidentiality label on the elements.

The second kind of generic is the principal generic, of the form `TestClass[principal P]`. The principal type parameter may be used in labels in the class. This allows for classes which have explicit policies, but which are generic in being 'owned' by any principal.

3.1.9 Runtime Features

Though JIF enforces its information flow policy primarily statically, it does provide runtime features. Labels and principals may be used as first class values throughout code, and so must exist at runtime. First-class (or *dynamic*) labels may be used in specifying other labels and as type parameters, which are still evaluated statically through dependent types. Note that dynamic labels do not,

for the most part, allow for the kinds of 'dynamic policy' that the policy model of Paragon revolves around.

A runtime label value may be constructed with an expression of the form:

```
final label lbl = new label {Alice->Bob};
```

JIF provides runtime testing of both labels and principals. Labels may be compared in a conditional statement of the following form, which will execute the block if `lbl` is of lower or equal confidentiality than `new label {Alice->*}`:

```
if (lbl <= new label {Alice->*}) {
    ...
}
```

Code within the block can then safely assume that `lbl` is of lower or equal confidentiality than `new label {Alice->*}`, and will compile under that assumption.

Similarly, principals may be compared using a conditional statement of the following form, executing the block if `Alice` acts for `Bob`:

```
if (Alice actsfor Bob) {
    ...
}
```

Within the block, it is safe to assume that `Alice` does in fact act for `Bob`.

3.2 Paragon

3.2.1 Relation to Plain Java

Like JIF, Paragon [5] is an extension of the Java language. Unlike JIF, Paragon includes support for a number of more recent Java features – most notably, Java generics. Paragon's own generic features, discussed in more detail in 3.2.6 below, make use of the standard Java angle brackets syntax and hence can be mixed with Java type parameters.

Like JIF, there are a number of differences between writing a program in Paragon and writing one in plain Java, including the following:

- Fields, method parameters and potentially local variables are annotated with read policies
- Locks and policies can be defined through special syntax, and locks can be opened and closed in the code
- Method signatures may include read and write annotations, as well as lock state annotations
- As with JIF, all exceptions are checked in Paragon, though it introduces the `notnull` modifier for variables which can prevent the need to catch `NullPointerException`
- Paragon's runtime must be on the classpath when running the program
- Interacting with plain Java classes requires writing a Paragon interface

3.2.2 Static Policies

A policy in Paragon consists of some number of semicolon-separated *clauses*, each of which consists of a *head* and a *body*. A policy `p` with two clauses would be defined using the following syntax:

```
policy p = {head1: body1 ; head2: body2};
```

The head of a clause specifies an *actor* which may view information under this policy; in a static policy, the body following the colon is empty. Unlike the equivalent principals in JIF, actors in Paragon are simply regular Java objects. So a simple static policy allowing `User`s Alice and Bob to read a file could be encoded as follows [32]:

```
User alice = new User("Alice");
User bob = new User("Bob");
policy aliceAndBob = {alice: ; bob:};
```

Note that in Paragon policies, and unlike those in JIF, the semicolon separation indicates that *both* Alice and Bob may access the data.

Actors in the head of a clause may also be universally quantified over, so in the following, information may flow to any instance of the type `User`:

```
policy allUsers = {User u:};
```

Using this policy definition, a top policy (which disallows all flow) and a bottom policy (which allows all flow) may be encoded:

```
policy top = {:};
policy bottom = {Object x:};
```

These policies form a lattice, and so as with Mandatory Access Control and JIF, they can be combined with least upper bound (join – $*$) and greatest lower bound (meet – $+$) operators:

```
policy charlesOnly = {charles: };
policy bobAndCharles = charlesOnly + {bob: };
// The below allows reading for the union - Alice, Bob and Charles
policy aliceBobAndCharles = aliceAndBob + bobAndCharles;
// The below allows reading for the intersection - only Bob
policy bobOnly = aliceAndBob * bobAndCharles;
```

The tools described here suffice for encoding a MAC-style static information flow policy.

3.2.3 Policy Annotations

Values in Paragon are associated with a policy via policy modifiers. Variables, method return types and parameters can have a *read effect* modifier on them, denoted by the ? symbol, indicating their security policy. For example:

```
?{Object x:} int x;
?somePolicy String y;
?(policy1 * policy2) float z;
public ?p int method(?q int param1, ?r int param2) { ...
```

Read effect modifiers allow the Paragon compiler to analyse the explicit flow of information through a program, ensuring that values may only be assigned to variables with appropriate policies. For example, in the above `x` may only be assigned values at the lowest confidentiality, since its policy allows flow to any object.

The *write effect* modifier for methods is needed to correctly analyse implicit information flows in a program with many methods.

A method's write effect, annotated using the ! symbol, denotes the lowest confidentiality level at which a method's side effects (such as writing to an output or performing different actions based on a high confidentiality conditional) would be observable. For instance, consider the following:

```
?high int secret;
?low int notSecret;

!low void setNotSecret(?low int value) {
    notSecret = value;
}
```

Then, consider the method definition:

```
void method() {
    if (secret > 0) {
        setNotSecret(1);
    } else {
        setNotSecret(0);
    }
}
```

The (low confidentiality) `notSecret` variable is assigned within a context dependent on the value of the high confidentiality `secret` variable, but the compiler can-

not immediately see this, as it occurs inside a method. However, the required `!low` annotation on `setNotSecret` allows the compiler to determine that `setNotSecret` has side effects at a low confidentiality level, and hence that it may not be used within the high conditional – so with this method the code will not compile.

3.2.4 Dynamic Policies with Paralocks

The primary innovation of Paragon's policy model is the 'Flow Lock' or 'ParaLock' construct: a way to build time-variant flow policies using a global state. ParaLocks (or more simply, just 'locks') in Paragon are boolean values which may be *open* or *closed* at a given point in a program's execution, but which are analysed statically in order to allow Paragon to encode policies that cannot be expressed with a static lattice.

A lock in Paragon may be declared only as a field, which is implicitly `static`. The lock may then appear in the body of a Paragon policy clause, denoting that flow to the actor or actors in the head of the clause is conditional upon the lock being open:

```
1 lock myLock;
2 policy p = {Object o: myLock};
```

The `open` and `close` statements may be used to change its state. Dynamic policies (i.e. those with non-empty bodies) are evaluated statically by erasing the locks known to be open. The policy of a given value once open locks are erased is known as its 'effective policy'.

```
1 ?p int x = 5;
2 ?{Object o:} y = x; // Compile error
```

The above will not compile – the policies do not match. However, the below example will compile, since the lock is open and hence `{Object o: myLock}` simply becomes `{Object o: }`.

```
1 ?p int x = 5;
2 ?{Object o: } z;
3 open myLock;
4 z = x; // Compiles successfully
5 close myLock;
```

Locks for Declassification

Locks can quite succinctly model selective declassification, similar to that performed by JIF, by having a declassifying lock which is closed at all times, except during a declassification. The following presents a simple example:

```
lock myLock;
public ?{Object x:} int declassifyValue(?{Object x: myLock} int x) {
    ?{Object x:} result;
    open myLock;
    result = x;
    close myLock;
    return x;
}
```

Values at level `{Object x: myLock}` may then be declassified via this method. In addition, Paragon provides syntactic sugar for this construct of opening a lock, performing some action and then immediately closing it. The following example is equivalent to the first:

```
lock myLock;
public ?{Object x:} int declassifyValue(?{Object x: myLock} int x) {
?{Object x:} result;
open myLock {
    result = x;
}
return x;
}
```

Parametrised Locks

Paragon supports a generalisation over the simple open/closed locks presented above. Locks may be parametrised by actors – Paragon's syntax supports parametrised locks of arity one and two.

Intuitively, parametrised locks may be thought of as *classes* of simple locks, so a lock `myLock(User)` may be open or closed for any given instance of `User`. Alternatively, parametrised locks can be considered as unary and binary relations over actors: so a binary lock `myLock(User, File)` is a relation such that the lock is open if the relation holds for a ⟨`User, File`⟩ pair.

Once declared, a parametrised lock may be used in the body of a policy.

```
1 lock unaryLock(User);
2 lock binaryLock(User, User);
3 policy p1 = {User u: unaryLock(u)};
4 policy p2 = {(User u) User v: binaryLock(u, v)};
```

Note that the use of `(User u)` in the head of policy `p2` indicates quantification, in this case existential. In natural language, `p2` states that: "Information may flow to any User `v` if there exists some `u` such that `binaryLock(u, v)` is open."

Commonly, unary locks are used to model dynamic *roles* – where the information that may flow to an actor depends on their current role or status. Binary locks are often used to model *relationships* between actors [32].

Lock Properties

In order to allow for clearer policy definition and reduce policy-related boilerplate code, Paragon allows for locks to be defined with *lock properties*, representing conditions under which a lock is implicitly open. For instance, on a binary lock, the following lock property could be defined:

```
1 lock myLock(User, User) {(User a, b) myLock(b, a): myLock(b, a)};
```

This states that for users `a` and `b`, `myLock(b, a)` is implicitly open whenever `myLock(b, a)` is open – as a binary relation, `myLock` is symmetric.

Many possible lock properties can be defined depending on the program's overall policy, but Paragon provides shorthand modifiers for symmetry, reflexivity and transitivity, allowing for a binary lock defined as:

```
1 reflexive symmetric transitive lock myLock(User, User);
```

Lock Annotations

Paragon's compiler keeps track of the global lock state statically, which requires statically analysing all possible program flows. Under certain circumstances – especially where lock state changes across many different methods – the compiler cannot track the state.

In order to aid the compiler's analysis, the programmer can add three kinds of lock annotations as modifiers on method signatures: `+myLock`, `-myLock`, `~myLock`.

The `+` lock annotation may be used on methods which *guarantee* they will open a given lock. The `-` annotation may be used on methods which *may* close a given lock. Finally, the `~` modifier indicates that a method may only be called from a context where the given lock is known to be open.

For instance, consider the following method signature:

```
1 public +lock1 -lock2(alice, bob) ~lock3(charles) void method() {
```

This method signature indicates the method *always* opens `lock1`, *may* close `lock2` for the actor pair ⟨`alice, bob`⟩, and may only be called in a context where `lock3` is open for actor `charles`.

One noteworthy limitation in Paragon's lock annotation syntax is that it is not possible to quantify over actors in lock annotations. So, for instance, it is not possible to produce an annotation indicating that a method may close a lock for all actors matching some criteria.

3.2.5 Sample Class: Diary

```
1  public class Diary {
2      public static final User alice = new User("Alice");
3      public static final User betty = new User("Betty");
4      public static final User mal = new User("Mal");
5      private lock declassifier;
6      public static final policy aliceAndBetty = {
7          alice: ; betty: ; mal: declassifier
8      };
9      public static final policy everyone = {
10         alice: ; betty: ; mal:
11     };
12     private ?aliceAndBetty int diaryEntry;
13
14     public Diary() {
15         diaryEntry = 5;
16     }
17
18     public ?aliceAndBetty int readDiary() {
19         return diaryEntry;
20     }
21
22     public ?aliceAndBetty boolean isTheSecret5() {
23         boolean result;
24         return diaryEntry == 5;
25     }
26
27     public ?everyone int declassifyDiary() {
28         int result;
29         open declassifier {
30             result = diaryEntry;
31         };
32         return result;
33     }
34 }
```

Listing 3.4: Paragon Diary Implementation

The implementation in Listing 3.4 is equivalent to the JIF version in Listing 3.1; here the `declassifier` lock is opened briefly in order to allow declassification.

Compilation Failure in Paragon

```
public class DiaryError {
    public static final User alice = new User("Alice");
    public static final User betty = new User("Betty");
    public static final User mal = new User("Mal");
    private lock declassifier;
    public static final policy aliceAndBetty = {
        alice: ; betty: ; mal: declassifier
    };
    public static final policy everyone = {
        alice: ; betty: ; mal:
    };
    private ?aliceAndBetty int diaryEntry;

    public DiaryError() {
        diaryEntry = 5;
    }

    public ?everyone int readDiary() {
        return diaryEntry;
    }
    ...
}
```

Listing 3.5: Erroneous Paragon Diary Implementation

As with the JIF version, modifying the policy on `readDiary` as in Listing 3.5 from its intended `?aliceAndBetty` to `?everyOne` produces a compiler error, stating that the returned information is higher confidentiality than the method's label. Paragon's compiler message in Listing 3.6, is more succinct than JIF's, although Paragon is unable to identify the line on which the error occurs.

```
In DiaryError.para at unknown line:
In the context of class DiaryError:
In the context of method readDiary:
In the context of lock state []:
Returned value has too restrictive policy:
Return expression: diaryEntry
  with policy: { alice#1 :; betty#2 :; mal#3 : declassifier}
Declared policy bound: { alice#1 :; betty#2 :; mal#3 :}
```

Listing 3.6: Erroneous Paragon Diary Implementation: Compiler Message

3.2.6 Security Type Polymorphism

As with JIF, the modularity and reusability of Paragon code is a concern, since the policy definition is embedded with the implementation. Paragon and JIF take the same approach to resolve this issue, relying on security type polymorphism and type parametrisation to make code more generic, and allowing 'interfaces' to be written for interoperability with plain Java code.

The `policyof` Operator

Paragon provides the special `policyof` operator to allow for policy-polymorphic variables and methods.

The special policy denoted by `policyof(this)`, which may be used on methods and `final` fields, is equivalent to JIF's special `{this}` label – the labelled field or method return value takes on the policy of the overall instance. This allows for simple datatypes which are completely polymorphic with respect to policy, such as the following immutable pair definition:

```
class Pair {
    public final ?policyof(this) int x;
    public final ?policyof(this) int y;

    public Pair(?policyof(this) int x, ?policyof(this) int y) {
        this.x = x;
        this.y = y;
    }
}
```

This class can then be used, and the `x` and `y` fields will take on the policy of the instance:

```
?{Object x:} Pair lowPair = new Pair(0, 0);
?{:} Pair highPair = new Pair(0, 0);

// Allowed
?{Object x:} int xFromLow = lowPair.x;
// Compile error
?{Object x:} int xFromHigh = highPair.x;
```

The `policyof` operator may also be used in method return types to allow methods to be polymorphic in the policies of their arguments: `policyof(param)` is

equivalent to using a {param} label in JIF. For example, a simple addition method can be made completely polymorphic in its arguments by joining the policies of its arguments:

```
static ?(policyof(a) * policyof(b)) int add(int a, int b) {
    return a + b;
}
```

Generics

Paragon also provides security type generics, for more complex polymorphic classes. Unlike JIF, Paragon supports Java's native generics functionality, and its security type generics simply extend this. Paragon provides *policy* generics and *actor* generics, equivalent to JIF's label and principal generics respectively.

A generic class in Paragon is declared with the following syntax:

```
public class MyClass<T, policy P, actor A> {
```

Here, T is a regular Java type parameter, P is a policy type parameter and A is an actor type parameter. This class could then be instantiated with:

```
final User alice = new User();
MyClass<String, {Object x:}, alice> instance;
instance = new MyClass<String, {Object x:}, alice>();
```

Policy generics are commonly used to give some internal component of an instance a different but polymorphic policy to the instance itself – for instance, an ArrayList allowing specification of the policy of each element. Actor generics allow for fixed, non-polymorphic policy where the policy may be applied to any actor – for instance, a diary may have a fixed policy (that only its owner may view the contents), but with an actor generic diaries belonging to any person may be constructed.

3.2.7 Runtime Features

As with JIF, Paragon provides some runtime features on top of its static checking. For the most part, Paragon does not need explicit syntax for this, since actors are just objects and policies are values of type `policy`.

Most importantly, Paragon allows for the runtime testing of lock states, by simply using locks as boolean values:

```
1 if (myLock) {
2     ...
3 }
```

In the above, the block will be executed only if the lock `myLock` is open. Hence, any code within the block can safely assume that the lock is open and will compile as such.

3.2.8 Java Interoperability

Paragon provides interoperability with plain Java classes through 'Paragon Interfaces'. Essentially, any Java class may be called from a Paragon class by writing its interface in a `.pi` file with the desired Paragon policy annotations attached.

This is similar to JIF's Java interoperability, although unlike JIF, Paragon's compiler does not compile Paragon source files directly to Java bytecode: the output of `parac` is a generated Java source (`.java` file) and a `.pi` Paragon interface.

Chapter 4

A Comparison of JIF and Paragon

As discussed in Chapter 3, the Java Information Flow (JIF) and Paragon languages take substantially different approaches to expressing security policies. JIF provides a concrete policy framework in the form of the Decentralised Label Model whereas Paragon abstracts over that to allow for logic-based policy expression.

The differences between these languages have an impact on how and where they can be applied to real world programs. In this section the languages are compared in concrete terms through a set of three case study applications, with each demonstrating the applicability or the limitations of each language to a class of problems.

In addition, the process of using these languages for development is described in terms of the additional burden they place on the programmer when compared to writing in standard Java. The practicalities of developing in each are discussed, though as research languages both JIF and Paragon suffer from a lack of documentation, and of 'production-ready' polish.

4.1 Case Study 1: Battleships

4.1.1 Overview

The first case study is an implementation of the well known board game Battleships. The premise of this game is that two opposing players each have a separate 'board', consisting of a grid of (x, y) coordinates. Each player places a set number of ships on the board, each of which covers some number of contiguous coordinates either horizontally or vertically. A player is not allowed to learn the layout of the other's board, but from here players take it in turns to guess (or 'query') coordinates, and the other player must reveal whether or not the guessed coordinate contains a ship – if so, that ship is destroyed. The game continues until one player has no remaining ships on the board.

This is an incomplete information game: the rules require that each player cannot know the layout of the opponent's board, but they must trust the integrity of the information they receive through querying (i.e. it is not permissible to lie about whether a queried position contains a ship).

4.1.2 Key Security Properties

There are two facets to the security policy represented in this case study: the confidentiality and the integrity of each player's board.

Confidentiality of the Board

The confidentiality policy is simply stated: a player's board is secret, and information about the positions of ships on the board may flow to the player owning the board and no-one else. The exception to this is the querying process: an opponent may, on their 'turn' in the game, query a coordinate and learn whether or not a ship overlaps the given point.

This exception represents a declassification of some information regarding the location of ships on the board (after all, the opponent could gain perfect information by simply querying every point on the board), but it restricts the *quantity* of information which is leaked to one coordinate and hence at most one ship per query, and it ensures that the information may be declassified through this method and no other.

Integrity of the Board

A corollary to the above confidentiality policy is that, in order for the game to proceed according to the rules, the player must trust their opponent to reveal the truth about a queried coordinate. In addition, a player must not be allowed to manipulate or falsify the other player's board.

4.1.3 Implementation Structure

Unlike the other case studies presented here, a JIF implementation of Battleships already exists – it comes standard with a download of the compiler [4]. This implementation not only provides the confidentiality policy described above, but also the integrity policy. Though this is relevant to the overall problem, expression of confidentiality policies is the main focus of this comparison and so the integrity policy is not examined in depth.

In order to provide an effective comparison, the Paragon implementation was developed using the same basic structure as the JIF implementation – this pattern is repeated for the other case studies, where a common code structure is used for both JIF and Paragon implementations.

This structure is uses the following Java / JIF / Paragon classes:

- `Coordinate`: a simple immutable representation of an (x, y) pair
  - Polymorphic with respect to security policy (i.e. it has no inherent policy, but a policy may be placed upon a coordinate)
- `Ship`: an immutable single ship in the game
  - Has a length, a Coordinate for the bottom left point, and a flag indicating whether the ship is aligned horizontally or vertically
  - Provides methods to check whether a given Coordinate is covered by the ship, and whether another given ship intersects this one
  - Polymorphic with respect to security policy
- `Board`: a board of coordinates containing a list of Ships
  - Provides a method to test whether a given Coordinate contains a Ship
  - Polymorphic in terms of policy: all ships take on the Board's policy
- `Player`: a player in the game, with their own secret Board
  - On creation, the player initialises its board with ship positions
  - Provides methods to generate new queries, and to receive and answer queries from the opponent
  - Defines the policy on its board: no-one but the player may view its board, except through its `processQuery` method (which declassifies information about the queried point)
- `BattleShip`: coordinates the gameplay between two players
  - Provides the method which runs the game logic
- `Main`: provides the program's main method

4.1.4 JIF Implementation

A condensed code printout, containing critical sections of the JIF implementation for Battleships, is listed in Appendix A.1.1.

Impact of Integrity Policy

As per 4.1.3, the existing JIF implementation, developed by the creators of JIF itself, includes an implementation of not only the game's confidentiality policy but also its integrity policy.

The inclusion of this policy significantly increases the complexity of the code: it requires the addition of a number of methods to allow for the 'endorsement' of information to a higher level of integrity, with method signatures such as the following from the `Board` class:

```
1 public Board[{p->*; p<-* meet o<-*}]{p<-* meet o<-*}
2                     endorseBoard{p<-* meet o<-*}
3             (principal{p<-* meet o<-*} p, principal{p<-* meet o<-*} o)
4 where {L} equiv {p->*; p<-*}, caller(p,o) {
```

These method signatures are not particularly readable, and when combined as in other areas with JIF's confidentiality labeling system, it can become very difficult to discern even what the return type and parameters of a method mean.

The integrity policy and its implications are not considered more generally here – for the purposes of this comparison, it is assumed that preventing the downward flow of information is the goal of developing with a security-type system, as opposed to preventing the upward flow of 'trust' or integrity.

Confidentiality Policy

In contrast with the integrity policy, the implementation of the confidentiality of ship locations is relatively simple.

The `Player` class has two principal type parameters `P` and `O`, specifying the principals acting as this player and their opponent. The `Board` class takes one label type parameter, specifying the policy on its ships, and hence each Player's Board is parametrised by the policy label `{P->*}`. This simply states that the Board belongs to `P`, and `P` allows this information to be viewed only by the top principal.

This policy prevents the opposing player (principal `O`) from accessing the informa-

tion. Only through a declassification statement may information about the state of the board flow to another principal.

The `processQuery` method is the only method which allows for this, through the following two lines (which run after various input checking conditions):

```
1  // find the result.
2  boolean result = brd.testPosition(query, new label {P<-* meet O<-*});
3  // declassify the result
4  return declassify(result, {P->*;P<-* meet O<-*} to {P<-* meet O<-*});
```

The result is calculated by querying the board. The `result` variable is inferred by the type system to be of confidentiality policy `{P->*}`, and so the `declassify` statement is used to declassify it to the empty confidentiality policy `{}` (equivalent to `{_->_}`). Hence, the return value of this method may flow to any principal.

The basis of this policy is quite simple, though it does introduce annotation burden throughout the application. JIF's policy model is able to quite effectively represent it, as the 'players' intuitively align with statically determined principals, and the query construct is naturally represented with a declassification.

Boilerplate & General Details

In addition to the annotations and additional code required to define the security policy, JIF's design requires further 'boilerplate' which is necessary for any program that takes input or produces output. In the case of Battleship, the program outputs information to standard out, which is external to the program and hence lacks confidentiality controls. The boilerplate code (from the Battleship example) illustrating this is as follows:

```
1  PrintStream[{}] out = null;
2  try {
3      Runtime[p] runtime = Runtime[p].getRuntime();
4      out = runtime==null?null:runtime.stdout(new label {});
5  }
6  catch (SecurityException e) {
7      // just let out be null.
8  }
9
10 // the PrintStream needs to be labeled {Alice<-* meet Bob<-*}, which
11 // requires a declassification and an endorsement.
12 PrintStream[{}] out1 = endorse(out,
13                              {p->*} to {p->*;Alice<-* meet Bob<-* meet p<-*});
14 PrintStream[{}] out2 = declassify(out1, {Alice<-* meet Bob<-*});
```

In short, this code represents the following steps:

1. Get a reference to the JIF `Runtime` for the program's calling principal (`p`)
2. Attempt to get a reference to an unclassified standard out stream
3. The stream itself has no integrity, so it requires an endorsement
4. Though the stream is parametrised by the empty label, the stream *itself* is confidential to the calling principal, requiring a declassification

This process adds a significant burden on the programmer, since making use of standard input and output streams is required for most useful applications. Since I/O streams are at the boundary of the system, once information flows to them the flow policy is no longer enforced; hence it is sensible to restrict them.

Nonetheless, the implementation requires significant boilerplate, and if the user attempts to construct a stream that belongs to any principal *other than* the calling principal, the code will compile but will fail silently at runtime, which is problematic from a practical standpoint.

4.1.5 Paragon Implementation

A condensed code printout, containing critical sections of the Paragon implementation for Battleships, is listed in Appendix A.1.2.

Data Types

The Paragon version of Battleships is based on the same overall structure as the JIF implementation. The first notable difference, seen in the simple data type classes `Coordinate` and `Ship`, is that Paragon has the special `policyof(this)` annotation, allowing these classes to be fully polymorphic in policy without being explicitly parametrised. Though use of the `policyof(this)` annotation is somewhat limited in practice – it is available only for methods and `final` fields – it greatly simplifies the implementation of simple data type classes.

In addition, defining the overridden Java methods `equals`, `hashCode` and `toString` for data types requires substantially less effort in Paragon than in JIF. JIF uses its own wrappers (`JIFObject` and descendents), whereas in Paragon, the `policyof(arg)` operator is simply used to allow for polymorphic policy on the standard Java forms of these methods.

The difference to annotation burden and readability can be easily seen in the method signatures of the JIF and Paragon equals methods respectively:

JIF:

```
1 public boolean{lbl;*lbl;L;o} equals(label lbl, IDComparable[lbl] o) {
```

Paragon:

```
1 public ?(policyof(this) * policyof(o)) boolean equals(Object o) {
```

Confidentiality Policy

As the focus of this comparison is on the use of information flow with respect to confidentiality, the integrity portion of the JIF program's design was not carried over into the Paragon version: hence, the Paragon implementation assumes the trustworthiness of the opponent's response to queries.

Though the Battleships game can quite easily be thought of in terms of interacting actors as under JIF's Decentralised Label Model, for the Paragon implementation modeling this was unnecessary. Paragon's 'Objects as Actors' approach allows the `Player` instance itself to be used as the actor in the security policy, without the

need for an external, static principal. The secrecy of a player's board is determined solely by the following policy definition:

```
public final Player self = (Player)this;
public final policy boardPolicy = {
      self :
    ; Object o : allowBoardAccess(self)
};
```

The first clause in the policy (`{self : }`) states that only the current instance may view this information. The `self` variable as opposed to the `this` keyword is necessitated by limitations of the Paragon compiler.

The second clause of the policy allows for the declassification via the query system. `allowBoardAccess(self)` is a unary paralock, taking a player as its argument. The policy clause states that any Object `o` may access the board only if the `allowBoardAccess` lock is open for the current player instance.

The core of this approach is that the lock remains closed at all times, *except* briefly during the declassification process. While the lock is open, the board policy reduces to `{Object o: }`, and at this point the required information is copied into a low confidentiality variable. Since this pattern is common (especially for implementations ported from JIF, where this form of declassification is the *only* dynamic policy mechanism available), Paragon provides a special declassification block construct, as seen in the code below:

```
?boardPolicy boolean res = this.board.testPosition(query);
?bottom boolean declassified = false;

// Briefly open the lock to declassify the result
open allowBoardAccess(self) {
    declassified = res;
}

return declassified;
```

Inside the scope of the braces, the lock is open, but it is closed everywhere else. Information can *only* leak inside of this block, and the compiler will prevent it from flowing down at any other point in the program.

While this is functionally equivalent to making use of a `declassify` expression in JIF, the Paragon version is significantly more readable and makes the scope of the intentional information leak more clear.

Boilerplate & General Details

Paragon does not require boilerplate to make use of standard output streams: there is a provided Paragon interface file for the Java `PrintStream` class, which has a policy type parameter. It suffices to simply use `System.out` as a `PrintStream<{Object x:}>` (i.e. a stream with no confidentiality). Building output streams with more complex policies requires more work, but is not necessary for this case study.

As a result of the conciseness of the policy definition and the lack of boilerplate, the Paragon implementation of Battleships quite closely resembles a plain Java implementation, whereas the JIF implementation has large numbers of annotations on each function as well as additional steps and boilerplate code required to perform simple Java tasks. It is worth noting, however, that some of this need for extra annotation may be attributed to the inclusion of the integrity policy.

4.1.6 Conclusion

The Battleships examples can be represented under the policy models of both JIF and Paragon with relative ease. The interacting principals of the Decentralised Label Model can be adapted to the problem simply by mapping a principal to each player of the game. In the case of Paragon, these interacting actors may be implicitly modelled by the `Player` instances themselves.

The declassification action required to allow for querying of the board aligns well with JIF's declassification construct: the query is an 'escape hatch' for the confidentiality policy, and the `declassify` statement is an 'escape hatch' from the security type system. Paragon's system is more general than this, but the use of a briefly opened paralock to approximate declassification is common enough that Paragon includes syntactic sugar for it. Hence, in both languages this is quite easily expressed.

The largest difference between the two implementations lies in how Paragon's syntax reduces the need for complex boilerplate code. Using standard output, necessary for virtually all useful programs, requires non-trivial setup in JIF and none in Paragon. In both languages, the `{this}` and `policyof()` constructs can eliminate the need for security type parametrisation in a number of common scenarios. When Paragon does use security type generics, it extends the standard Java generics syntax where JIF uses entirely different syntax incompatible with Java generics.

Hence, while this example can be equivalently represented using both JIF and Paragon, Paragon's syntax produces a more concise and more readable program.

4.2 Case Study 2: Conference Management System

4.2.1 Overview

The second case study is a simplified Conference Management system. Papers have a title and list of authors; these papers may then be submitted to the conference. Some submitted papers are then accepted and allocated to a conference session, in secret. Then at some point in time the final conference schedule is made available, and previously secret information about which papers were accepted becomes publicly available.

In addition, until the release of the schedule by the conference organiser, the submitted papers' titles are publicly viewable, but their authors lists are secret (this represents a policy for blind review).

4.2.2 Key Security Properties

There are essentially two parts to the security policy required for this program, which both derive from essentially the same principle of 'timed release':

1. Until the timed release, whether a paper has been accepted or not may be known only to the conference organiser
2. Until the timed release, the list of authors of a paper is not available to anyone except the conference organiser

Timed Release

Timed release requires that confidentiality be time-variant – equivalent pieces of code may be acceptable or unacceptable depending on *when* they are called.

This is an inherently dynamic policy: it cannot be represented in a static lattice structure. Hence, its representation requires either some notion of declassification so that information can be allowed to 'escape' from a lattice policy structure, or alternately some way of representing policy state directly. As will be discussed in the following sections, the former method is problematic as declassification bypasses the security type system entirely, preventing the type system from properly enforcing the conditions of the timed release.

Paper Acceptance

The first aspect of the policy, that a paper's acceptance status remain hidden until release, provides an illustration of how implicit information flows must be taken into consideration. It is relatively straightforward to prevent an explicit leak of this information (for example, from a method that returns whether or not the paper has been accepted), but in order to prevent *all* flow, the security type system must also prevent any information whose value depends on the acceptance status from being leaked.

In the example application, this requirement is demonstrated through the ability for users to retrieve the session allocation for a given paper. Since any paper which has been allocated to a session must have been accepted, leaking the session allocation implies a leak of the acceptance status. This precise issue occurred in the EDAS conference management system [2], and is discussed in relation to an alternative approach to information flow security by Polikarpova, Yang, Itzhaky, *et al.* [3].

Hence, the session allocation for a given paper must also be guarded at the same level of confidentiality as its acceptance status.

Author Confidentiality

The other aspect of the policy is more straightforward: until the timed release, papers' authors must be hidden from everybody except the conference organiser. This is novel when compared to the Battleships example in that the papers themselves — their title, abstract and contents — are not secret; so only *some* information about the paper need be constrained by the timed release policy.

4.2.3 Implementation Structure

The JIF and Paragon implementations for the Conference Management case study were written to follow the same class structure. Each implementation attempts to provide essentially the same security policy mechanisms, but the degree to which the security typing can correctly encode the timed release mechanism varies between them.

The structure uses the following classes:

- `Author`: a simple immutable paper author with a name
  - Polymorphic with respect to security policy
- `Organiser`: an immutable conference organiser with a name
  - Polymorphic with respect to security policy
- `Paper`: an immutable paper with a title, abstract and list of authors
  - Title and abstract may have a separate policy to the list of authors, though both are polymorphic
- `ConferenceSystem`: a conference representation with an organiser, a set of submitted papers, and a set of accepted / allocated papers
  - Contains a list of submitted papers, as well as a map of accepted players to their allocated sessions, and provides the methods which perform the allocation in secret and perform the timed release
  - Defines the security policy placed upon the acceptance status / session allocation, as well as that placed upon the author lists of submitted papers
- `Main`: a main class which creates and runs a conference system

4.2.4 JIF Implementation

A condensed code printout, containing critical sections of the JIF implementation for the Conference Management System, is listed in Appendix A.2.1.

'Semi-parametric' Paper Policy

The `Paper` class requires that the title and abstract fields have low confidentiality, but that the list of authors be restricted. Since `Paper` is essentially a data type class, it is desirable to make it as generic as possible. In order to avoid hard coding either policy, it is possible to simply use two label type parameters. However, this reduces readability, as it is not immediately clear what the implications of instantiating a `Paper[{O->*}, {O->_}]` are.

For this reason, the implementation instead uses a single label type parameter to represent the restriction on the author list, and makes use of the special `{this}` label so that a given `Paper`'s title and abstract have their policy determined by that of the instance itself.

```
1 public class Paper[label AuthorLabel] implements JifObject[AuthorLabel] {
2     private final String{this} title;
3     private final String{this} paperAbstract;
4     private final Author{AuthorLabel}[]{AuthorLabel} authors;
5 }
```

Data Structures in JIF

The Conference Management system requires the use of a map data structure in order to store the mapping from an accepted paper to its session allocation. This exposes one of the practical concerns that working with JIF presents: JIF is built upon Java 1.4 and so lacks Java's generics.

The process of actually retrieving a value from the map is complicated further by JIF's own type system. Maps have label generics for the keys and value types, and as a result both the key and value objects must be of type `JifObject[L]` where `L` is the label for the key or the value respectively. Hence, it is not trivial to use any type as the key or value for a map – it must either conform to one of JIF's provided types or be wrapped in one.

In the case of the allocations map, where the desired value is simply an integer classified to be visible only to the conference organiser `O`, this requires wrap-

ping the value in a `JifInteger[{O->*}]` (so that it is a subtype of the required `JifObject[{O->*}]`). Retrieving the object is then a complex process, made more complex when combined with the declassification of the result, as is necessary in this case.

This process, of retrieving the session number for an accepted paper and declassifying it, is most easily described by annotating the code for it.

Retrieving and Declassifying the Session Number

Note references are annotated in the code in the style of <1> to remain distinct from the code itself.

```
1  int{O->_} <1>
2  declassifySessionNumber{O->_}<2>(String{O->_} title):{O->_}
3          where authority(O) <3> {
4      JifString[{O->_}] titleObj = new JifString[{O->_}](title); <4>
5      JifObject[{O->*}]{O->*} sNoObj;
6      try {
7          sNoObj = allocations.get(new label {O->*}, titleObj); <5>
8      } catch (NullPointerException e) { <6>
9          return -2;
10     }
11
12     // Declassify the object retrieved from the map
13     JifObject[{O->*}]{O->_} sNoObjDeclass <7>
14                                 = declassify(sNoObj, {O->*} to {O->_});
15
16     // Convert it to a JifInteger
17     JifInteger[{O->*}]{O->_} sNo;
18     try {
19         sNo = (JifInteger[{O->*}])sNoObjDeclass; <8>
20     } catch (ClassCastException ex) {
21         return -3;
22     }
23
24     // Pull the int from the JifInteger
25     int{O->*} r;
26     try {
27         r = sNo.intValue(); <9>
28     } catch (NullPointerException ex) {
```

```
29          return -4;
30      }
31
32      // Declassify the int
33      int{O->_} result = declassify(r, {O->*} to {O->_}); <10>
34      return result;
35  }
```

1. The return label is `{O->_}` – the information belongs to the organiser but may be read by anyone

2. The begin and end labels of the method must also be `{O->_}` since after timed release this declassification process must be available to any principal.

3. The authority clause is necessary since only principal `O` (the organiser) may actually perform this declassification.

4. Rather than using the `Paper` as the key, its title is wrapped in a `JifString` and used as the key. To avoid this would require making `Paper` a subtype of `JifObject` and cause issues with security type polymorphism.

5. The actual `get` call itself requires a runtime label parameter, and the resulting object is at the combination of this label and the key label parameter of the map. In this case the label, `{O->*}`, is the same as that of the internally stored value but it is not necessarily the case.

6. The extensive use of try-catch constructs is required because *all* exceptions are checked in JIF, including `NullPointerException` and `ClassCastException`. In this method negative sentinel values are returned to represent failure.

7. This declassification merely allows the code to work with the `JifObject` – the value stored within is still at the level indicated by the type parameter, `{O->*}`.

8. Since JIF lacks generics, the result of the map must be explicitly cast to `JifInteger`. This cast would fail and give a `ClassCastException` if the object stored actually had a different policy than `{O->*}`.

9. At this point, the actual integer value may be retrieved at the classification of the `JifInteger`'s type parameter.

10. The result which has finally been retrieved can then be declassified appropriately and returned.

Timed Release Policy

The JIF implementation for the Conference Management system is made more complex by the nature of JIF's Decentralised Label Model. The core security principle of timed release cannot be modelled by the DLM, as it is inherently stateful. That is, timed release is tied to a boolean variable which may change at runtime.

A straightforward lattice model cannot represent a runtime-variant policy, and while the DLM allows for more complex policy formulation than a traditional lattice model, the reader set for a variable must be known statically.

JIF's security type system is unable to model the timed release policy and so the functionality must be approximated through the use of declassification.

Modeling Timed Release with Declassification

The code for declassifying the session number of an accepted paper (and hence also its acceptance status) was described in detail above. However, nothing in the declassification procedure *required* that it only be performed once timed release had occurred.

This presents a significant issue with using JIF to implement a timed release policy. Since the security type system cannot encode timed release, implementing it necessarily requires giving up the safety that the security type system provides.

This is best illustrated with the relevant code snippet:

```
1 if (allocationsVisible) {
2     return declassifySessionNumber(title);
3 } else {
4     return -1;
5 }
```

This method prevents the leak of the session number of a paper, and hence its acceptance status, by guarding with a boolean condition that is true only after timed release. If the method is called before timed release it will return a sentinel value that gives no information about acceptance status.

With the JIF implementation, the following snippet – only a single character change from the previous – would also compile:

```
1 if (!allocationsVisible) {
```

```
2     return declassifySessionNumber(title);
3 } else {
4     return -1;
5 }
```

This snippet instead releases the session number (and hence leaks the acceptance status) *before* timed release. This violates the security properties of the case study, but JIF's type system does not provide any means by which to control this, and hence provides no more guarantee of confidentiality than plain Java would in this scenario.

4.2.5 Paragon Implementation

A condensed code printout, containing critical sections of the Paragon implementation for the Conference Management System, is listed in Appendix A.2.2.

'Semi-parametric' Paper Policy

As with the JIF implementation, consideration must be given to the need for differing policies on the fields of the `Paper` class. To allow for polymorphism, the Paragon implementation uses a similar strategy as for JIF: use of the `policyof(this)` policy for the title and abstract, and a policy type parameter for the restriction upon authors.

Unlike JIF, Paragon uses the standard Java syntax for generics, so the class is defined as `public class Paper<policy authorRestriction>`. Beyond this, the Paragon implementation benefits from being able to use the standard Java signatures for `equals`, `hashCode` and `toString`, but is broadly similar to the JIF implementation.

Data Structures

Where the JIF implementation made interacting with data structures (like the map used to store sessions allocation) quite verbose, in Paragon the process is mostly similar to plain Java. Where a `HashMap<K, V>` in Java takes as type parameters the key and value type, in Paragon, a `HashMap<K, policy KeyPol, V, policy ValPol>` takes a further two – the policies for keys and values. Policy and actor type parameters use the same notation as regular Java generics and can be mixed with them as required.

In addition, manipulating or retrieving values from a map is simple in Paragon. Compare the code sample in 4.2.4 for retrieving and declassifying a session allocation with the below Paragon equivalent:

```
return allocations.get(paper);
```

Note that there is no explicit declassification here; this is handled by the lock state context in which the above is called, and the design of this is discussed below.

Timed Release Policy

Where timed release cannot be explicitly modeled in the Decentralised Label Model used by JIF's policy language, the logic-based policy language of Paragon combined with the Paralock construct allows for timed release to be simply and clearly stated.

The policies used in the Conference System are the following:

1. The `bottom` policy, allowing any actors to access the context of data labelled with it:

```
static final policy bottom = {Object x :};
```

2. The `ifAllocationsVisible` policy, allowing conference organisers to access the data at any time, and other actors to access it *provided* the timed release has occurred:

```
static ?bottom lock allocationsVisible;

static final policy ifAllocationsVisible = {
        Object x : allocationsVisible
    ; Organiser o :
};
```

There are two different usages of this timed release policy in the application: the paper acceptance status, and the hiding of author details. Unlike JIF, Paragon essentially provides multiple methods for the enforcement of a policy, and so each usage of the policy is discussed separately below, in relation to a specific method of handling enforcement.

'Default Value' Handling

In JIF, there is precisely one mechanism by which information can be downgraded in terms of confidentiality: the `declassify` statement. As discussed in the JIF Implementation section, this declassification is an 'escape hatch' from the type system, and as a result a procedure which declassifies information may be called from anywhere which has the appropriate authority.

Hence, to ensure that the session number of papers did not leak (thereby also revealing *whether* a paper was accepted), the JIF implementation guarded the declassification by a boolean value, and returned a sentinel value of `-1` if the timed release had not occurred:

```
1 if (allocationsVisible) {
2     return declassifySessionNumber(title);
3 } else {
4     return -1;
5 }
```

The Paragon form is superficially quite similar:

```
1 if (allocationsVisible) {
2     return (int)allocations.get(paper);
3 } else {
4     return -1;
5 }
```

Though the runtime behaviour of both is identical, the key difference here is that in JIF, `allocationsVisible` is simply a boolean value, whereas in `Paragon` it is a `lock`. Because it is a Paralock, within the following branch the compiler statically knows the lock is open. Hence, inside that branch the `ifAllocationsVisible` policy reduces to:

```
1 static final policy ifAllocationsVisible = {
2       Object x :
3     ; Organiser o :
4 };
```

This allows any object to access the information, and so no further declassification action is required: the value retrieved from the map may simply be returned.

In addition, since the timed release is properly encoded into Paragon's policy langauge via the lock, the issue encountered in JIF regarding flipping the condition cannot occur. The following code, with the conditional flipped, will not compile in Paragon:

```
1 if (!allocationsVisible) {
2     return (int)allocations.get(paper);
3 } else {
4     return -1;
5 }
```

The end result is a method which is functionally the same as its JIF counterpart – the method may be called anywhere, but will only return a meaningful value after timed release – but the Paragon version is type safe, where the JIF implementation is not.

Compile-time Handling

In both JIF and Paragon, methods may be annotated such that they may only called within a certain policy context (for example – a method which may only be called at high confidentiality). JIF uses the program counter label and method begin / end labels to achieve this; Paragon instead uses write effect annotations.

However, since JIF's policy language cannot express timed release, it is not possible to write a method with a begin label requiring that the method *only* be called after timed release. For the Paragon implementation this is possible through lock state annotations. This method is demonstrated in the case study for the author visibility restriction.

Implementing this requires very little programmer burden: the paper authors are already at the correct restriction level, since each `Paper` is parametrised by the `ifAllocationsVisible` policy. The only requirement is that any method which exposes these details to a lower level of confidentiality should be annotated with `~allocationsVisible` (specifying that the method may only be called when the `allocationsVisible` lock is open).

In the case study's `Main` class, the `printOutAuthors` method, which prints out the authors of all papers in an array to standard output, has this annotation:

```
1 private static ~allocationsVisible !bottom void printOutAuthors(
2     ?bottom List<Paper<ifAllocationsVisible>, bottom> retrievedPapers) {
```

Then, in the `main` function, the effect is demonstrated as follows:

```
// Cannot attempt to show authors of submissions
// printOutAuthors(retrievedPapers);

// Mark allocations as visible (i.e. notification date expired)
conferenceSystem.setAllocationsVisible();

// Can now print out authors of submissions
printOutAuthors(retrievedPapers);
```

If the first `printOutAuthors` line is uncommented, the program will not compile since the lock will not be known to be open at that point of execution. However, the same code executed after a method call known to open that lock (via the `+allocationsVisible` annotation) will be accepted by the type checker.

4.2.6 Conclusion

The timed release policy which is central to the Conference Management case study is quite easily and naturally represented under Paragon's policy model: the Paralock construct neatly maps onto this policy and the annotation tools it provides allow the compiler to explicitly reason about this policy to ensure the security type system is actually providing type safety. In addition, the programmer may choose to implement the policies in multiple ways, whether using default sentinel values to allow methods to be called but guard sensitive results, or using lock annotations to ensure at compile time that sensitive methods are only called in the proper context.

The JIF policy model, however, cannot represent timed release, and so the implementation has to 'escape' from the type system using declassification in order to represent it. As a result, the JIF type system provides little security benefit over a plain Java implementation: the declassification may be used essentially anywhere, and hence information can leak due to simple programmer error.

The case study also demonstrated some of the practical challenges involved in working with JIF in particular: using data structures, such as a simple HashMap, requires significantly more work in JIF than in plain Java, or indeed in Paragon. JIF's lack of support for Java generics increase the burden further.

4.3 Case Study 3: Calendar Scheduler

4.3.1 Overview

The final case study is an implementation of a multi-user calendar scheduler application. The application has multiple users, each with their own calendar, which consists of a list of meetings. Each meeting has a title, and is associated with a day of the week and starting and finishing time. In addition, each meeting has a list of users who are attending the meeting.

The application allows users to schedule new meetings so as to avoid conflicting with existing ones. Each meeting's details are secret – only the attendees may know the title of the meeting. However, the times of the meetings need to be available to all users in order to schedule new meetings.

4.3.2 Key Security Properties

Initially, the security property here may seem to lack complexity. As with the second case study there is a data structure in the form of a meeting which contains some high confidentiality data (the title) and some low confidentiality data (the time). Here however, the data is not time-variant but instead dependent on the involvement of particular users – which seems like a natural fit for the principals of JIF.

There are two key differences that distinguish this policy from those discussed previously:

1. Rather than being 'owned' by a single actor or principal, a meeting is confidential to a *set* of principals
2. It is required that meetings be able to be created at runtime with any possible attendees, so the policy model must be able to reason about the restrictions on such dynamically created meetings

4.3.3 Implementation Structure

Both calendar scheduler implementations follow the following class structure:

- `User`: a simple immutable user with a name
  - Polymorphic with respect to security policy
- `Meeting`: a single immutable meeting with a time, a title and a list of attendees
  - The time information about the meeting is low confidentiality
  - The list of attendees is treated as low confidentiality
  - The title is secret and is accessible only to attendees of the meeting
- `Calendar`: a calendar of scheduled meetings belonging to a user
  - The user must attend all meetings on their calendar
  - Since attendance of meetings on a user's calendar may vary, other users may be able to access different portions of a given user's calendar
- `Scheduler`: keeps a calendar for each user in the system, and allows for checking whether a given time slot conflicts with existing meetings for some set of users
- `Main`: contains a main method to set up dummy data and run the scheduler

4.3.4 Simplification of Implementation

The implementation described above, with the use of confidentiality states which depend on sets of users, is difficult to implement in either JIF or Paragon. In JIF, it is not possible to quantify over a set of principals, which makes encoding the desired policy effectively impossible. Paragon *does* allow for such quantification, but this does not extend to the lock annotations necessary to encode the policy of a `Meeting`.

As a result, the JIF implementation instead uses a simplified security policy, where each meeting is designated a single *owner* principal who controls its confidentiality. The Paragon implementation functions with both the original policy and the simpler ownership model policy, but the compiler fails to enforce either. This does not quite produce the desired application in either case, but even so the simplified implementation still runs up against language and compiler limitations in both Paragon and JIF.

4.3.5 JIF Implementation

A condensed code printout, containing critical sections of the JIF implementation for the Calendar Scheduler, is listed in Appendix A.3.1.

'Sets' of Users

The desired confidentiality property, where a meeting's title is visible to its attendees, is difficult to encode in a purely statically checked manner, regardless of the policy model used, since the set of attendees of a meeting is a runtime value. In JIF's case, there is no model for expressing the concept of an arbitrary set of principals through the type system: a class may be parametrised by *one* principal, but not a set of them.

Since this quantification over sets of principals cannot be encoded within JIF, the implementation proceeded as per the simplified policy in 4.3.4.

'Ownership' Policy Structure

Restricting the confidentiality policy to a single meeting owner rather than a list of attendees made implementation more feasible. Under this model, the `Meeting` class must be associated with an 'owner' principal. This may be achieved in JIF either through a principal type parameter, and so the low confidentiality data is labelled `{Owner->_}`, and the high confidentiality title is labelled `{Owner->*}`.

Dynamic Users

The initial design of the case study implementation has the `User` class, each instance of which is associated with a user of the application. Unlike Paragon, JIF does not use object instances as actors, and so there is no connection in the type system between a `User` instance and the principal it would represent.

This presents a challenge for development: for the application to make sense, it should not be hard-coded to some predefined list of principals. In JIF, this is not possible at the top level: principals are types (i.e. classes) and it is not possible to dynamically create them without reflection (which JIF does not support).

This problem was encountered by researchers at Pennsylvania State University in developing the JPMail secure email client [36]. An email client should support multiple, arbitrary users and as a result a set of policy tools were built for JPMail,

alongside the 'JIFclipse' suite of tools [37]. However, this system requires extensions to the JIF language and runtime, and even with these, in practice JPMail cannot handle new users added 'on the fly' – it requires recompilation of certain components.

4.3.6 Paragon Implementation

A condensed code printout, containing critical sections of the Paragon implementation for the Calendar Scheduler, is listed in Appendix A.3.2.

Dealing with Sets of Users

The quantification issue also affects Paragon. Paragon's policy language uses objects as actors and allows quantification over them, and so it is able to get closer to modelling the concept. A binary lock `isAttending(User, Meeting)` may be defined, where the lock is open for User `u` and Meeting `m` (i.e. `isAttending(u, m)`) if `u` is in the list of attendees of `m`.

Since the list of attendees itself is a runtime value, this status cannot automatically be 'linked' to the relation via a lock property: the lock must be opened manually for each attendee when the Meeting object is created, via the code snippet below from the constructor of `Meeting`:

```
1 this.users = new LinkedList<User, {Object x:}>();
2 for (int i = 0; i < attendees.size(); ++i) {
3     // attendees is the list of attendees passed to the constructor
4     User u = attendees.get(i);
5     open isAttending(u, self);
6     this.users.add(u);
7 }
```

While the policy language allows this to be expressed, the Paragon compiler is not able to perform the necessary static analysis to enforce this, and hence when secret data is accessed (e.g. by printing out the Meeting via its `toString` method), the compiler cannot tell whether the lock is in fact open. Paragon estimates the lock state conservatively, and therefore since it cannot be *sure* the lock is open, it treats it as closed, causing compile errors.

Paragon has two methods for getting around this, corresponding to the two mechanisms demonstrated in the Conference Management example: either method annotations can be used to restrict where the method may be called, or a run-time

check can be used to provide a code context in which the lock is known to be open.

The former option cannot be applied to this case, because while it is possible to require a lock to be open (through something of the form `~someLock`), for a binary lock one cannot quantify over the lock's parameters. That is, to express this policy an annotation would have to encode the concept "`isAttending(u, m)` must be open for the current instance `m`, *for all* users `u` such that `u` ∈ `attendees`."

The latter method can be expressed by a runtime check in a for loop, and will produce compiling code; this is presented in the following snippet from the `Calendar` class:

```
1  public ?{viewer: } String getCalendarRepresentation(notnull User viewer) {
2      ?{viewer: } String s = user.toString() + "'s Calendar:\n---";
3      for (int i = 0; i < meetings.size(); ++i) {
4          Meeting m = meetings.get(i);
5          if (Meeting.isAttending(viewer, m)) {
6              s += m.toString();
7          }
8      }
9      return s;
10 }
```

However, this approach was also unsuccessful in implementation because while the policy language allows for it, the Paragon compiler failed to then use this information correctly: it would actually *violate* the required policy and allow a non-attendee to print out the secret information.

Ownership model issues

The simplified 'owner policy' was also defined for the Paragon implementation. As with JIF, a type parameter-based implementation is possible. Unlike JIF, Paragon's policy language has explicit support for dynamic actors (since actors are simply object instances), and its extension of Java's generics allow for more flexibility, as well as generic methods, which JIF lacks entirely.

The Paragon implementation of this policy avoids parametrising meetings by their owner, instead opting to keep a reference to the owner instance via a final field `user`. This concept requires the 'objects as actors' model and hence has no clear mapping to JIF.

Functionally, this means that the binary lock `isOwner(User, Meeting)` guards

access to the confidential meeting title information. Since this will only be true for one user, the quantification issue does not arise. This policy's version of the `getCalendarRepresentation` method is listed below:

```
1  public ?{user:} String getCalendarRepresentation() {
2      ?{user:} String s = user.toString() + "'s Calendar:\n---";
3      for (int i = 0; i < meetings.size(); ++i) {
4          ?{Object x:} Meeting m = meetings.get(i);
5          if (Meeting.isOwner(user, m.self)) {
6              s += m.toString();
7          }
8      }
9      return s;
10 }
```

In practice, Paragon's compiler was still unable to correctly analyse this: runtime checks are still required since the compiler is unable to statically determine the correct user, and once these checks are added the compiler incorrectly allows users other than the owner to print out information. This appears to be a practical issue with the immaturity of the Paragon compiler: interpreting the verbose-mode output of the compiler indicates that the actor identifier being associated with the owners of each meeting is the same, where in fact at runtime these owners will be different instances.

4.3.7 Conclusion

This case study could not be implemented correctly and with the desired information flow safeguards in either Paragon or JIF.

JIF runs into a dead end in attempting to deal with dynamic principals. This seems to be a fatal flaw in any attempt to use JIF to model this problem. More complex policy model and policy store systems like those implemented in Jifclipse [37] may provide a solution to some of these issues, but these are not under active development, and work only with older versions of JIF.

Paragon's policy model is able to represent the policy, but still struggles with statically analysing a policy requiring quantification over a list of attendees. Ultimately, Paragon cannot enforce even the simplified owner model policy. A type parameter-based implementation would still face some of these issues, but may be worth investigating in a future project as a way around the Paragon compiler issues encountered.

4.4 Practicalities

Developing the three case studies presented above led to some observations regarding the practical usage of the JIF and Paragon languages. Both were developed primarily for research, and so both lack the kind of thorough documentation and tooling that most languages used at a commercial scale have. In addition to this, the theory of information flow is not widely taught and a solid understanding is required in order to effectively write programs in either language.

4.4.1 Conceptual Burden

The largest initial barrier to writing programs using JIF or Paragon is the conceptual burden inherent in any language with places controls on information flow. Access control is well understood and is commonly used both in application development and in the day-to-day operation of a computer, and Discretionary Access Control in particular is both intuitive and so common that virtually every person involved in the process of developing, deploying and using an application is familiar with it.

The same cannot be said for security typed information flow. Implementing a program using a security typed language requires first understanding the theory behind the Bell-LaPadula lattice model, non-interference and declassification, which are not intuitive to a programmer lacking a background in mathematics or theoretical computer science. In addition, policy models differ between languages: writing JIF code requires an understanding of the DLM, and writing Paragon code requires conceptualising policy in terms of the ParaLock model.

JIF

Paragon has a significant advantage in terms of conceptual burden. JIF requires formulating policy in terms of interacting, potentially mutually distrustful principals. It requires first understanding the hierarchy of principals, and then the model by which ownership-based policies are transformed into the final ‘reader set’. This model makes sense for some problems, as in the Battleships case study where the two players may be aligned with principals, each with data which should remain confidential to them except through a clearly defined declassification channel.

However, defining a policy in JIF becomes difficult when the desired policy is inherently dynamic or not built upon interacting principals, as in the Conference

Management case study, where the *timing* of declassification was fundamental to the policy.

Paragon

Paragon by contrast builds its policy model on predicate logic. Any programmer familiar with access control is able to understand the intention of labels like `{alice: }`, and labels like `{User u: }` require only an understanding of universal quantification, aided by the use of 'objects as actors', which allows for standard Java concepts such as inheritance to be easily mapped into the policy domain. Though the use of join and meet to combine policies does require some understanding of lattice structures, these concepts also map neatly to the ideas of 'intersection' and 'union' of policies respectively. Hence, constructing a static policy in Paragon is generally straightforward.

Dynamic policies, too, can easily be mapped between the desired security property and its Paragon representation. Unparametrised Flow Locks are simply boolean values, and parametrised ParaLocks can be conceptualised as unary and binary relations. Many practical confidentiality policies can be expressed in natural language in the form "User x may read information y if condition p holds", which can be translated almost verbatim into Paragon's policy language.

4.4.2 Annotation Burden & Boilerplate

Paragon and JIF both require the programmer to annotate policies throughout the code. This results in burden on the programmer as they have to keep the policy in mind at all times.

Policy Agnosticism, Type Inference and Null Pointer Analysis

Ideally, information flow mechanisms would be *policy agnostic* [38][39]; that is, flow policy would be defined separately to code. This model is common in access control – for instance, the Java security model defines access policies in an external file, which are then enforced at runtime by the Security Manager [18]. Policy agnosticism allows for code reuse and eases the conceptual burden on the programmer.

While neither Paragon nor JIF are policy agnostic languages, they do both provide some mechanisms to reduce annotation burden both via security type polymorphism and via type inference: it is generally possible to leave local variables without

annotations, and the compiler will attempt to determine the correct label for that variable. In theory this is always possible and a failure of type inference usually means that the code is somehow incorrect, but at times it is difficult to debug a label issue without explicitly adding in annotations to local variables.

In addition to type inference, Paragon and JIF both provide null pointer analysis. Since all exceptions – including `NullPointerException` – are checked in JIF and Paragon, it is therefore often necessary to use `try-catch` constructs, which significantly increases the code's line count and decreases readability. Both languages use static analysis techniques to try and determine which values are known not to be null, and the compiler then allows these variables to be used without catching the `NullPointerException`.

In practice, JIF is better able to perform this than Paragon; Paragon's analysis often fails to determine that a value cannot be null, even when its state was explicitly checked, such as in the following snippet:

```
1 if (myObject == null) return;
2 myObject.method();
```

If the above code were included in a JIF program, the null pointer analysis would determine that `myObject` cannot be null on the second line, and therefore a `try-catch` is not required. Paragon's null pointer analysis would sometimes fail here, requiring a `try-catch` construct around the method call, or the use of Paragon's `notnull` annotation to make it clear.

Boilerplate and Interfacing with Java

One concern that potentially limits the applicability of these languages to developing real applications is the additional boilerplate code required when writing programs in JIF or Paragon which need to interface with external resources, whether with the Java standard library or other external Java libraries.

Both languages have a system which allows a JIF or Paragon interface to be written for a regular Java class, without that class being reimplemented. The standard distribution of each comes with a set of interfaces for commonly used subsets of the Java standard library.

In practice, interacting with the Java standard library is substantially simpler in Paragon than in JIF. In Paragon, the APIs for most Java standard library components may be used without modification, with some extensions such as the `PrintStream` class, which in Paragon is parametrised by a policy in order to allow for output streams of varying confidentiality.

JIF also provides a label-polymorphic `PrintStream` class, but JIF also requires specialised implementations of even the most core Java concepts: the programmer cannot simply override the `equals`, `hashCode` and `toString` methods provided by the Java `Object` class; they must instead override those provided by the `JIFObject[label L]` interface in order to allow comparisons with objects with differing labels.

Interacting with the user in Paragon is generally similar to doing so in plain Java, but as discussed in 4.1.4, merely accessing standard out in JIF involves significant boilerplate code.

4.4.3 Documentation & Tooling

Tooling

Neither JIF nor Paragon provide tooling to automate the process of development. There exist two projects, JIF IDE [4] and JIFClipse [37] designed to integrate JIF into the Eclipse IDE. During the development of these case studies, it was not possible to set either of these up: JIFClipse works only for older versions of JIF, and in practice the JIF IDE plugin crashed every time a JIF source file was opened or displayed in the Eclipse editor.

In order to make development of the three case studies smoother, scripts were written using a combination of bash and Apache Ant to allow JIF and Paragon projects to be compiled, run and tested effectively.

Documentation

JIF and Paragon both have limited language documentation available. JIF's webpage [4] provides a user's manual [40], as well as generated JavaDocs for the internals of the runtime. The JIF manual contains material introducing the Decentralised Label Model, outlining the features of the JIF language and providing information about interacting with Java code.

However, the JIF manual has not been updated since JIF version 3.3.0, released in 2009, and so is missing some content which applies to the current JIF version 3.5.0. The difference between these versions has some consequences, particularly with regard to a new "robust declassification" mechanism which means that some existing JIF code – including JIF code directly included in examples in the manual – will no longer compile with the current version.

Paragon's webpage [5] does not provide a user's manual, but it does include a tutorial for the language [41], available both in PDF form and as an interactive webpage including editable code which may be run through the Paragon compiler straight from the webpage.

4.4.4 Compiler Maturity

While neither JIF nor Paragon have compilers which are as well documented or as well tested as those of commercial languages, JIF has been under active development significantly longer than Paragon, and as a result the JIF compiler has fewer noticeable bugs and generally produces more detailed error messages when compilation fails.

Installation & Compiler Bugs

As research languages, JIF and Paragon do not provide automatic installers or other conveniences that most commercially used languages do. In particular, installing JIF requires building the runtime (and potentially the compiler as well) yourself. On Windows this installation depends on tools from the Cygwin project [42] to compile.

Paragon also has installation-related issues; though binaries are provided for both the compiler and runtime, the Windows version of the Paragon v0.2 compiler is non-functional: it compiles code using a mix of conventions from Paragon v0.1 and v0.2 in such a way that the resultant `.java` file can never compile with `javac`. The compiler's Linux binary does not have this issue.

Regardless of the version of the Paragon compiler, it does still occasionally produce incorrect results. The most clear case of this that was encountered while developing the case studies is that it fails to correctly compile runtime checks of lock state. Indeed, in order to compile the case studies first from Paragon code to Java code, and then to Java `.class` files, it is necessary to manually modify the output Java source files in order to correct the bug.

Compiler Error Messages

When JIF presents an information flow error during compilation, it displays a short description of the error, alongside the file and line number; where possible, it also includes the specific variable or statement which caused the error.

The JIF compiler also includes a `-explain` flag, which causes error messages to include details of the flow constraint which was violated. There are subtle differences between platforms with these errors: in particular, the unicode symbols such as those indicating the join and meet operators do not print correctly when run under Cygwin, being replaced with question mark characters before even being output to the terminal. This makes interpreting compiler errors on Windows very difficult.

Paragon similarly shows a short message along with the file, method and line number, though Paragon's static checker is often unable to correctly determine the line number on which the error occurs. Paragon includes a verbose compiler flag which provides more information, and also includes verbosity 'levels' which can log more of the compiler's actions.

Paragon's error messages, however, are often less explicit than those produced by the JIF compiler. A common compiler error message in Paragon states that "the system failed to infer the set of unspecified policies". This message generally occurs as a result of a failure of local variable type inference, which is usually caused by the programmer attempting to perform an action which violates a policy; however, there is no detail to the message and it provides little information as to *where* the violation occurred.

In addition, the Paragon compiler sometimes simply crashes during compilation; finding the element of the input file which caused the compiler to fail is often difficult – for instance, objects used as actors should be marked `final`, but if the modifier is missing under certain circumstances the compiler will simply crash, giving no indication that this is the cause.

Chapter 5

Conclusions & Further Work

5.1 Information Flow

Language-based information flow security has potential application in any program which enforces a confidentiality policy, and where programmer error has the potential to lead to accidental violation of that policy.

As this has been a topic of research for several decades, a number of security typing-based models have been developed. Each of these builds on those that came before and attempts to improve policy definition and enforcement. The model implemented by Denning and Denning [22] allows for verification of programs under a standard MAC lattice. The Decentralized Label Model improves upon this by allowing policies with no central source of truth, and by framing policies in terms of interacting principals. The ParaLock model [33] attempts to address the deficiencies of prior systems in defining dynamic policies in more sophisticated ways than straightforward declassification.

Using the tools these models provide it is possible to construct and enforce many useful security policies, from simpler systems which enforce a MAC lattice policy to those which require a dynamic 'timed release' policy, and through the application of these techniques it is possible to mitigate the real-world risks of improperly enforced confidentiality policies that programs that interact with data in any meaningful way face.

5.2 JIF & Paragon

The JIF and Paragon programming languages pair relatively mature implementations of security typed information flow controls with the syntax and standard library of Java, a widely used language. Both can provide powerful confidentiality guarantees through information flow checking which is primarily performed at compile time, avoiding the potential overhead of a runtime mechanism.

Both systems can be used to encode meaningful security policies. JIF's model allows for policies built around interacting principals and does not require a central source of truth to determine what policy should be applied to information. Its declassification mechanism allows for a much larger class of programs, beyond those which are strictly non-interfering, to be written, with some level of control applied through the 'authority' construct.

Paragon's policy model is more general, and has a lower conceptual barrier to entry as a result of building itself upon the familiar constructs of predicate logic. Its use of Flow Locks and ParaLocks allows for a wider range of dynamic flow policies than can be encoded using JIF's Decentralised Label Model.

5.2.1 Case Studies

The case studies presented in Chapter 4 demonstrate both the application of the two languages and their limitations.

Battleships

The Battleships case study presented in 4.1 requires modelling a 'game board' containing information secret to one of the game's players. This information must not leak to the other player, except through a specific querying mechanism as part of the game's rules. This policy is easily encoded in JIF by identifying each player with a principal and using a declassifying method to handle queries. Paragon similarly encodes this; it does not require an explicit principal for each player, because Paragon uses 'objects as actors' – the player instance itself *is* the principal. Paragon performs declassification using a simple flow lock, cleanly emulating the same declassification strategy as the JIF implementation.

Conference Management

The Conference Management case study presented in 4.2 models a 'timed release' policy, where the confidentiality status of session allocations for a paper submitted to a conference changes during the system's operation. Until the conference gives notification, the status is secret, and authors cannot learn whether their paper was accepted; after the notification the status is public information.

JIF's Decentralised Label Model cannot naturally encode this policy; timed release cannot be expressed in terms of interacting principals. As a result, the JIF implementation of this case study relies on using declassification as an 'escape hatch' from the security typing system. Developing this application in JIF is feasible, but doing so requires giving up some of the confidentiality guarantees that security typing provides.

Paragon, by contrast, encodes this policy with a single lock, which is opened when the timed release condition is met. Hence, the Paragon implementation of this application is able to provide more stringent confidentiality guarantees than the JIF version.

Calendar Scheduler

The Calendar Scheduler case study presented in 4.3 models a policy where the confidentiality of a meeting's details depends on a runtime list of attendees. Here, neither JIF nor Paragon could usefully encode the desired confidentiality policy.

JIF's policy model fundamentally cannot express the concept of confidentiality dependent upon a dynamic *set* of users. A class may be parametrised by a fixed number of principals, and runtime principals and labels may be used under some circumstances, but the type system is unable to actually encode or enforce the policy required to provide meaningful verification of the confidentiality properties of this application.

Paragon comes closer to encoding the policy; its policy definition language allows for universal quantification over objects, and a parametrised lock may be used to denote attendance status. However, the compiler's type checker is then not able to properly enforce this policy and as a result, Paragon cannot provide the desired security guarantees.

Practicalities

Setting up either JIF or Paragon and developing the tooling required for streamlined development is a non-trivial task; documentation is sparse and when debugging programs both compilers often output compiler errors which are difficult to interpret. The JIF compiler is (unsurprisingly, given it has been in development for far longer) substantially more mature than the Paragon compiler, which has a number of unresolved bugs.

While these rough edges are to be expected for research languages, they do make it difficult to recommend using either JIF or Paragon to write a serious application, and they emphasise the need for polish in any information flow language which seeks to become practical outside of academic research.

In addition, the conceptual burden of both languages is still high. Paragon's policy model is simpler to understand than JIF's, making use of theory intuitive to any programmer who has studied basic logic, but even so a background knowledge of lattice-based policy, information flow and non-interference is required to begin writing even trivial applications.

5.3 Conclusions

The more streamlined policy model of Paragon combined with its greater flexibility make it both easier to use and better able to represent the confidentiality policies of real-world applications. JIF's policy model is effective for modelling certain kinds of policies, and the implementation of the JIF distribution is substantially more mature than that of Paragon, but its comparative lack of expressiveness and its high programmer burden present a significant challenge to its use in practical development.

The case studies implemented covered a range of policies, including policies which could not be adequately represented in either language. These issues, along with a general lack of maturity in the ecosystem of information flow tools, substantially limit the applications for which security typed information flow is a feasible choice. Regardless, the concepts have potential for large scale application in real world development and the models being implemented in research have (and will likely continue to) become increasingly powerful and expressive over time.

5.4 Further Work

The analysis of the JIF programming language presented in this thesis focuses on development using recent versions of JIF and the standard JIF runtime. The JPmail project [36] at Pennsylvania State University developed a set of policy tools for JIF which were not thoroughly examined for this comparison, including an Eclipse plugin, JIFclipse, along with tools supporting a policy store with dynamic look-up of principals. These tools were, however, designed for a significantly outdated version of JIF and require the use of a modified JIF runtime, and the project is no longer under active development.

In the literature, there are other approaches to alleviating some of the limitations of JIF. One such approach is that taken by the RX language [43], which builds upon the Decentralised Label Model but attempts to introduce support for dynamic policy through *policy updates*, using both static analysis and runtime checks to reason about the information flows through a program. Though the principles of the RX language have been developed through two papers [43][44], it has not been implemented in practice and the most recent paper was published in 2006.

Another system, Java Object-sensitive ANAlysis (JOANA) [45], makes use of a separate static analyser and annotations in Java code, rather than implementing information flow checks by extending the type system. It analyses Java bytecode against traditional lattice-based policies by generating a dependency graph of flows through the system. This approach is is built on substantially different theoretical foundations to those of JIF and Paragon, and it has potential upsides in requiring lower programmer burden, not needing extensive syntax changes to standard Java. As a result, it would make a potentially useful comparison with more traditional security typed languages if performed in depth.

In a 2016 paper, Kozyri, Arden, Myers, *et al.* [46] introduce an approach to information flow with dynamic policy which aims to address limitations in JIF, based on 'Reactive Information Flow (RIF) automata'. Whereas a DLM label as used in JIF encodes a set of principals who may read the data, a RIF label encodes a finite state automaton, with each state associated with a set of principals who may read the data *when the automaton is in that state*. Dynamic policies are represented through the use of transitions in these automata.

The paper [46] introduces JRIF, a language based upon JIF which implements RIF automaton-based security typing, and presents example implementations of Battleships, and a calendar application (which is conceptually similar to the Calendar Scheduler case study presented in 4.3). The JRIF language potentially presents a way to resolve many of the issues encountered with JIF's label model through the

development of the three case studies in this thesis, but the JRIF label syntax is substantially more complex than even JIF, so it is unlikely to be a panacaea for the programmer burden issues encountered with JIF.

A radically different approach to information flow security is that implemented by Polikarpova, Yang, Itzhaky, *et al.* [3] with 'Liquid Information Flow Types' (LIFTy). LIFTy is an extension of the Haskell programming language that combines information flow with program synthesis techniques: rather than *verifying* a program as respecting an information flow policy, the compiler synthesises runtime checks to enforce them. Hence, the language is 'policy agnostic': the user does not need to annotate their code at all, instead merely specifying the policy separately. The compiler will then insert runtime checks that ensure the policy may never be violated.

These newer advances in the information flow literature (and the implementations of them) are for the most part quite recent, and lack the broad supporting literature or the maturity of implementation provided by JIF or Paragon. Nonetheless these systems are designed with the limitations of earlier security typing-based languages in mind, and may well play a part in addressing the issues that make JIF and Paragon impractical for application development in most contexts.

5.5 Project Reflection

5.5.1 Focus of Evaluation

Initially, several approaches to the idea of evaluating information flow security were considered. One approach was to compare the use of information flow-based security with specific, existing mechanisms for access control. This approach would have the advantage of providing a direct comparison with systems (such as Java's security model) which are both well understood and which are actively used within industry. However, there are significant differences in the *intent* of these different models, and hence in the kinds of problems they attempt to solve.

Java's security model focuses primarily on controlling the potential actions of *code*, rather than the confidentiality of *data* per se. These are related, but they have distinct purposes. For instance, information flow mechanisms do not attempt to resolve concerns about the trustworthiness of code which is loaded dynamically at runtime and potentially sourced from the internet, but this is fundamental to Java's security model. And similarly, the actual information revealed by a

program's execution through both variables in memory and implicit flow is not a problem considered by the Java security model.

A second approach was to focus on the ways in which information flow mechanisms could impact on or mitigate the effects of security vulnerabilities within Java itself and commonly used Java libraries. This approach would have involved examining known vulnerabilities listed in the Common Vulnerabilities and Exposures (CVE) system, and relating them to the guarantees provided by an information flow mechanism.

This approach was also problematic, as many vulnerabilities do not directly relate to the flow of information, and those that do primarily fall outside the realm of what can be addressed by existing information flow implementations. For instance, they often rely on reflection, which security typed languages like JIF and Paragon do not implement.

Ultimately, the approach shifted to considering information flow mechanisms on their own terms, with a couple of guiding questions: first, "Are they useful?", and second, "Which mechanisms are most useful?". The first question is not easily answered directly; there are clear applications for information flow which derive from its nature, but its usefulness more broadly is inherently tied to the specific implementation, and hence to the second question, which became the focus of the project.

The key decision which flows from this approach is the question of which mechanisms to consider. Information flow encompasses a number of related fields, but even within the scope of *language-based* information flow security, a number of mechanisms have been proposed and implemented. It was clear that JIF should form a part of any comparison – it is by far the most mature implementation of language-based information flow, and is used as a baseline in many other parts of the literature. The decision was made to stick to 'mostly static' implementations of information flow, as runtime-enforced information flow tends to be focused on web technologies (for instance, the FlowFox web browser [24]) rather than on languages like Java.

Paragon was then a natural language to compare against, as it explicitly describes itself as a 'third generation' security typed language, representing a generational leap over 'second generation' languages like JIF.

5.5.2 Comparison Methodology

The driving reason behind the decision to compare only two security typed languages, JIF and Paragon, was that in order to meaningfully evaluate how these languages can be used in practice it is necessary to spend time developing non-trivial programs with them. Attempting to do this thoroughly with more than two languages would have been impractical under the project's time constraints, though this does result in some promising avenues (such as the RJIF [46] and JOANA [45] systems) not being explored.

Early in the project, it was anticipated that the comparison would centre around the development of a single large application, and that from this development conclusions about both the security properties and programmer burden of each language would be drawn. A candidate for this larger application was an email client – similar to that developed by the JPmail project [36]. However, it became apparent quite quickly that developing an application of this scale would require a substantial level of skill in developing with security typed languages, and that neither JIF nor Paragon was mature enough for this implementation to be practical – note that the JPmail project had to build new tools and modify the JIF runtime in order to complete their implementation.

The final approach, to instead build three more modest case studies, was substantially more practical given the constraints of the project. This decision also allowed for a more meaningful comparison between different kinds of security policies that might be desired by application developers.

The Battleships case study was chosen first, as the JIF implementation is bundled as an example with the JIF compiler, making it a natural program to use for comparison. The Conference Management System case study was selected based on its prevalence as an example in the literature [2][3]. Finally, the Calendar Scheduler was selected as it represents a commonly used program design which necessitates a more complex form of dynamic policy.

5.5.3 Reflection: Conclusion

Overall, though the general project direction took a significant amount of time to stabilise, the resulting comparison based on case studies produced clearer and more insightful comparisons than an attempt to build a single larger application would have. However, the limitation in scope caused by restricting the comparison to be only between JIF and Paragon produces some limits on the usefulness of the conclusions drawn, since more recent and less mature attempts to address the issues exposed by the case studies are not thoroughly considered.

In addition, lack of familiarity with the languages and concepts made the development of the case studies relatively slow and error-prone. JIF and Paragon both have a high conceptual burden, and as documented in 4.4 merely setting up these languages for development is a non-trivial task. Skills and familiarity with the languages were developed incrementally over the lifespan of the project, and so the case studies could have been improved, made clearer and designed with more sophisticated confidentiality properties if they had been developed after a more in-depth knowledge of the languages and the theory had been acquired.

Nonetheless, these languages are widely used within the literature of the field of information flow, and crucially, there is relatively little literature about the application of these languages to developing real-world programs. Hence, there is value in comparing and evaluating these languages on those terms, and case studies provide a methodology for performing this comparison in a way which both provides a wide range of confidentiality problems and relates to practical application development.

Bibliography

[1] P. Laverty, *About user enumeration*, `https://blog.rapid7.com/2017/06/15/about-user-enumeration/`, 2017.

[2] S. Agrawal and B. Bonakdarpour, "Runtime verification of k-safety hyperproperties in HyperLTL", in *IEEE 29th Computer Security Foundations Symposium (CSF)*, IEEE, 2016, pp. 239–252.

[3] N. Polikarpova, J. Yang, S. Itzhaky, and A. Solar-Lezama, "Type-driven repair for information flow security", *Computing Research Repository (CoRR)*, 2016. [Online]. Available: `http://arxiv.org/abs/1607.03445`.

[4] A. C. Myers, O. Arden, J. Liu, and T. Magrino, *Java Information Flow homepage*, `http://www.cs.cornell.edu/jif/`, 2016.

[5] N. Broberg, B. van Delft, and D. Sands, *Paragon homepage*, `http://www.cse.chalmers.se/research/group/paragon/`, 2013.

[6] R. L. Krutz and R. D. Vines, "Confidentiality, integrity, availability", in *Cloud Security: a Comprehensive Guide to Secure Cloud Computing*. Wiley Pub., Inc., 2010, p. 63.

[7] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models", *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[8] R. S. Sandhu and P. Samarati, "Access control: Principle and practice", *IEEE communications magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[9] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations", MITRE Corp., Bedford, MA, Tech. Rep. MTR-2547, Vol. 1, 1973.

[10] R. S. Sandhu, "Lattice-based access control models", *IEEE Computer*, vol. 26, no. 11, pp. 9–19, 1993.

[11] A. K. Jones and R. J. Lipton, "The enforcement of security policies for computation", in *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, ser. SOSP '75, Austin, Texas, USA: ACM, 1975, pp. 197–206. DOI: 10.1145/800213.806538.

[12] L. Gong and Sun Microsystems, *Java SE documentation: Platform security architecture*, https://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc1.html, 1999.

[13] Sun Microsystems, *Design goals of the Java programming language*, http://www.oracle.com/technetwork/java/intro-141325.html, 1997.

[14] G. McGraw and E. Felten, *Securing Java*. John Wiley & Sons, Inc., 1999.

[15] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Pearson Education, 2014.

[16] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 2014.

[17] Oracle Corporation, *SecurityManager Class JavaDocs*, https://docs.oracle.com/javase/9/docs/api/java/lang/SecurityManager.html, 2017.

[18] L. Gong and G. Ellison, *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*, 2nd. Pearson Education, 2003, ISBN: 0201787911.

[19] A. Sabelfeld and A. C. Myers, "Language-based information-flow security", *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003, ISSN: 0733-8716.

[20] E. Cohen, "Information transmission in computational systems", *SIGOPS Operating Systems Review*, vol. 11, no. 5, pp. 133–139, Nov. 1977, ISSN: 0163-5980. DOI: 10.1145/1067625.806556.

[21] H. Søndergaard and P. Sestoft, "Referential transparency, definiteness and unfoldability", *Acta Informatica*, vol. 27, no. 6, pp. 505–517, 1990.

[22] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow", *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[23] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis", *ACM Sigplan Notices*, vol. 44, no. 8, pp. 20–31, 2009.

[24] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "Flowfox: A web browser with flexible and precise information flow control", in *Proceedings of the 2012 ACM conference on Computer and communications security*, ACM, 2012, pp. 748–759.

[25] W. Landi, "Undecidability of static analysis", *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, Dec. 1992, ISSN: 1057-4514. DOI: 10.1145/161494.161501.

[26] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification", in *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, IEEE, 2005, pp. 255–269.

[27] A. W. Roscoe and M. H. Goldsmith, "What is intransitive noninterference?", in *Computer Security Foundations Workshop, 1999. Proceedings of the 12th IEEE*, IEEE, 1999, pp. 228–238.

[28] D. E. Denning, P. J. Denning, and G. S. Graham, "Selectively defined subsystems", Purdue University, Tech. Rep. 74-124, 1974.

[29] D. Volpano and G. Smith, "Verifying secrets and relative secrecy", in *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 2000, pp. 268–276, ISBN: 1-58113-125-9. DOI: 10.1145/325694.325729.

[30] A. C. Myers and B. Liskov, "A decentralized model for information flow control", *SIGOPS Operating Systems Review*, vol. 31, no. 5, pp. 129–142, Oct. 1997, ISSN: 0163-5980. DOI: 10.1145/269005.266669.

[31] N. Broberg and D. Sands, "Flow locks: Towards a core calculus for dynamic flow policies", *Lecture Notes in Computer Science*, vol. 3924, p. 180, 2006.

[32] N. Broberg, B. van Delft, and D. Sands, "Paragon for practical programming with information-flow control", in *APLAS*, Springer, vol. 8301, 2013, pp. 217–232.

[33] N. Broberg and D. Sands, "Paralocks: Role-based information flow control and beyond", in *ACM Sigplan Notices*, ACM, vol. 45, 2010, pp. 431–444.

[34] A. C. Myers, "Jflow: Practical mostly-static information flow control", in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1999, pp. 228–241.

[35] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, no. 4, pp. 410–442, 2000.

[36] B. Hicks, D. King, P. McDaniel, and K. Ahmadizadeh, *Jpmail: An email client with information-flow control*, http://siis.cse.psu.edu/jpmail/, 2006.

[37] B. Hicks, D. King, and P. McDaniel, "Jifclipse: Development tools for security-typed languages", in *Proceedings of the 2007 workshop on Programming languages and analysis for security*, ACM, 2007, pp. 1–10.

[38] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies", in *ACM SIGPLAN Notices*, ACM, vol. 47, 2012, pp. 85–96.

[39] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama, "Faceted execution of policy-agnostic programs", in *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*, ACM, 2013, pp. 15–26.

[40] S. Chong, A. C. Myers, K. Vikram, and L. Zheng, *JIF Reference Manual*, `https://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html`, 2009.

[41] B. van Delft, N. Broberg, and D. Sands, *Paragon tutorial*, `http://www.cse.chalmers.se/research/group/paragon/`, 2013.

[42] The Cygwin Project, *Cygwin Homepage*, `https://www.cygwin.com/`, 2017.

[43] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic, *Managing policy updates in security-typed languages*, 2006.

[44] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic, "Dynamic updating of information-flow policies", in *Proceedings of the International Workshop on Foundations of Computer Security (FCS)*, Chicago, IL, 2005.

[45] J. Graf, M. Hecker, and M. Mohr, "Using JOANA for information flow control in java programs - a practical guide", in *Software Engineering (Workshops)*, vol. 215, 2013, pp. 123–138.

[46] E. Kozyri, O. Arden, A. C. Myers, and F. B. Schneider, "JRIF: Reactive information flow control for java", Cornell University Computing and Information Science, Tech. Rep. 1813-41194, 2016.

Appendix A

Case Study Code Samples

A.1 Case Study 1: Battleships

A.1.1 JIF Implementation Sample

Listing A.1: Battleships JIF Implementation

```
1  /*
2   * Immutable (x, y) pair;
3    * overrides JifObject's versions of hashCode(), equals(), toString()
4   */
5  public class Coordinate[label L] implements JifObject[L] {
6      public final int{L} x;
7      public final int{L} y;
8  }
9
10 /*
11  * Immutable ship covering a line of coordinates vertically or horizontally
12  */
13 public class Ship[label L] implements JifObject[L] {
14     final Coordinate[L]{this} pos;
15     final int{L} length;
16     final boolean{L} isHorizontal;
17 }
18
19 /*
20  * Mutable list of ships representing a player's board
21  */
22 public class Board[label L] {
23     private final List[L]{this} ships;
24 }
25
26
27
28
29
30
31
32
33
```

```
34 /*
35  * Player, associated with principal P. Their opponent is principal O.
36  * Allows the opponent to learn information about the board
37  * via declassification
38  */
39 public class Player[principal P, principal O] authority(P) {
40     private Board[{P->*}]{P->_} board;
41
42     private final List[{P->_}]{this} opponentQueries;
43
44     // Tell the opponent whether a ship exists at the queried coordinate
45     // This declassifies information about the board
46     boolean{P->_} processQuery(Coordinate[{P->_}]{P->_} query) {
47         boolean result = this.board.testPosition(query, new label {P->_});
48         return declassify(result, {P->*} to {P->_});
49     }
50 }
51
52 /*
53  * Handles the main game logic
54  */
55 public class BattleShip[principal P1, principal P2] authority (P1, P2) {
56     public void play(PrintStream[{}] output)
57     throws SecurityException, IllegalArgumentException
58     where authority(P1, P2) {
59         ...
60     }
61 }
62
63
64
65
66
67
68
69
70
71
72
73
```

```
74  /*
75   * Provides the main method
76   */
77  public class Main authority (Alice, Bob) {
78      public static final void main{Alice<-* meet Bob<-* meet p<-*}
79          (principal{} p, String[] args) : {Alice<-* meet Bob<-*}
80      throws SecurityException, IllegalArgumentException
81      where authority (Alice, Bob), caller(p)
82      {
83          PrintStream[{}] out = null;
84          try {
85              Runtime[p] runtime = Runtime[p].getRuntime();
86              out = runtime==null?null:runtime.stdout(new label {});
87          }
88          catch (SecurityException e) {
89              // just let out be null.
90          }
91
92          // the PrintStream needs to be labeled {Alice<-* meet Bob<-*}, which
93          // requires a declassification and an endorsement.
94          PrintStream[{}] out1
95                  = endorse(out, {p->*} to {p->*;Alice<-* meet Bob<-* meet p<-*});
96          PrintStream[{}] out2 = declassify(out1, {Alice<-* meet Bob<-*});
97
98          // instantiate an instance of the BattleShip game for Alice
99          // and Bob, and play it.
100         new BattleShip[Alice, Bob]().play(out2);
101     }
102 }
```

A.1.2 Paragon Implementation Sample

Listing A.2: Battleships Paragon Implementation

```
/*
 * Immutable (x, y) pair; overrides normal
 * Java hashCode(), equals(), toString()
 */
public class Coordinate {
    public ?policyof(this) final int x;
    public ?policyof(this) final int y;
}

/*
 * Immutable ship covering a line of coordinates
 * vertically or horizontally
 */
public class Ship {
    final ?policyof(this) Coordinate pos;
    final ?policyof(this) int length;
    final ?policyof(this) boolean isHorizontal;
}

/*
 * Mutable list of ships representing a player's board
 */
public class Board<policy shipPolicy> {
    private final ?policyof(this) LinkedList<Ship, shipPolicy> ships;
}
```

```
/*
 * Player, associated with principal P. Their opponent is principal O.
 * Allows the opponent to learn information about the board
 * via declassification
 */
public class Player {
    public final Player self = (Player)this;

    private lock allowBoardAccess(Player);
    public final policy boardPolicy = {
        self: ;
        Object x: allowBoardAccess(self)
    };

    private ?{Object x: } Board<boardPolicy> board;

    !{Object x: } boolean processQuery(?{Object x: } Coordinate query)
    throws -allowBoardAccess(self) !{Object x: } IllegalArgumentException {
        boolean result = this.board.testPosition(query);
        ?{Object x: } boolean declassified = false;
        open allowBoardAccess(self) {
            declassified = result;
        }
        return declassified;
    }
}

/*
 * Handles the main game logic
 */
public class BattleShip {
    public !{Object x: } void play(?{Object x:} PrintStream<{Object x:}> output)
    throws !{Object x: } IllegalArgumentException {
        ...
    }
}
```

```
77 /*
78  * Provides the main method
79  */
80 public class Main {
81     public static final !{Object x: } void main(String[] args)
82     throws !{Object x: } IllegalArgumentException {
83         ?{Object x: } PrintStream<{Object x: }> out = System.out;
84 
85         new BattleShip().play(out);
86     }
87 }
```

A.2 Case Study 2: Conference Management System

A.2.1 JIF Implementation Sample

Listing A.3: Conference Management JIF Implementation

```
/*
 * Immutable author of a paper
 */
public class Author {
    private final String{this} name;
}

/*
 * Immutable conference organiser (i.e. a privileged user)
 * This class is superfluous in JIF,
 * since there is no explicit object <-> principal connection
 */
public class Organiser {
    private final String{this} name;
}

/*
 * Immutable paper which may be submitted to a conference.
 * The author list can have a separate policy to the overall paper.
 */
public class Paper[label AuthorLabel] implements JifObject[AuthorLabel] {
    private final String{this} title;
    private final String{this} paperAbstract;
    private final Author{AuthorLabel}[]{AuthorLabel} authors;
}
```

```
32 /* The conference system which allows for
33  * paper submission and session allocation.
34  * O is the principal representing the organiser
35  */
36 public class ConferenceSystem[principal O] authority (O) {
37     private boolean{O->_} allocationsVisible;
38
39     private Paper[{O->*}]{O->_}[]{O->_} submissions;
40     private ArrayMap[{O->_}, {O->*}] allocations;
41
42     // Performs declassification of the session number
43     // *if* the timed release has occurred
44     private int{O->_}
45     getSessionNumber{O->_}(Paper[O->_]{O->_} paper) : {O->_}
46     where authority(O) {
47         if (paper == null) return -1;
48         String{O->_} title = paper.getTitle();
49         if (title == null) return -1;
50
51         if (!allocationsVisible) {
52             return -1;
53         }
54
55         JifString[{O->_}] titleObj = new JifString[{O->_}](title);
56         JifObject[{O->*}]{O->*} sNoObj;
57         try {
58             sNoObj = allocations.get(new label {O->*}, titleObj);
59         } catch (NullPointerException e) {
60             return -2;
61         }
62         // Declassify the object retrieved from the map
63         JifObject[{O->*}]{O->_} sNoObjDeclass
64                     = declassify(sNoObj, {O->*} to {O->_});
65         // Convert it to a JifInteger
66         JifInteger[{O->*}]{O->_} sNo;
67         try {
68             sNo = (JifInteger[{O->*}])sNoObjDeclass;
69         } catch (ClassCastException ex) {
70             return -3;
71         }
```

```
72          // Pull the int from the JifInteger
73          int{O->*} r;
74          try {
75              r = sNo.intValue();
76          } catch (NullPointerException ex) {
77              return -4;
78          }
79          // Declassify the int
80          int{O->_} result = declassify(r, {O->*} to {O->_});
81          return result;
82      }
83
84      public void setAllocationsVisible() {
85          allocationsVisible = true;
86      }
87  }
88
89  public class Main authority (Organiser) {
90
91      public static void main(principal p, String[] args) {
92          ConferenceSystem conference = new ConferenceSystem();
93
94          // Add some papers to the conference
95          // (submitted papers are also added to a list)
96          setupDummyData(conference);
97
98          // Attempt to print out the allocations
99          // All session allocations will show as -1 due to runtime check
100         printAllocations(conference);
101
102         // Perform timed release - information is no longer secret
103         conference.setAllocationsVisible();
104
105         // Print out the allocations:
106         // will now print out the real session allocations
107         printAllocations(conference);
108     }
109
110 }
```

A.2.2 Paragon Implementation Sample

Listing A.4: Conference Management Paragon Implementation

```
/*
 * Immutable author of a paper
 */
public class Author {
    private final ?policyof(this) String name;
}

/*
 * Immutable conference organiser (i.e. a privileged user)
 */
public class Organiser {
    public final ?policyof(this) String name;
}

/*
 * Immutable paper which may be submitted to a conference.
 * The author list can have a separate policy to the overall paper.
 */
public class Paper<policy authorRestriction> {
    public final ?policyof(this) String title;
    public final ?policyof(this) String paperAbstract;
    private final ?authorRestriction Author[]<authorRestriction> authors;

}
```

```
37 /*
38  * Conference system which allows for paper submission
39  * and session allocation. Imposes 'timed release' policy
40  * on session allocations and author lists.
41  */
42 public class ConferenceSystem {
43     public lock allocationsVisible;
44     public static final policy bottom = {Object x: }
45     public static final policy ifAllocationsVisible = {
46         Organiser o: ;
47         Object x: allocationsVisible
48     };
49
50     private ?bottom LinkedList<Paper<ifAllocationsVisible>, bottom>
51                                                             submissions;
52
53     private ?bottom HashMap<Paper, bottom, Integer, ifAllocationsVisible>
54                                                             allocations;
55
56     public ?bottom int
57     getSessionNumber(?bottom Paper<ifAllocationsVisible> paper) {
58         if (allocationsVisible) {
59             return (int)allocations.get(paper);
60         } else {
61             return -1;
62         }
63     }
64
65     public +allocationsVisible !bottom void setAllocationsVisible() {
66         open allocationsVisible;
67     }
68 }
69
70
71
72
73
74
75
76
```

```
77 /*
78  * The main class which runs a conference and demonstrates
79  * where allocations and author lists may and may not be printed out
80  */
81 public class Main {
82     private static final policy bottom = {Object x :};
83 
84     public static !bottom void main(String[] args) {
85         ConferenceSystem conference = new ConferenceSystem();
86         LinkedList<Paper<ConferenceSystem.ifAllocationsVisible>, bottom>
87                                                     conferencePapers;
88         // Add some papers to the conference
89         // (submitted papers are also added to a list)
90         setupDummyData(conference, conferencePapers);
91 
92         // Attempt to print out the allocations
93         // All session allocations will show as -1 due to
94         // runtime check in ConferenceSystem.getSessionNumber
95         printAllocations(conference);
96 
97         // Cannot print out the authors of the papers:
98         // would cause compile error due to author list restriction
99         // printOutAuthors(conferencePapers);
100 
101         // Perform timed release - information is no longer secret
102         conference.setAllocationsVisible();
103 
104         // Print out the allocations:
105         // will now print out the real session allocations
106         printAllocations(conference);
107 
108         // Can now print out the authors of the papers
109         printOutAuthors(conferencePapers);
110     }
111 }
```

A.3 Case Study 3: Calendar Scheduler

A.3.1 JIF Implementation Sample

Listing A.5: Calendar Scheduler JIF Implementation

```
/*
 * Immutable user in the system
 * Complicated in JIF since dynamic principals don't work as in Paragon
 */
public class User {
    public final String{this} name;
}

/*
 * Represents a single meeting at a time, title and list of attendees
 * Simplified policy: meeting title only visible to the owner
 * Implemented here using a type parameter
 */
public class Meeting[principal Owner] {
    public final String{Owner->*} title;
    public final int{Owner->_} day;
    public final int{Owner->_} startHour;
    public final int{Owner->_} endHour;

    public String{Owner->*} toString() {
        ...
    }
}
```

```
34 /*
35  * A calendar of meetings belonging to a user
36  * The user is represented by the Owner principal type parameter
37  */
38 public class Calendar[principal Owner] {
39     private LinkedList[{Owner->_}]{Owner->_} meetings;
40
41     public boolean{Owner->_; day; startHour; endHour}
42     freeAtTime(int day, int startHour, int endHour) {
43         ...
44     }
45
46     public String{Owner->*} getCalendarRepresentation() {
47         try {
48             String s = "Calendar:\n---";
49
50             for (int i = 0; i < meetings.size(); ++i) {
51                 JifObjectWrapper[{Owner->_}] w
52                         = (JifObjectWrapper[{Owner->_}])meetings.get(i);
53                 Meeting[Owner] m = (Meeting[Owner])w.getObject();
54                 s += "\n";
55                 s += m.toString();
56             }
57             return s;
58         } catch (IndexOutOfBoundsException ex) {
59             return "";
60         } catch (ClassCastException ex) {
61             return "";
62         } catch (NullPointerException ex) {
63             return "";
64         }
65     }
66 }
67
68 /* Scheduler system which has many users with calendars */
69 public class Scheduler {
70     // Implementation fails here: need to
71     // dynamically associate a principal with a user
72     private HashMap[{_->_}, {_->_}]{_->_} calendars;
73 }
```

```
74
75  /*
76   * The main class which creates a scheduler and demonstrates
77   * the security policy
78   */
79  public class Main authority (Alice, Bob, Charles) {
80      Scheduler scheduler = new Scheduler;
81
82      public static void main(principal{} p, String[] args)
83      throws NullPointerException, IllegalArgumentException
84      where authority(Alice, Bob, Charles), caller(p) {
85          PrintStream[{}] out = getStandardOut();
86          Scheduler scheduler = new Scheduler();
87          setupDummyData(scheduler);
88
89          LinkedList[{_->_}]{_->_} allUsers = new LinkedList[{_->_}]();
90          addAllUsers(allUsers);
91
92          // Times of meetings are not classified
93          // and so can be printed out freely
94          boolean scheduleCheck1 = theScheduler.attendeesFreeAtTime(allUsers,
95          Meeting.getDayValue("Monday"), 9, 10);
96          out.println("Are all users available at 9 - 10 on a Monday? "
97                      + scheduleCheck1);
98          boolean scheduleCheck2 = theScheduler.attendeesFreeAtTime(allUsers,
99                  Meeting.getDayValue("Tuesday"), 9, 10);
100         out.println("Are all users available at 9 - 10 on a Tuesday? "
101                     + scheduleCheck2);
102
103         // Cannot construct an output printstream for any principal
104         // other than p. In JIF, attempting to do so just fails silently,
105         // setting the stream to null
106
107         // Attempting to print out Alice's calendar to standard out
108         // should fail! But it does not - JIF can't properly analyse
109         // the dynamic use of Alice
110         out.println(scheduler.getCalendar((Principal)Alice)
111                                 .getCalendarRepresentation());
112     }
113 }
```

A.3.2 Paragon Implementation Sample

Listing A.6: Calendar Scheduler Paragon Implementation

```
/* Immutable user in the system */
public class User {
    public final ?policyof(this) String name;
}

/* Represents a meeting with a time visible only to attendees */
public class Meeting {
    public static final policy bottom = {Object x: };
    // Lock which is open when a user is attending a given meeting
    public ?{User u:} lock isAttending(User, Meeting);
    public final policy attendeesOnly = {User u: isAttending(u, self)};

    public final ?attendeesOnly String title;
    public final ?policyof(this) int day;
    public final ?policyof(this) int startHour;
    public final ?policyof(this) int endHour;
    private notnull ?bottom LinkedList<User, bottom> users;

    public !bottom Meeting(?policyof(this) int day,
                           ?policyof(this) int startHour,
                           ?policyof(this) int endHour,
                           ?attendeesOnly String title,
                           ?bottom LinkedList<User, bottom> attendees) {
        ...
        for (int i = 0; i < attendees.size(); ++i) {
            User u = attendees.get(i);
            // Indicate that this user is attending the meeting
            open isAttending(u, self);
            this.users.add(u);
        }
    }

    public ?(policyof(this) * attendeesOnly) String toString() {
        ...
    }
}
```

```
37
38 /*
39  * A calendar of meetings belonging to a user
40  */
41 public class Calendar {
42     public static final policy bottom = {Object x: };
43     public final User user;
44     private LinkedList<Meeting, bottom> meetings;
45
46     public ?(policyof(day) * policyof(startHour) * policyof(endHour)) boolean
47     freeAtTime(int day, int startHour, int endHour) {
48         ...
49     }
50
51     public ?{viewer: } String getCalendarRepresentation(notnull User viewer) {
52         ?{viewer: } String s = user.toString() + "'s Calendar:\n---";
53         for (int i = 0; i < meetings.size(); ++i) {
54             Meeting m = meetings.get(i);
55             if (Meeting.isAttending(viewer, m)) {
56                 s += m.toString();
57             }
58         }
59         return s;
60     }
61 }
62
63 /*
64  * Scheduler system which has many users with calendars
65  */
66 public class Scheduler {
67     public static final policy bottom = {Object x:};
68     private ?bottom HashMap<User, bottom, Calendar, bottom> calendars;
69 }
70
71
72
73
74
75
76
```

```
77 /* The main class demonstrating the scheduler */
78 public class Main {
79     public static final policy bottom = {Object x:};
80     private static final ?bottom User alice = new User("Alice");
81     private static final ?bottom User bob = new User("Bob");
82     private static final ?bottom User charles = new User("Charles");
83
84     public static !bottom void main(String[] args)
85     throws !bottom NullPointerException {
86         Scheduler scheduler = new Scheduler();
87         setupDummyData(scheduler); // Setup dummy calendars and meetings
88         // Mock 'private' output for Alice
89         IOChannel<bottom, alice> aliceOut = new IOChannel<bottom, alice>();
90
91         // Times of meetings are not classified: they may be printed freely
92         LinkedList<User, bottom> allUsers = new LinkedList<User, bottom>();
93         addAllUsers(allUsers);
94         boolean scheduleCheck1 = theScheduler.attendeesFreeAtTime(allUsers,
95                 Meeting.getDayValue("Monday"), 9, 10);
96         System.out.println("Are all users available at 9 - 10 on a Monday? "
97                             + scheduleCheck1);
98         boolean scheduleCheck2 = theScheduler.attendeesFreeAtTime(allUsers,
99                 Meeting.getDayValue("Tuesday"), 9, 10);
100        System.out.println("Are all users available at 9 - 10 on a Tuesday? "
101                            + scheduleCheck2);
102
103        // A user may print out every event in their own calendar,
104        // since they attend all of them
105        aliceOut.println(scheduler.getCalendar(alice)
106                            .getCalendarRepresentation(alice));
107        // Printing out another user's calendar will result in
108        // some meetings being omitted. Only meetings this user attends
109        // on the other user's calendar should be visible
110        aliceOut.println(scheduler.getCalendar(bob)
111                            .getCalendarRepresentation(bob));
112
113        // Actual effect: the full calendar prints out both times
114        // The type checker thinks alice and bob are the same 'actor'
115    }
116 }
```