# 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. As described in this paper, we have developed the first real-world, security-typed application: a secure email system written 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. We find 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. We also show how the strong guarantees of Jif in conjunction with our policy tools can be used to evaluate security. 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 [7, 8]. 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 [28] such that the compiler can statically guarantee that the program will enforce *noninterference* [11]. 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 [21].

Developer tools and programming experience have not evolved in concert with language features. There are currently only two significant language implementations, Flow Caml [24] and Jif [18] and only two applications [1, 18], both written in Jif. The literature frequently postulates on practical, distributed applications with many principals and complex policy models such as tax preparation [16], medical databases [25] and banking systems [26]. However, the only completed applications have both been "toy" applications with only two principals within a simplistic distributed environment. For this reason, many language features such as dynamic principals and declassification, as well as integration with conventional security mechanisms such as cryptography, certificates, certificate authorities and network authentication protocols were yet to be explored (prior to this work).

To address this lack of practical experience, we 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. Two key criteria we used for defining "real-world" were that 1) the application should interact with other non-security-typed, networked components while still maintaining the security policy of its data and 2) the security policy should be easily re-configurable such that the application could be of general use (not just in a military, MLS setting, but also in a corporate setting, for example). We conducted this experiment by implementing an *email system* in the language Jif,

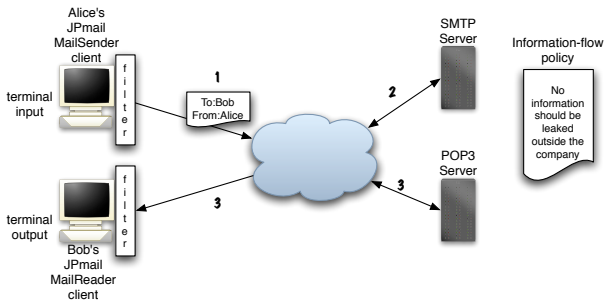


**Figure 1. Sending email**

a security-typed variant of Java. Throughout, we reflect on the advantages and limitations of language-based security tools and the requirements of future development.

A principal result of this study is that we succeeded in developing a real-world application for which we can easily assess that there is no information leakage beyond what is allowed by a clear, user-defined, high-level policy. We found 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 [12]) 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 integrated development environment (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.

The remainder of this paper is organized as follows. We begin in the next section by providing a sketch of an email system, the threats it faces and the kinds of security policies it requires. Section 3 discusses the security that can be provided by Jif, the limitations of Jif and some solutions to these limitations. Section 4 concisely describes the architecture of our JPmail system. Section 5 describes in detail the tools we have built to overcome these challenges. Section 6 provides a limited security evaluation of our email client, 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 7. We conclude in Section 8.

## 2 Overview

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 [20, 7]. 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 [9]. Finally, email policy is naturally distributed, with unique principals interacting across potentially distant clients. We seek to support policies that involve these diverse and dynamic principals.

Illustrated in Figure 1, the *JPmail system* (*JP = Jif/Policy*) 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.

### 2.1 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 until June of 2006, Jif could only handle confidentiality properties (the most recent release introduces integrity labels [5]). Future revision of our work will also embrace integrity. Secondly, our email client is not inherently limited to sending email only to authorized receivers. The way JPmail handles unauthorized recipients depends on the user-defined policy (see Section 5.1).

We make the following assumptions. 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 nothing about the servers' ability to prevent leakage of user data: i.e., any information sent to them is deemed *public*.

Consider some dangers in email. 1) In the case of a malicious insider, email was used to leak classified documents [20]. 2) In another case, a programmer mistake led to a privacy violation for a list of patients using anti-depressant medication [7]. 3) An email application also handles passwords for logging into remote servers and could leak a pass-

word by sending it to the server as plaintext (a protocol that some servers use, in fact). 4) An email client which uses PGP or other systems could accidentally or maliciously leak keys.

Given these threats, which involve both malice and mistakes (on the part of both the programmer and the user), how can one be sure that an email client is safe to use? The answer to this critical question lies in two realms: the proper configuration of an email policy and the application's faithful, verifiable implementation of that policy. How should this verification be done? It is not unreasonable to verify some of this by hand, but the parts that are verified by hand should be small and straight-forward. It would be desirable to be able to verify the remainder of these complex systems automatically.

Jif provides the basis for this in performing automated verification of information flows through security-annotated type-checking. For this reason, it promises to be a powerful, key tool for developing secure applications. At the same time, however, Jif has practical limitations when it is being used to build components in a distributed system. In the next section, we explore these features and limitations.

## 3 Building a secure application with Jif

Jif is an object-oriented, strongly-typed language based[1] on Java. In Jif, the programmer labels types with security annotations according to the decentralized label model (DLM) [17]. 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 syntax indicating that a particular value is owned and readable only by `alice`. Thus, the following code would produce a type error, because it attempts an illegal flow of information from a sensitive string owned by `alice` to a string owned by `bob`.

```
String{alice:} password = "1fth2;zg";
String{bob:} leak = password; // causes error
```

This provides the starting point for implementing a secure email client in Jif. It suggests that if we properly label the data we want to keep secret (the bodies of emails, passwords and keys), then Jif will handle the rest. Jif implements a single, strong, information-flow policy—noninterference—parameterized by principals and delegations. One of the advantages of noninterference is that it is an end-to-end policy (the same policy applies for the whole lifetime of data—from its creation to its destruction). Consider the following code for an email data structure :

```
public class Email {
   String{} toAddress;
   String{} fromAddress;
   String{this} body;
   public Email(String{} to, String{} from,
                String{this} body) { ... }
}
```

If Alice wants to send an email to Bob, she could use the following declaration:

```
Email{bob:} msgToBob = new Email(
   "bob@psu.edu","alice@psu.edu","Hi Bob!");
```

Here, the email headers are public (`{}` is the Jif syntax for a public label) and the body of the email will be labeled `{bob:}` (since the `{this}` label in the class definition is always replaced with whatever label is used when an instance of this class is created). Suppose that a delegation also exists from Bob to his wife Charlotte. Under a strict noninterference policy, we could be certain, based solely on this declaration, that no one but Bob (and Charlotte, to whom he delegates) could ever read the body of this email. Furthermore, Jif prevents the programmer from leaking information through email. For example, the following code would generate an error, because `password` is labeled as `{alice:}` while the constructor for `Email` requires that the body be labeled `{this}` (which is `{bob:}` in this case):

```
Email{bob:} msgToBob = new Email(
   "bob@psu.edu","alice@psu.edu",password); // error
```

Finally, observe one more important property of Jif: compositionality. Jif requires that a method's information flows be accurately indicated on the method header and then verifies that the header and the information flows in the body are consistent. After that, the body never needs to be examined again by the type checker. These analyses are used in the later evaluation of calling functions. Thus, we recursively build upon smaller analyses toward a total view of system information flow.

Detailed below, Jif presents several challenges as a tool for system development.

### 3.1 A principal store

One challenge is in managing principals beyond the limited domain of a single Jif program execution. Principals need to be defined explicitly in the program, along with the policies they enforce (whether they allow certain declassifiers, e.g.). Furthermore, for our email client to be useful in practice, the principals persist beyond a single execution so that labeled data may leave one Jif application (through a network socket, for example) and later re-enter another one. Intuitively, these principals should be anchored in principals in the real-world, e.g., users of the parent operating system.

This is a problem that the past simple Jif demonstration applications [1, 18] did not face, because they did not communicate with the non-Jif world and they used trivially simple policies. Being merely games, they only had to define

[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 [15], 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].

principals "me" and "opponent" and those principals only needed to have meaning for the duration of one execution of the program. In our application, to the contrary, it is necessary to utilize principals with persistent meaning across multiple applications. Additionally, for these principals to be robust, it should not be possible for the user to impersonate a principal illegally.

One solution would be to map operating system principals one-to-one onto Jif principals and leverage the existing technologies of distributed systems. It is desirable, however, that there be a one-to-many or many-to-one mapping between our application's principals and operating system principals, in order that our application can be sufficiently general. For example, in an MLS setting, it might be desirable for many different users to be able to send "secret" mail—this is a many-to-one mapping. Conversely, a single user should be able to take on the role of "classified" and of "top secret"–this is a one-to-many mapping. Furthermore, defining a distinct principal-space for a Jif application frees the application from being tied to a particular operating system instance. This brings us to the following technical challenge:

**Problem 1** Jif principals must be developed which are persistent across multiple executions of an application and consistent across multiple applications and operating systems. Jif principals must also be unique so that they cannot be impersonated.

**Observation 1** A public-key infrastructure (PKI) provides this uniqueness, persistence and consistency.

**Solution 1** By mapping Jif principals to public/private key pairs and leveraging existing PKI technologies for creation of certificate authority and public key certificates, we describe a principal-space with the desired properties. Furthermore, we prevent illegal impersonation of a principal by requiring that a user have access to the principal's private key before taking on that role. We describe this in more detail in Section 5.1

### 3.2 A policy store

Returning to the above code, let us see how Jif prevents illegal information leaks. Consider this code fragment in which `msgToBob` is sent out on a Socket:

```
JifSocketFactory socketFactory =
   new JifSocketFactory();
Socket[{}] outchannel =
   socketFactory.createSocket(mailhost,mailport);
outchannel.write(msgToBob); // causes error
```

In general, a `Socket` could be trusted to keep a certain level of data confidential (using IPsec in a trusted operating system, e.g.) and so the `Socket` class is parameterized by a label (*class parameterization* is indicated with `[]`'s). In this case, the label must be `{}`, because part of our security policy (as stated in Section 2.1) is that we don't trust the internet to keep our data confidential. We implement this by requiring that our `socketFactory` only return public sockets. Then Jif can catch security violations such as the one above. A socket's `write` method requires that input parameters are no more secret than the label on the socket. Thus, trying to send `msgToBob`, whose label `{bob:}` $\not\leq$ `{}`[2], causes an error.

This brings us to the most serious, practical problem with the code above: this email could never be sent to Bob! Because it is labeled as `{bob:}`, Jif prevents it from being placed on a public channel and sent to the SMTP server. The only way around this would be if there were a channel directly to Bob that no one else could see, but this would preclude using existing mail servers and existing networks. Another obvious solution would be to use encryption. However, under the strict noninterference policy, even encryption would be disallowed, because putting a ciphertext on a public channel is a possibilistic leak, releasing a small amount of information about the plaintext.

We might decide that the information leaked through encryption is an acceptable leak, however. Then a Jif solution is to relax the policy slightly through declassification. For this purpose, Jif provides a primitive, the `declassify`-statement:

```
outchannel.write(
   declassify(AES.encrypt(key,msgToBob), {}));
```

This introduces a new problem. Although we have successfully published the email, we have now lost the meaning of the policy `{bob:}`. Allowing *any* relaxations of the policy leaves the programmer wondering what the new policy actually is. The label `{bob:}` no longer means that only Bob and Charlotte can read the data. It now means that only Bob and Charlotte can read the data, modulo some information about the data that might be released by some declassification statements somewhere in the program. This is problematic, because the declassification statements have nothing to limit them and could actually release all the information to any security level, including public. At the same time, it is not a total loss, because we know, at least, that the information could only be leaked through declassification statements. A security analysis of Jif only needs to focus on the declassification statements to gauge whether the information leakage is dangerous or unacceptable. Such an analysis was done in the `jifpoker` case study [1].

The security analysis would be easier and safer, however, if it could be localized to a small, single policy file, separate from the application itself. Rather than treating every declassification as a potential wildcard, it is possible to place some limits on the allowed kinds of declassification for a particular principal and specify these in a small policy file. For example, the policy file might specify that Bob's data can only be declassified if it is also encrypted. This restores

[2]The $\not\leq$ operator indicates that bob does not delegate to public.

local meaning to a label such as {bob:}. Consider an example policy file, focusing only on Bob's policy statements:

```
bob -> charlotte % Bob trusts his wife with all data
bob allows smtp.DeclassMsgBody(family)
bob allows crypto.AES(public)
bob allows crypto.MD5(public)

% Bob's children
family -> john
family -> sarah
```

With this policy file, when the programmer sees that the email is labeled {bob:}, she knows that this email is limited in the ways it can flow: it can be sent on public channels, but only if it is encrypted or hashed first. Bob can send information via Email, but only to his family. This eliminates the need to scour the code for all declassifiers that could leak Bob's data. The policy file states explicitly which declassifiers are allowed.

**Problem 2** Connecting with non-security-typed components requires declassification, but it introduces confusion about the security policy of a program and potentially broadens information leakage.

**Observation 2** Based on this, we make the observation that in order to understand the meaning of security-policy labels in a security-typed program with declassification, it is necessary to know three things: 1) the principals used in the program, 2) the delegations they make and 3) the declassifiers they trust. With this information in hand, the meaning of the policy {bob:} is restored. If we know, for example, that Bob delegates to no one and trusts only AES encryption, then we know that the only information which will be released about the body of this email will be the extremely small amount of information released by AES encryption.

**Solution 2** To address this challenge, we added a policy infrastructure to Jif that allows the programmer to define, up front, the principals, delegations and declassifiers which may be used in a program. We describe this infrastructure in Section 5.1; we give even more detail about this policy infrastructure in a technical report [12], including a proof of the security it maintains.

### 3.3 Certified user input

Consider again the code for an email data structure, given above. If Alice wishes to send an email to Bob, she must first type in the email from her terminal. Thus, the email text enters the JPmail client from an input stream, stdin, labeled {alice:}. If she then wishes to send this string to Bob, it must be relabeled to {bob:}. Let us also introduce a new concept, a *dynamic principal*, which allows the sending of an email to be parameterized based on two dynamic values: the user who is sending and the chosen recipient.

```
Email{rcpt:} send(String{} to, String{} from,
                  Principal user, Principal rcpt) {
  String{user:} body = stdin.readLine();
  Email{rcpt:} msg =
    new Email(to,from, declassify(body,{rcpt:});
  return msg; }
```

Thus, if alice and bob are principals defined elsewhere, the email could be created and sent as follows:

```
Email{bob:} msgToBob = Email.send(
    "bob@psu.edu","alice@psu.edu",alice,bob);
outchannel.write(
   declassify(AES.encrypt(key,msgToBob), {}));
```

Here again, we need declassifying filters. In this case, we need to leak more information than in the encryption declassifier described above—we need to leak the body of the text. Should such leakage be possible? **This is a policy decision that should not be buried in the code, but should be declared at a high level.**

The answer depends on the security model. In an MLS setting, this should not be possible unless Alice and Bob are both working at the appropriate relative security levels. In other words, this declassification should not be allowed at all and the method Email.send(...) should return null unless user delegates to rcpt, written rcpt actsfor user in Jif (e.g. in an MLS setting, user could be secret and rcpt could be secret or top-secret). In a corporate setting, it may be acceptable to declassify email text so that anyone in the company can read it. If it is going to an external principal, it may be necessary to perform an audit or add a disclaimer. We accommodate such security policies in the following way[3]:

```
Email{rcpt:} send(String{} to, String{} from,
                  Principal user, Principal rcpt) {
  String{user:} bodyIn = stdin.readLine();
  String{rcpt:} body = null;
  if (rcpt actsfor user) body = bodyIn;
  else if (authorize(user,rcpt,DeclassMsgBody))
    body = DeclassMsgBody(user,rcpt,bodyIn);
  else if (authorize(user,rcpt,DeclMsgBodyAudit))
    body = DeclMsgBodyAudit(user,rcpt,bodyIn);

  Email{rcpt:} msg = new Email(to,from,body);
  return msg; }
```

The authorize method checks whether the principal in the first argument trusts the declassifier (third argument) to declassify information to the principal in the second argument. Thus, an MLS policy should not allow either declassifier to be used, while a company policy may allow DeclassMsgBody if both principals are in the company and DeclMsgBodyAudit if the first principal is in the company, but the recipient is external. These details are specified in a policy file, which is compiled into Jif with our policy compiler and

[3]Note that this code does not correspond directly to the Jif implementation. We use Jif Closures for this which are such flexible constructions that they become syntactically cumbersome. We present a syntactically simplified but semantically equivalent form here to aid the reader.

established at the start of a Jif application. By teasing out the policy, we have made it possible to change the policy model of an application merely by changing the high-level policy file.

In the above code, we glossed over some details about reading data from the standard input. Who decides how input from the terminal should be labeled? Intuitively, all data read from the terminal should be labeled with the principal corresponding to the user who originally executed the application. In other words, the data that the user enters should be protected according to her security policy.

Jif implements this by first adding a special native principal as an input parameter to `main` (the method called when the application is executed). It then requires that standard input be labeled at least as confidential as this principal:

```
static void main(principal user, String[] args) {
  jif.runtime.Runtime[user] runtime = null;
  try {
    runtime = jif.runtime.Runtime[user].getRuntime();
  } catch (SecurityException e) {}
  InputStreamReader[{user:}] inS = null;
  try {
    inS = new InputStreamReader[{user:}](
          runtime.stdin(new label{user:}));
  } catch (SecurityException ex) {
  } catch (NullPointerException e) {}
  ...
}
```

**Problem 3** Jif only provides a single native principal corresponding to the user executing the program. Furthermore, access to operating system resources belonging to the user, such as standard I/O and local files are provided, by Jif, but must be labeled with this native principal. The JPMail policy uses an entirely distinct set of principals (e.g. Alice may send email to someone who has no user account on her system), but Jif does not allow for any way to equate native principals to user-defined principals.

**Observation 3** In order to give the native, user principal an identity in our email system, we need to identify it with one of the principals in the principal store.

**Solution 3** This required modifications to the Jif runtime system to allow native principals to establish delegations. At the same time, we had to add some form of authentication to ensure that a malicious principal could not simply log in as `bob` and read Bob's emails. Thus, we require the user to provide, as authentication, Bob's certified private key (see Section 5.1). Note, that a more general form of compliance between an entire operating system information flow policy and an application-level policy is an interesting problem, left to future work [13].

# 4 JPmail architecture

We now give a description of the process of sending and receiving an email in JPmail. In this description, we focus on the information flows that are necessary for sending an email from one principal to another. In both the sending and receiving processes, the data must pass through software filters (points of processing that may audit or modify data) that serve to relabel and/or modify it. In sending email, there are two filters involved; in retrieving it, there is only one (strictly speaking, this one may not be necessary because the information is being upgraded). The only requirements on these filters is that they are authorized by the owner of the data and that they produce the properly labeled output.

The following example refers to the numbered Figure 2 in which a principal Alice uses JPmail to securely send an email to another principal Bob, who in turn reads that mail.

**Sending email** Alice initializes a `MailSender` with a policy and her principal name (`alice` in this case—the policy and principals are explained in more detail in Section 5.1) as well the necessary parameters for the outgoing mail server (address, user name, etc.). **1**) Then Alice enters an email, including the header information and the text for the body of the email. This email is labeled as `alice` since it came from an input stream owned by Alice. **2**) The email must then undergo two transformations. First, in order to send out an email, the headers must be readable by the mail server. This requires that they be declassified to public. Secondly, the body must be readable by the recipient, Bob, without being readable by the public. These two steps are performed by a reclassifier, as shown. At this point, the email headers are visible to the server while the body is visible only to the recipient. **3**) The next step is to make the entire email visible to the server so that it can be sent out. At the same time, we must not compromise the policy on the body, which requires that it should only be visible to `bob`. To do this, we use a random one-time symmetric key approach. The one-time key ($k$) is generated, used to encrypt the email body ($b$), and encrypted with `bob`'s public key ($k^+_{bob}$). Then the original body is replaced with the encrypted body along with the encrypted, one-time key, i.e. the message body contains $E(k, b), E(k^+_{bob}, k)$. The encrypted values can be declassified to be visible by the server without compromising `bob`'s privacy. **4**) Finally, the email is sent to the SMTP server, which in turn delivers it to the POP3 server.

**Reading email** Bob retrieves his email from a POP3 server using the `MailReader` class. **5**) After connecting to the server, the mail reader takes in each email and examines the label field in the header (`Label` in the figure). The header information can remain public, but the text of the body must be decrypted and reclassified according to the label field. **6**) To do this, we require Bob's private key. Since Bob has access to his own private key, it can be read in from the file system, labeled as `bob`. If another user were trying to impersonate Bob, the private key would not be available and the attempted decryption would fail. **7**) Since

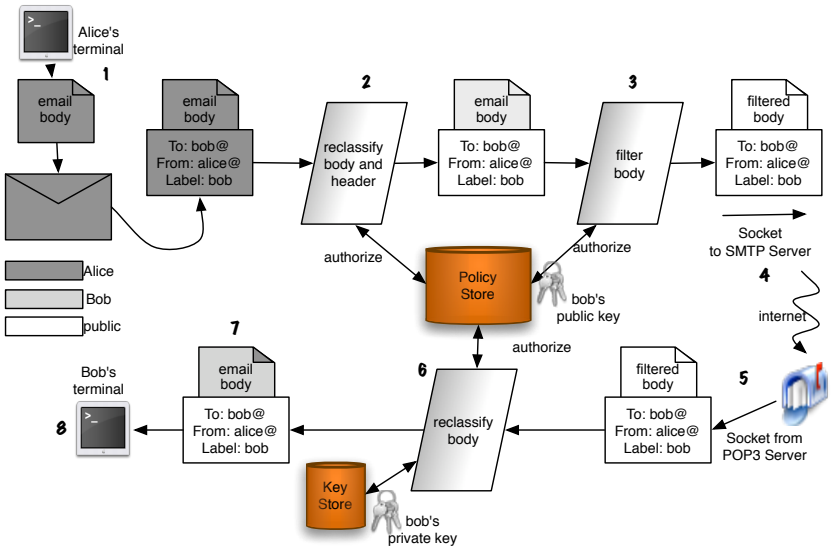


**Figure 2. Sending and retrieving a message using the JPmail client.**

Bob's private key is labeled as `bob`, decrypting the body of the email automatically raises the plaintext's security level to `bob`. Now that the body is safely in the confines of the Jif sandbox, it can be decrypted without fear that it will be leaked. **8**) Finally, since the user who is running the mail reader is `bob`, this email can be printed to `bob`'s terminal.

## 5 Tools

Our experience in JPmail has highlighted the need for additional policy and application development tools. Although we have focused our attention on Jif, these tools would be necessary in any security-typed language. The following describes the tools we developed to address areas we found most challenging and/or time-consuming: high-level, policy development in a distributed system and software engineering.

### 5.1 Policy tools

Jif lacks a policy management infrastructure. We developed such an infrastructure consisting of two components: a runtime policy store that provides dynamic access to principals, and a policy compiler that automatically generates principals and initializes the policy store. We describe the policy language and compiler more fully and more formally in a recent, prior work [12], including a proof of correctness and the security property they maintain. In this work, we have made this policy infrastructure more robust and useful in a distributed system through binding principals to public/private key pairs. We briefly consider these tools below and describe the new extension for cryptographic principals and give some example usage in the context of JPmail.

| | | |
|---|---|---|
| principal | $p ::=$ | `alice` \| `bob` \| ... |
| declassifier | $D ::=$ | `method1` \| `method2` \| ... |
| delegation | $Del ::=$ | $p$ `->` $p$ |
| trust stmt | $Allow ::=$ | $p$ `allows` $D(p)$ \| $p$ `allows None` |
| policy stmts | $Stmt ::=$ | $(Del \mid Allow)*$ |

**Figure 3. Policy language syntax.**

**Policy language and compiler** Jif lacks high-level policy specification tools. We desire to automatically generate Jif policy infrastructure (code) from high level specifications. To this end, we developed a policy language and an accompanying compiler. Currently, the policy language consists of only two types of policy statements: delegations and declassifier-allowances (implicitly, there is also the declaration of the principals themselves). The syntax is given formally in Figure 3. Recalling **Observation 2** from Section 3.2, we hold that this simple policy is complete for capturing the global meaning of local policy labels.

Illustrated in Figure 4, policy is integrated into a Jif program by using the policy compiler. The compiler interprets the policy specification, given in a separate file, and generates the associated Jif code. The created code provides functions for the creation of principals (as identified in the specification) and for their insertion in the policy store. The policy specification also includes explicit authorizations of all the declassifiers that the principals trust. Functions for the policy initialization and principal delegations described in the policy file are also created. Finally, this automatically generated code is introduced into the Jif application with a single line of Jif code.

Jif provides a Principal interface which allows for policy to be implemented directly in dynamic principal objects. In

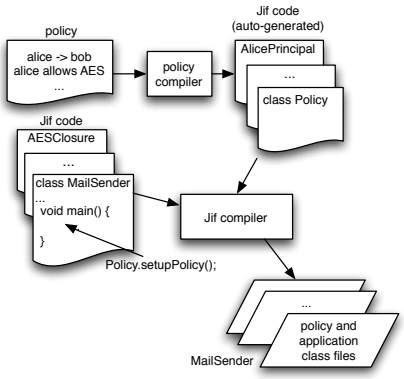


**Figure 4. The policy compiler automatically generates Jif policy objects from user specifications.**

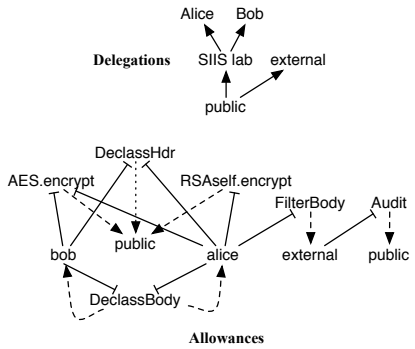


**Figure 5. Delegation hierarchy and declassification allowances for a sample, research lab security policy.**

particular, Jif Principals maintain a list of principals they delegate to and they also allow the programmer to implement a method which is called to authorize a declassification. The Principal interface can also be implemented with additional member data, allowing us to push public keys (and if available, private keys) directly into dynamic principals. Our policy compiler automatically generates a Principal implementation for each principal given in the policy file. It automatically generates the authorization method based on the `allow`-statements for the permissible declassifications associated with the principal. It also generates the `JifPolicy` class which instantiates and initializes each principal, establishes its delegations (according to the policy file) and loads its public and, if available, private key. Finally, it creates a policy store in which it stores all the newly created principals (they can be looked up by name and used in policy labels) and it returns this policy store to the calling application. To use a policy, an application must merely retrieve a policy store by calling `JifPolicy.setupPolicy`.

As an example, a security research lab could design a policy in which all of the members are listed in the policy and their public keys are certified by a lab's certificate authority. Emails can be sent freely throughout the lab. Emails destined for recipients outside the lab are handled by a separate filter that imposes the lab's policy on external mail (whether it be adding a disclaimer, limiting the number of outgoing messages, auditing outgoing messages, etc.). In Figure 5, we illustrate this lab policy. Principals begin with lower-case letters and declassifiers with uppercase (`public` is repeated only for clarity of reading.) The solid arrows indicate delegations, the "T" arrows indicate allowances and the dashed arrows indicate the lowest level a filter may declassify to. Note that this lab policy was used for the development testing of JPmail.

**Cryptographic principals** Cryptography provides two central functions within JPmail[4]: it is critical for ensuring data is not leaked as it passes outside of a Jif application and it plays an essential role in maintaining the consistency and integrity of principals from one Jif application to another. This former function is achieved via encryption of email bodies. In the latter function, principals are uniquely identified via an association with a public key (certificate). We leveraged existing facilities for creating and verifying X.509 certificates for this purpose. For certificate signing and verification, we created a JPmail-specific certificate authority (CA).

Our use of certificates required us to bind public keys to Jif's principals. In Jif, principals are created by implementing a `Principal` interface. We created our own `KeyPrincipal` by implementing the standard `Principal` interface (which requires a name, closure authorization testing, equivalence testing and delegation testing) and also adding fields for a public and, if it's available to the current user, a private key. Before allowing a public key to be associated with a principal, the certificate containing the public key is validated using the public key of the trusted CA. For sending email to users outside the system, the `external` principal can be used which can be declassified with the `FilterBody` declassifier (this could be used to audit emails or add a disclaimer, e.g.). In order to add a new principal to the system, the principal's public key certificate must be distributed and a delegation should be added by each principal to their policy to include the new user. Because the policy is decentralized, it could be stored on a common server and each user could update his own policy, while possibly also

[4]JPmail uses the following algorithms: DES, TripleDES, AES were used in CBC mode for all symmetric key operations, and RSA Electronic Codebook (ECB) mode with PKCS1 padding for asymmetric key operations. We also used MD5 hashes on passwords for authentication with the email servers according to the POP3 and SMTP protocols.

maintaining some mandatory requirements. These are areas of future investigation.

In order to integrate the delegations and authorizations defined in our policy with the local file system, we had to augment the Jif compiler's runtime environment. By introducing a method to delegate from a `NativePrincipal` to a non-native principal, we make an association between the user running the program and one of our internal principals. In order to authorize this association, we require that the user can only delegate to a principal for whom the user can provide the private key. In other words, JPmail authenticates the claimed principal by checking that it has a signed certificate from the CA with the principal's identity and the associated private key before it allows the user to assume that identity. It does this by loading the private key from the user's key store, for which it requires the user's password and checking it against the JPMail-specific CA.

### 5.2 Practical tools for software engineering

**An integrated development environment (IDE)** To partially address the limitations of the current development environment, we developed an IDE for Jif. This tool integrates the Jif compiler with the open-source Eclipse[5] integrated development platform. Eclipse has been used to build IDEs for a wide variety of languages such as C and Java. This IDE, even in its most basic form, represented 80-100 manhours of work. For brevity, we discuss only the impact of the IDE.

The main benefit of the Jif IDE (so far) is that it considerably reduces the edit/compile/repair development cycle. As Eclipse runs the Jif compiler in the background, it quickly tags syntax errors, missing exceptions, missing import statements and even security label violations. Thus, errors, especially vexing book-keeping errors, can be quickly fixed without having to switch to a terminal window and recompile the source. Furthermore, the integrated support for syntax-highlighting and easy integration with version control tools such as CVS and subversion were welcome additions to the Jif programming environment. Future development in the Jifclipse feature will include more complex refactoring algorithms, greater contextual awareness of labels, integration with our policy tools, and a host of runtime and development utilities.

**Declassifiers** Declassifiers play a key role in providing security guarantees. Because all realistic security-typed applications need to have some way of declassifying information (since even functions like encryption and password checks leak small amounts of information), it is valuable to build up a collection of commonly used declassifiers. For example, encryption and auditing functions can declassify data–they expose some amount of data, but (under certain circumstances) it is not enough to be deemed a leak. Such functions are similar to seal classes [1, 22], which provide a declassifying filter that limits *when* information is released. Libraries of common declassifiers can be carefully engineered and formally verified to prevent unacceptable or dangerous leakage (as opposed to the vanishingly small leak from encryption or the acceptable leak from an audit or password check). Applications can benefit from this vetting, and avoid declassifying through potentially dangerous or untested interfaces [23].

We have built a library of declassifiers for use in Jif applications. The declassifiers we constructed for encryption have widespread value and could be re-used without modification in other applications. Some of the declassifiers we created are special-purpose, because they are made to handle only email objects. Even these, however, are useful as blueprints for other application developers. Because of its extensive use of declassification, JPmail required the exploration of new features available in Jif 2.0: `Closures` and the `Principal` interface. Our code serves as the first explorations into the utility and flexibility of these features[6].

## 6 Evaluation

Based on our experience of implementing an email client, we now evaluate the use of Jif to build the JPmail application and suggest improvements to aid future developers. Some preliminary timing measurements indicate that the overhead of Jif, especially for an I/O-bound application like email, is not significant [13]. The greatest slow-down is caused by the encryption, which could possibly be improved by entrusting the encryption to the operating system, using, for example, IPsec. In this section, however, we are primarily concerned with evaluating the security and usability of Jif.

Jif provides a strong basis for assessing the correctness of an security policy implementation. Recall our security policy given in Section 2.1: "*The body of an email should be visible only to the authorized senders and receivers.*" We can be sure that this policy has been correctly implemented by examining the label on emails and cross-checking with the policy file about what information flows are possible with that label. In particular, by examining the `smtp.MailSenderCrypto` class and the `readMessage` method, we find that an email is read in from an input stream which has the user's label (the user who ran the mail client) and consequently is also labeled with the user's label. Furthermore, the user provides the name of the principal `rcpt` to whom he wishes to send the email. That principal is looked up in the principal store and associated with a Jif principal (which was created from the policy file). The body of the email is then passed through a declassifier, `DeclassMsgBody` which will relabel the body to `{rcpt:}` if

[5] `http://www.eclipse.org/`

[6] Our code is made available at `http://siis.cse.psu.edu/jpmail.html`.

rcpt is one of the allowed recipient principals, given in the policy file. From this point, the rcpt policy governs all information flows. Namely, before this email can be placed on the public socket, it must pass through another declassifier based on rcpt's policy. If rcpt allows for AES encryption, then the email is encrypted as described in Section 4, using a one-time randomly generated key (which cannot be leaked, because it is labeled rcpt) and the principal's public key.

We can repeat this evaluation for other sensitive data such as keys and passwords. For these items the analysis is even simpler, because they are not dynamically labeled like emails (which depend on user input). The password is given a label when a MailSenderCrypto object is created. Checking the policy file, we can see that Strings cannot be declassified except through an MD5 filter. Creating an MD5 hash is necessary for authentication with the mail server. This was made clear when we tried to send the password as plaintext over the mail server socket when establishing a connection. This insecure practice was automatically disallowed by Jif. At this point, we face a limitation in Jif's security analysis. Namely, the SMTP and POP3 protocols' password authentication ensures that a nonce is used to prevent replay attacks. This is not encoded in the Jif labels in any way. Merely knowing the declassifier that is used is not enough to ensure that replay attacks are avoided—only that the plaintext has not been leaked. We have to trust in the protocols for protection against more subtle attacks (a non-trivial assumption).

There are other caveats to security that must not be overlooked. Firstly, the security properties of a program are dependent on the correctness of the Jif compiler (and our policy compiler). Secondly, the security properties may also be dependent on supporting infrastructures. This includes the correctness of encryption libraries and the strength of used cryptographic algorithms, the protection on keystores and correctness of public-key cryptographic libraries as well as the security enforced by the local file system. Moreover, for the system to be secure, the enforced policy must be consistent across all clients. We defer integrity to future work.

One advantage of Jif is that it forces the programmer to think in terms of information flows and to consider security concerns from the outset. Interestingly, there is a strong consensus in the software engineering community that performing these kinds of security analysis at design time is essential to the security of the resulting system [6].

Finally, we observed that the policy tool effectively decoupled policy from the programs that they govern. This allowed us to modify policy easily in order to accommodate different security models. By instrumenting the code during development with different options for each filter, we could implement distinct security models without altering the code. Furthermore, by gathering the policy into a single file, it was easier to do a security analysis and gauge what information flows could take place for a given principal, in contrast to leaving declassify statements in the code.

**The difficulty of programming in Jif** The shortcomings of Jif are frequently not specific to Jif, so much as they are issues that any security-typed compiler must face. Jif is the most advanced security-typed compiler available and the Jif team should be commended for their substantial efforts, but it is not yet ready for industrial development. Implementing the JPmail client took hundreds of man-hours (not including the time necessary to learn Jif) to generate around 6,000 lines of code. Furthermore, despite the substantial amount of work involved, our mail client is neither flashy nor full-featured. It uses text-based I/O and handles a minimal subset of the MIME standard just enough to allow communication between various principals. This should be contrasted with the more modest efforts needed to retrofit composable security properties onto the full-featured, GUI-based email client, Pooka, by using the Polymer security policy system [2].

### 6.1 Needs for improvement

**Debugging** A serious issue we faced was in the difficulty of debugging Jif applications. No advanced debugging tools exist for Jif, leaving us only with rudimentary print statements. Furthermore, because Jif is designed to hide information, we, in turn, had to *overcome* its propensity for hiding information in order to reveal it for debugging.

**Implicit flows** Another source of repeated consternation in Jif programs involves implicit flows. Jif tracks not only explicit flows of information as data passes from one variable to another, but also implicit flows, in which data is leaked through the control path. For example, making a low confidentiality assignment in the body of a conditional with a highly confidential guard releases a small amount of information about the guard through the assignment. To prevent this, Jif raises the security level of everything in the body of the conditional to the level of the guard (by assigning this security level to a pc-label and label-joining the pc with every information source in the body) for the length of the body. This applies also in the presence of loops and exceptions. With nested conditionals, loops and exceptions, it can become quite difficult to determine by inspection what the security level of the pc is at a given point in the code. Furthermore, even in knowing the pc-label, it can be challenging to determine how it got that way.

**Reader lists** One unexplored area of Jif that holds great promise in expressing flexible policy is the use of Jif reader lists. The decentralized label model (DLM) provides for labels such as {alice:bob}, meaning that Bob cannot copy the value, but only view it on a channel that both Alice and Bob trust. Using this policy, an email could be sent to Bob which he could not save on his hard drive or forward to an-

other user, but could only view on a terminal certified by both him and Alice. This would be a very useful policy, allowing Alice to retain control over her data, even on another user's machine.

**Distributed policy** One of the great boasts of security-typed languages is that noninterfering components are composable. Thus, secure programs can be built in a modular way, a block at a time, and when composed can still make end-to-end security guarantees. The problem is that in order to have security-typed distributed applications, all necessary support structures, including sockets, network stacks, file systems, operating systems, etc. would need to be built in security-typed languages. Since this dream will not be fulfilled in the near future, Jif at least allows incremental deployment of applications with interfaces to existing support services. Thus, in the mean time, conventional security enforcement mechanisms such as certificates, encryption, key stores, SSL, etc. must be used. One shortcoming of Jif is that it does not provide a secure integration of these conventional security enforcement mechanisms.

**Incremental development** Future applications will require many more support libraries be integrated with Jif. Currently, Jif has a small library of security-typed versions of Java's `Container` classes. Most other support must still be developed however. In our case study, we have provided a subset of the `javax.mail` library. Hopefully other projects will continue to fill the gaps, and thus make Jif more accessible to application developers.

## 7 Related Work

The concept of information-flow control is well established. After the first formulation by Bell and La Padula [3] and the subsequent definition of noninterference [11], Smith, Volpano, and Irvine first recast the question of information flow into a static type judgment for a simple imperative language [28].

The notion of information flow has been extended to languages with many other features, such as programs with multiple threads of execution [27, 14], functional languages and their extensions [19, 29] and distributed systems [14]. For a comprehensive survey of the field, see the survey by Sabelfeld and Myers [21].

Two robust security-typed languages have been implemented that statically enforce noninterference. Flow Caml [24] implements a security-typed version of the Caml language that satisfies noninterference. JFlow [16] and its successor Jif [18] introduce such features as a decentralized label model and run-time principals in an extension to the Java language. Jif is actively in development, with the latest release in June 2006 (v. 3.0) introducing integrity labels [5].

A central theme in this paper is declassification. Sabelfeld and Sands provide a survey of this field [22]. We add our own, modest work to this collection [12], introducing the notions of trusted declassification and noninterference modulo trusted methods. What sets our work apart is a demonstration of the practical utility of these tools.

Some of our work uses concepts (PKIs, email encryption, etc.) explored more extensively in such systems as OpenPGP. Our purpose in developing the JPmail client, however, was not to replace the state-of-the-art secure mail clients (for a survey, see [10]), nor to replace extensive secure email infrastructures such as OpenPGP [4], but rather to investigate the interaction of security-typed programming with real-world security tools, such as certificates, symmetric and asymmetric encryption, etc.

The most closely related work is the paper by Askarov and Sabelfeld, detailing a mental poker protocol application in Jif. Our work is a natural successor to theirs, exploring areas of policy and distributed policy that were not developed in their work. Our application is also slightly larger than theirs (about 15%) and an order of magnitude larger than the other security-typed applications [18, 26].

## 8 Conclusions

This paper has described the first real-world application built in a type-secure language: the JPmail email client. In so doing, we have exposed the advantages and limitations of the state of the art. On the positive side, Jif provides extensive and usable interfaces for developing information flow-governed applications. These features provide a basis from which concrete security guarantees can be built.

Our work in JPmail also uncovered two central deficiencies. First, at present, working with Jif is exceedingly difficult. This is because of a dearth of developer tools and the difficulty in determining the source and meaning of errors. We introduce an IDE and policy design patterns to help address these developer tool limitations. Second, there is little or no infrastructure to formulate policy, communicate that policy beyond a single application, or map the guarantees onto surrounding security infrastructure. Here, our policy compiler and policy store address each of these areas.

Even in the face of the considerable challenges we encountered in this project, we are heartened by the experience. To be sure, the tools and practice of using Jif, and in a larger sense security-typed languages, must mature before their promise is met. We see this work as another step in that maturation.

## 9 Acknowledgements

We thank Steve Chong for his endless patience with and prompt responses to our questions about Jif. We thank the reviewers for their helpful comments. This research was supported in part by NSF grant CCF-0524132, "Flexible, Decentralized Information-flow Control for Dynamic Environments" and in part by Motorola through the Software Engineering Research Consortium (SERC).

## References


[1] A. Askarov and A. Sabelfeld. Secure implementation of cryptographic protocols: A case study of mutual distrust. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS '05)*, LNCS, Milan, Italy, September 2005. Springer-Verlag.

[2] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM Press.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[4] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. Openpgp message format. IETF RFC 2440, November 1998.

[5] S. Chong and A. C. Myers. Decentralized robustness. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*, Venice, Italy, July 2006. to appear.

[6] P. T. Devanbu and S. Stubblebine. Software engineering for security: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 227–239, New York, NY, USA, 2000. ACM Press.

[7] Federal Trade Commission. Eli Lilly settles FTC charges concerning security breach. FTC Press Release, January 18 2002. `http://www.ftc.gov/opa/2002/01/elililly.htm`.

[8] Federal Trade Commission. Cardsystems Solutions settles FTC charges. FTC Press Release, February 23 2006. `http://www.ftc.gov/opa/2006/02/cardsystems_r.htm`.

[9] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 4(3):224–274, 2001.

[10] S. L. Garfinkel, E. Nordlander, R. C. Miller, D. Margrave, and J. I. Schiller. How to make secure email easier to use. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, Portland, Oregon, April 2005. SIGCHI.

[11] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[12] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: High-level policy for a security-typed language. In *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '06)*, Ottawa, Canada, June 10 2006. ACM Press.

[13] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. Breaking Down the Walls of Mutual Distrust: Security-typed Email Using Labeled IPsec. Technical Report NAS-TR-0049-2006, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Sept. 2006.

[14] H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *J. Computer Security*, 11(4):615–676, 2003.

[15] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, January 1999.

[16] A. C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachussetts Institute of Technology, University of Cambridge, January 1999. Ph.D. thesis.

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

[18] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[19] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL*, pages 319–330, January 2002.

[20] B. Ross and R. Esposito. Espionage case breaches the white house. ABC news report, Oct. 5 2005. `http://abcnews.go.com/WNT/story?id=1187030`.

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

[22] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of the IEEE Computer Security Foundations Workshop*, Aix-en-Provence, France, June 2005.

[23] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *NDSS*. The Internet Society, 2006.

[24] V. Simonet. FlowCaml in a nutshell. In G. Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, March 2003.

[25] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *CSFW '06: Proceedings of the 19th IEEE Computer Security Foundations Workshop*, 2006.

[26] S. Tse and G. Washburn. Cryptographic programming in jif. Project report for CIS-670, Spring 2003, Feb. 11 2004. `http://www.cis.upenn.edu/~stse/bank/main.pdf`.

[27] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *JCS*, 7(2–3):231–253, November 1999.

[28] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *JCS*, 4(3):167–187, 1996.

[29] S. Zdancewic. A Type System for Robust Declassification. In *Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics*. Electronic Notes in Theoretical Computer Science, March 2003.