From Languages to Systems: Understanding Practical Application Development in Security-typed Languages

Boniface Hicks, Kiyan Ahmadizadeh and Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory (SIIS)
Computer Science and Engineering, Pennsylvania State University
{phicks,ahmadiza,mcdaniel}@cse.psu.edu

Abstract

Security-typed languages are an evolving tool for implementing systems with provable security guarantees. However, to date, these tools have only been used to build simple "toy" programs. In this paper, we explore the process and machinery of building provably secure applications using security-typed languages. We develop a secure email system in the Java language variant Jif. Real world policies are mapped onto the information flows controlled by the language primitives, and we consider the process and tractability of broadly enforcing security policy in commodity applications. In this investigation, we found that while the language provided the rudimentary tools to achieve low-level security goals, additional tools, services, and language extensions were necessary to formulate and enforce application policy. We detail the design and use of these tools. This work serves as a starting point–we have demonstrated that it is possible to implement real world systems and policy using security-typed languages. However, further investigation of the developer tools and supporting policy infrastructure is necessary before they can fulfill their considerable promise of enabling more secure systems.

1 Introduction

The exposure of private data is an increasingly critical concern of online organizations [8, 9]. The huge costs of exposure can be measured both in financial and in human terms. The central cause is, of course, the systems themselves. The security provided by existing systems is largely due to secure design and implementation–practices that have yet to fully mature. Furthermore, the subsequent evaluation of these systems relies on ad hoc or inexact quality and assurance evaluations. What are needed are tools for formulating and ensuring more precise notions of security. *Security-typed languages* fulfill this need.

Security-typed languages annotate source code with security levels on types [33] such that the compiler can statically guarantee that the program will enforce *noninterference* [14]. In a broader sense, these languages provide a means of provably enforcing a security policy. Theoretical models for security-typed languages have been actively studied and are continuing to evolve. For example, researchers are extending these models to include new features, such as exceptions, polymorphism, objects, inheritance, side-effects, threads, encryption, and many more [25].

However, developer tools and programming experience have not evolved in concert with language features. There are currently only two significant language implementations, Flow Caml [28] and Jif [22] and only two applications [1, 22], both written in Jif. The literature frequently postulates on practical, distributed applications with many principals and complex policy models such as tax preparation [20], medical databases [30] and banking systems [31]. However, the only completed applications have both been "toy" applications with only two principals within a simplistic distributed environment. For this reason, many important language features such as dynamic principals and declassification, as well as integration with conventional security mechanisms such as cryptography, certificates and certificate authorities are yet to be fully explored.

To address this lack of practical experience, we set out to build a realistic application in a security-typed language. We sought to discover whether this tool

for secure programming could hold up to its promise of delivering real-world applications with strong security guarantees. We conducted this experiment by implementing an email system in the language Jif, a security-typed variant of Java. Throughout this paper, we reflect on the advantages and limitations of these language-based security tools and the requirements of future system development.

A principal result of this study is that while language tools were robust and expressive, additional development and runtime tools were necessary. We extend the language with additional policy formulation tools (a policy compiler [15]) and runtime support infrastructure (policy store) to enable the enforcement of policy in a distributed environment. We also provide tools for secure software engineering including a Jif IDE in the Eclipse extensible development platform. Finally, we provide a critical evaluation of the Jif language, highlighting its effectiveness at carrying out the promised security goals, the difficulties involved in using it and the ways in which it still needs improvement.

In a sense this paper is the second chapter in an ongoing story, because we are not the first to offer our experience with Jif. Askarov and Sabelfeld gave the first detailed experience paper on Jif last year [1]. Various aspects of this work set it apart from theirs. These center around the fact that our goals were divergent—while they sought to verify the security properties of a cryptographic protocol, we have sought to build a real, distributed, interactive application, which interoperates with other existing applications. As a consequence of their goal, the policy used in their program is a relatively simple, two-principal policy. In contrast, a significant part of the contribution of this work lies in the tools we have implemented for handling more complex and expressive distributed policies. Another contribution includes the cryptographic infrastructure we develop which was necessitated by the distributed, real-world nature of our application.

The remainder of this paper is organized as follows. We begin in the next section by providing a sketch of an email system and the kinds of security policies it requires. Section 3 discusses the details of our implementation in Jif, identifying the challenges we faced. Section 4 describes in detail the tools we have built to overcome these challenges. Section 5 discusses our experience with Jif, evaluates the difficulty and effectiveness of using Jif for building an email client and indicates some areas of Jif which need improvement. A number of related works are discussed in Section 6. We conclude in Section 7.

2 Overview

This section presents an overview of the architecture of the Jif/Policy email system (*JPmail*), its operation, and its security goals.

2.1 JPmail

An email system is particularly useful for the study of application development in security-typed languages. This is not only because email is ubiquitous, but also because it has been a frequent avenue for security leaks [24, 8]. Moreover, email has a wide variety of security policies that it might need to enforce: including policies from military multi-level security (MLS) [3] to organizational hierarchies [12]. Finally, email policy is naturally distributed, with unique principals interacting across potentially geographically distant clients. Through our design, we seek to flexibly support these policies that involve diverse and dynamic principals.

Illustrated in Figure 1, the *JPmail system* consists of three main components: JPmail clients, the Internet and public mail servers. Written in Jif, the *JPmail client* (or just JPmail throughout) is a functional email client implementing a subset of the MIME protocol. The JPmail client software consists of three software components: a POP3-based mail reader, an SMTP-based mail sender and a policy store. The client provably enforces security policy from end to end (sender to recipient). Policy is defined with respect to a principal hierarchy. Each environment defines principal hierarchies representative of their organizational rights structure.

We make the following assumptions about the operating environment. The JPmail-local file systems are trusted to store information securely, based on the access control list on a given file (thus if a file is readable only by the user, it is considered safe from leakage). Internet communication is generally untrustworthy, and is deemed as *public* channels throughout. The SMTP and POP3 servers are not written in Jif, and do not enforce any security policy save that which is provided by their implementation and administration. For the purposes of this work, we assume noth-

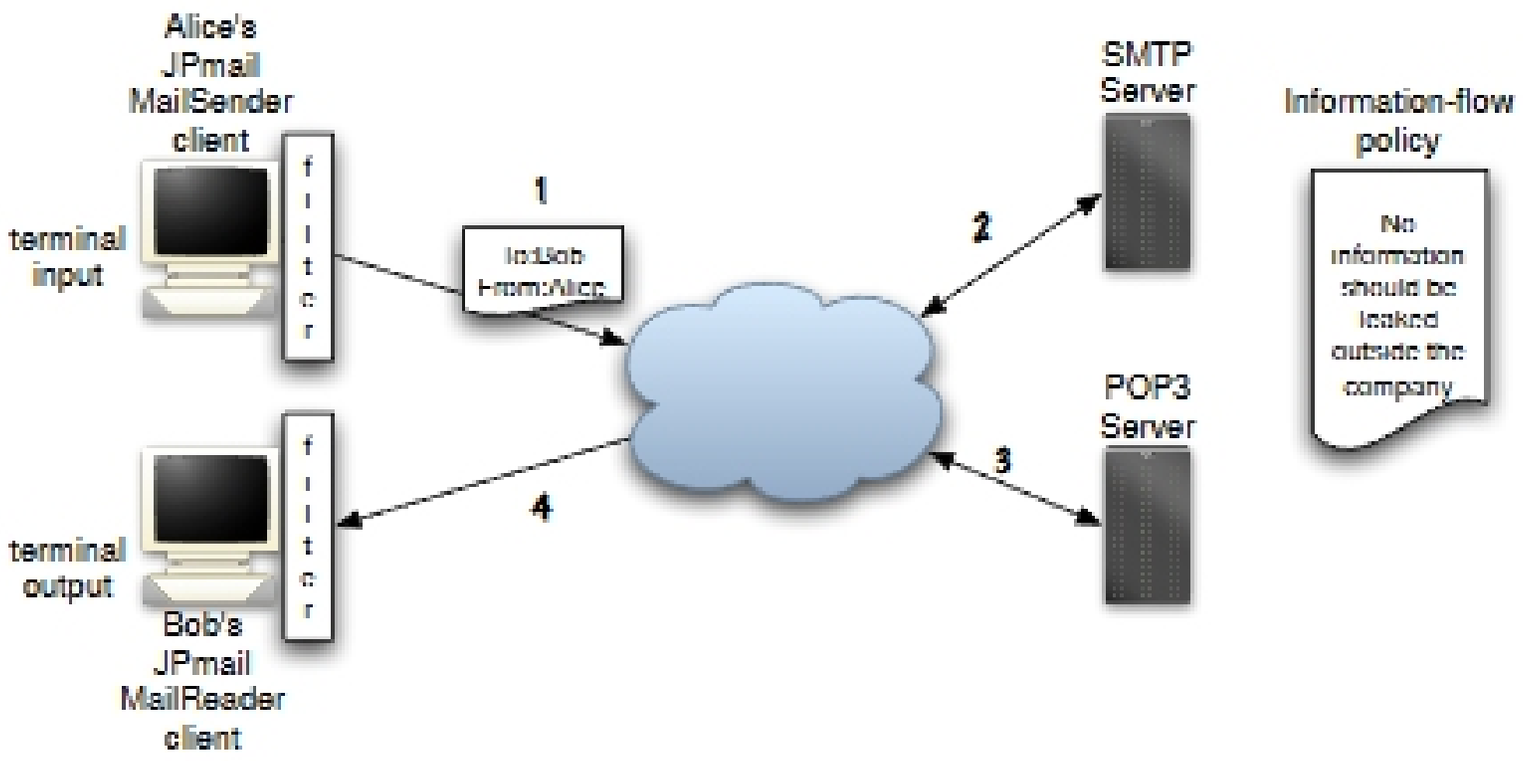

Figure 1: Sending email requires four steps: 1) Alice's enters email and sends through the Internet to an SMTP server, 2) The SMTP Server sends the email to the POP3 server associated with Bob's address, 3) Bob retrieves the mail from his POP3 server.

ing about the servers' ability to prevent leakage of user data: i.e., any information sent to them is deemed *public*.

2.2 Security policy

The single real-world security policy we defined at the outset of this work was seemingly simple:

> *The body of an email should be visible only to the authorized senders and receivers.*

However, provably realizing this policy was more complex than it would initially appear.

We make two clarifications about this policy. Firstly, in this paper, we are only concerned with privacy (confidentiality). This is because the most recent release of Jif only handles confidentiality properties. As Jif evolves (the next release will include integrity [6]), our models will also evolve. Secondly, our email client is not inherently limited to sending email only to authorized receivers. The way the client handles unauthorized recipients depends on the user-defined policy that is in place (we consider this in more detail in Section 3.2).

We centrally enforce policy within the client through *noninterference modulo trusted declassification*. Developed in our prior work [15], this property parameterizes a security policy based on the *declassifiers* that are explicitly trusted by the principals in an application. Declassifiers are the set of declassifying filters that must be used in order for an application to function (such as the filters which make the headers public or make the body public through encryption). The allowed declassifiers for each principal are defined in a separate policy file which is integrated into our Jif application using policy tools developed for this work (see Section 4.2 for more detail). This security policy states that all leakage occurs through explicitly allowed declassifiers. This has the benefit of reducing security analysis to an analysis of only trusted declassifiers. Note that this mapping between principals, declassifiers, and objects is reminiscent of the TP-user-CDI access matrices in the Clark-Wilson integrity model [7] (though our policy is, as noted, focused on confidentiality rather than integrity).

2.3 The Jif programming language

Jif is an object-oriented, strongly-typed language based[1] on Java. In Jif, the programmer must label types with security annotations according to the decentralized label model (DLM) [21]. The compiler uses these annotations during type-checking to ensure noninterference. For example, assuming `alice` and `bob` are principals, `{alice:}` is a DLM-label in Jif

[1] Jif does not provide support for inner classes or threads, because of the ways they complicate information flow analysis. Jif is described most completely by Myers [19], has online documentation at `www.cs.cornell.edu/jif/` and a helpful, practical overview, along with expository examples, is given by Askarov and Sabelfeld [1].