Cryptographic Programming in Jif

Stephen Tse Geoffrey Washburn
University of Pennsylvania

Abstract

We propose developing a library of cryptographic primitives that uses information flow policies to enforce additional security properties on encrypted data and data obtained through decryption. The goal of the library is to support communications via shared or public keys, restrict information leaks, impose the dependency between keys and encrypted values, and integrate authentication information into the types. We evaluate the usefulness of such library by implementing simulation of the communication between a bank server, an ATM machine, a bank card, a bank database, and an auditing agent. This example not only makes use of a hierarchy of principals and declassification in the decentralized label model, but also aims to show how tracking information flow can increase security confidence in practical cryptographic programming.

1 Introduction

Confidentiality is a major challenge for distributed systems where communication between components as well as components themselves cannot be trusted. Encryption is a standard mechanism to secure communication over public channels. Cryptographic programming refers to the development of software with encryption and decryption of messages as well as key management. There are many efforts to provide a standard library for cryptographic programming. However, their goal is invariably the security of the communication channel but fail to address the security of data manipulated by the components themselves. For example, access control can be used to control the release of confidential data but cannot prevent users from accidentally (or prevent malicious code from intentionally) sending out confidential data such as the private keys of the users, or data before it has been encrypted.

An information flow policy is a security policy that controls the propagation of secure data [6]. Language-based security employs type system and program analysis to enforce security policies at compile time. By contrast, run-time

[1]Stephen Tse (stse@cis.upenn.edu) and Geoffrey Washburn (geoffw@cis.upenn.edu)
Project report for CIS-670, 2003 Spring. Last update: February 11, 2004.

mechanisms such as reference monitoring are more expensive and are only capable of enforcing safety properties [21]. Language-based security can also minimize the trusted computing base so that programmers and code need not be trusted to enforce the security policies - only the compiler must be trusted, and with certified binaries only the runtime system needs be trusted. However, traditional systems require a centralized host to specify policies and impose a rigid lattice model on security levels, causing policies to be too restrictive in large-scale applications.

The decentralized model for information flow control solves the dilemma by having decentralized labels and allowing a trusted principal to explicitly declassify his own secure data [14]. There is no need for a universally trusted host and each principal can independently verify that his policies are enforced. Jif is a language based on Java that implements the decentralized model [17]. It implements a significant subset of Java features such as inheritance and exceptions in order to express realistic programs.

In this paper, we design and specify a security policy for cryptographic programming in Jif. By doing this, we restrict declassification to secure situations that are guaranteed by the underlying cryptography system. Our contributions include

1. The design of a cryptographic library: The library supports communications with shared or public keys, restricts information leaks, imposes the dependency between keys and encrypted values, and integrates authentication information in types. The primitives are key generation, encryption and decryption, and signing and verification. Users can encode security policies in the language Jif so that the type system will statically check the invariants.

2. The implementation of a realistic simulation of network application: a banking example with communications between a bank server, a ATM machine, a bank card, a bank database, and an auditing agent. The communication protocol handles encrypted and signed messages for authentication and transaction between the bank server and the ATM machine. Declassification is used to control the information flow of the client's private key from the bank card to the ATM, and the flow of the transaction log from the bank database to the audit agent. It shows how to increase confidence in security for practical cryptographic programming.

3. The application of information flow control in Jif: The example not only makes use of important features such as hierarchy of principals and declassification in the decentralized model, it is also the largest program written in Jif to date.

The rest of the paper is organized as follows. Section 2 motivates with the banking example. Section 3 introduces the cryptographic library. Section 4 enumerates information flow policies and proves the main theorem of program security. Finally, Section 5 and Section 6 conclude with the related work and a discussion of the Jif system.

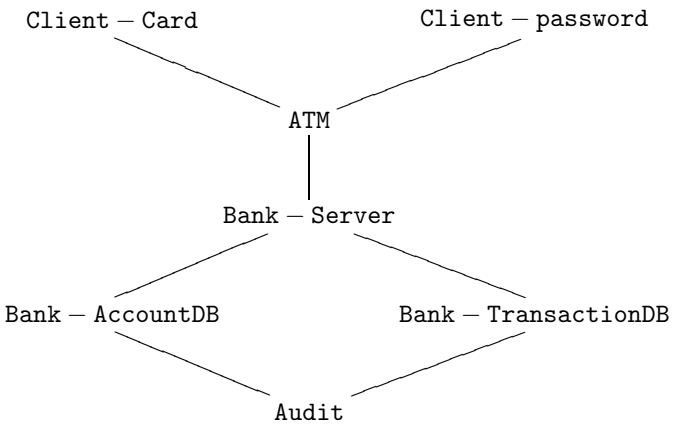

Figure 1: Principal Interactions

2 The Jif Language

In order to motivate the use of information flow policies in a cryptographic library, we will use a running example of a banking system. This example comprises enough distinct principals to make the interactions between them interesting, and the importance of confidentiality in the communication protocol is obvious.

2.1 Principals

Figure 1 diagrams the interactions between the principals in our model of banking system.

- **Bank**: An institution that provides the `Bank-Server` to take requests from an `ATM`, updates the balances in `Bank-AccountDB` for `Client`, and records transactions with the signatures of `Client` in `Bank-TransactDB` for examination by `Audit`.
- **Client**: A customer who has an account with the `Bank` in `Bank-AccountDB`, possesses a `Client-Card` with his private-key protected by `Client-Password`, and interacts with `ATM` for financial transactions.
- **ATM**: A machine that interacts with a `Client`, and sends transaction requests (deposit, withdraw or balance query) to the `Bank`.
- **Audit**: An agent that verifies the financial transactions recorded by the `Bank` against the signatures of `Client`s, and tallies the asset of the `Bank`.

2.2 Protocols

We now describe the complete four-stage protocol used to communicate between an `ATM` and the `Bank` over a public network. For simplicity, each financial transaction a customer invokes from the `ATM` requires an entirely separate session. It should be straightforward to optimize communication between an `ATM` and the `Bank` by allowing multiple transactions per session.

1. **Authentication Request** (`ATM` to `Bank-Server`): obtain the `account-id` of a `Client` from her `Client-Card`. Generate a nonce, `atm-nonce`, and lastly encrypt the message with `bank-pubkey` (the public key of `Bank`):

```
bank-pubkey{ account-id, atm-nonce }
```

2. **Authentication Response** (`Bank-Server` to `ATM`): decrypt the authentication request with `bank-prikey` (the private key of `Bank`) to obtain the `account-id` and `atm-nonce`. Obtain `client-pubkey` (the public key of the `Client`) from `Bank-AccountDB` with `account-id`, generate a `session-key`, generate `bank-nonce`, and lastly encrypt the message with `client-pubkey`:

```
client-pubkey{ session-key,
               atm-nonce,
               bank-nonce }
```

3. **Transaction Request** (ATM to Bank-Server): obtain `client-prikey` (the private key of the `Client`) from her `Client-Card` with `Client-Password`. Decrypt the authentication response with `client-prikey` to obtain `session-key`, `atm-nonce` and `bank-nonce`, verify that `atm-nonce` matches, generate a new `atm-nonce`, pack the `transact` field with the `account-id`, `action` type (deposit, withdraw or balance query), `amount` and the new `atm-nonce`. Sign `transact` with `client-prikey` to obtain `signature`, and lastly encrypt the message with `session-key`:

```
transact = < account-id,
             action,
             amount,
             atm-nonce >
signature = client-prikey{ transact }
session-key{ transact,
             signature,
             atm-nonce,
             bank-nonce }
```

4. **Transaction Response** (`Bank-Server` to `ATM`): decrypt the transaction request with the `session-key` to obtain `transact`, `signature`, `atm-nonce` and `bank-nonce`. Verify the `bank-nonce`, update `Bank-AccountDB` with

the `transact` to obtain the response `action` (transaction completed, insufficient funds, or balance). Store the `transact` and the `signature` to `Bank-TransactDB`, generate a new `bank-nonce`, and lastly encrypt the message with `session-key`:

```
session-key{ action, atm-nonce, bank-nonce }
```

3 Cryptographic Library

In order to implement the protocol described in the previous section, we need the following cryptographic primitives. Their interfaces are described in terms of Java syntax.

- **Key Generation Primitives**:

  ```
  Key aes_key();
  KeyPair rsa_key();
  ```

  A `KeyPair` is composed of a `PublicKey` and a `PrivateKey` object.

- **Encryption and Decryption Primitives**:

  ```
  Object aes_encrypt(Key key, Object o);
  Object aes_decrypt(Key key, Object o);
  Object rsa_encrypt(PublicKey key, Object o);
  Object rsa_decrypt(PrivateKey key, Object o);
  ```

- **Signing and Verification Primitives**:

  ```
  byte[] sign(PrivateKey key, Object o);
  boolean verify(PublicKey key,
    Object o, byte[] sig);
  ```

4 Information Flow

Having established the protocol and cryptographic primitives used as part of our banking simulation, we are now ready to choose a set of information flow policies to ensure a practical level of confidentiality in the system. We first review our assumption about the security model, then enumerate information flow policies as lemmas, and finally conclude by stating as a theorem that our chosen policies are sound and complete.

4.1 Security Assumptions

We make the following assumptions about the security of our banking system:

- The network is public and cannot be trusted.
- Standard cryptosystems are trusted to be unbreakable within the desired tolerances.
- Clients can be trusted to protect their passwords.
- Statistical data is trusted to not reveal confidential information.

4.2 Information Flow Policies

Lemma 1 (Policy: Network). *Do not trust the network.*

Proof. We restrict all network communication to use the following interface:

```
interface Net {
  void send (Object{} o);
  Object{} recv ();
}
```

Here we label the input parameter of the `send` method as bottom. This label ensures that any secure data, such as
`PrivateKey{bank:} bank_prikey`, will not be accidently leaked by sending it over the network. □

Lemma 2 (Policy: RSA). *The system's use of the RSA public-key cryptosystem is secure.*

Proof. RSA public-key cryptosystem is secure in our security model and the Jif labels in the following code enforce the proper use of declassification.

```
class PubKey authority(atm) {
  java.security.PublicKey k;
  principal p;
  Object{} encrypt (Object{atm:bank} o)
    where authority(atm) {
      if (p != bank)
        throw new Error("Wrong principal");
      Object{atm:bank} o2 =
        Crypto.rsa_encrypt(k, o);
      return declassify(o2, {});
  }
}
```

Here we label the object parameter as `atm:bank`, so that only data intended for Bank is encrypted with the public key of `Bank`. We enforce that public key's principal matches by use of a run-time equality check. The `authority` clauses in the function `encrypt` declaration and in the class `PubKey` declaration are used to ensure that the authority of the data's owner, `ATM`, has as `ATM` been granted before the ownership over the encrypted data can be discarded. □

Lemma 3 (Policy: AES + authentication). *The system's use of the AES symmetric-key cryptosystem once, authentication has occurred, is secure.*

Proof. The use of AES symmetry-key cryptosystem following authentication is secure in our security model, and the assigned labels in the following code enforce the proper use of declassification:

```
class SymKey authority(atm) {
  java.security.Key k;
  principal p;
  Nonce n;
  boolean auth;
  SymKey (Nonce{bank:atm} n, ...) {
    this.n = n;
    ...
  }
  void authenticate (Nonce{atm:bank} n) {
    auth = n.equal(this.n);
  }
  Object{} encrypt (Object{atm:bank} o)
    where authority(atm) {
    if (p != bank)
      throw new Error("Wrong principal");
    if (!auth)
      throw new Error("Not authenticated");
    Object{atm:bank} o2 =
      Crypto.aes_encrypt(k, o);
    return declassify(o2, {});
  }
}
```

Here the use of declassification and authority clauses are similar to those in used in Lemma 2. The primary difference is the constraint that the key may only be used after authentication has occurred. As explained in Section 2.2, the protocol authenticates by checking the equality of the nonce generated by `ATM` and the nonce decrypted by and sent from `Bank`. The boolean variable `auth` is used to control the state of authentication such that the function throws an exception at runtime if an attempt to use the key is made before authentication has occurred.

When `Bank` constructs a new symmetric key for use as the session key in the protocol, it passes the nonce it decrypts from the authentication request.

When `ATM` receives the authentication reply, it passes its own nonce to enable the key in the function `authenticate`. Because the nonce in the constructor has the label `bank:atm`, only the nonce from `Bank` can be passed to it. Similarly, because the nonce in the function `authenticate` has the label `atm:bank`, only the nonce from `ATM` can be passed to it. This eliminates the possibility of the `ATM` mistakenly constructing a session key and authenticating the key with the same local nonce. □

Lemma 4 (Policy: password). *The client password remains confidential.*

Proof. The client password is secure in our security model and the labels in the following code enforce the proper use of declassification:

```
class Card authority(client) {
  PrivateKey{client:} k;
  String password;
  PrivateKey{atm:} get (String p)
    where authority(client) {
    if (p != password)
      throw new Error("Wrong password");
    return declassify(k, {atm:});
  }
}
```

Here we declassify the private key from `client:` to `atm:`. Although the policy is simple, we have made an important assumption about the security of the client password in our model. This external assumption cannot be quantified nor verified in our program, illustrating the fact that security is more than a mechanical process.

Moreover, we have made an interesting design choice here. The return label of the private key can be labeled as `client:atm`, but doing this requires the authority of `Client` again for declassifying encrypted objects when `ATM` sends messages to `Bank`. Labeling it as `atm:` instead, `Client` gives up the ownership of the private key and transfers the responsibility to `ATM`. There are other design alternatives such as acts-for relationship of `ATM` and `Client`, illustrating the fact the security allows interpretations of different precisions. It is also possible to consider a system where the `Client`'s smart card has the ability to perform the encryption, only requiring that the password be revealed to the `ATM` instead of both the password and her private-key. □

Lemma 5 (Policy: audit). *The leaking statistics data is secure.*

Proof. Leaking statistics information is acceptable in our security model and the labels in the following code enforce the proper use of declassification:

```
class Audit authority(bank) {
  long{audit:} audit () where authority(bank) {
    long{bank:} total = 0;
    for (int i=0; i<transact.length; i++) {
      int id = transact[i].id;
      PublicKey key = account[id].key;
      if (!Crypto.verify(key,
        transact[i], signature[i]))
        throw new Exception("Wrong signature");
      if (transact.deposit())
        total += transact[i].amount;
      else
        total -= transact[i].amount;
    }
    return declassify(total, {audit:});
  }
}
```

Here we declassify the total assent of `Bank` from `bank:` to `audit:`. Although the policy is simple, we have made an important assumption about the statistical properties of data leaked with respect to our model. This assumption, however, may be checked or monitored by, say, limiting the number of audits per year. The checking is done outside the code fragment and we have not considered incorporating this kind of constraints into our system.

Note also that, for demonstration purpose, we put the code of verifying signatures of transactions inside the same function with `Bank` authority. It shows that sometimes naturalness and performance lead programmers to intermingling secure code with an insecure fragment. It usually means a bad style of security programming and we should strive to confine secure code as much as possible. □

4.3 Confidentiality Preservation

Lemma 6 (Jif Soundness). *The Jif theory and compiler is sound: a program that type-checks with the Jif compiler preserves confidentiality, modulo the policies imposed by declassifications.*

Proof. See Myers' doctoral thesis [13] for proofs of the soundness of the Jif language. □

Theorem 7 (Program Security). *The chosen information flow labels preserve the desired confidentiality in the banking system simulation.*

Proof. Lemma 1 says the program does not trust the network, which satisfies our security model. The program contains only four `declassify` clauses (which can be verified by searching the source code), and each of them can be accounted for by the following lemmas:

1. Lemma 2 says the use of RSA public-key cryptosystem is secure.
2. Lemma 3 says the use of AES symmetry-key cryptosystem together with authentication is secure.
3. Lemma 4 says the client password is secure.
4. Lemma 5 says that leaking statistics information does not violate confidentiality.

The program also type-checks with the Jif compiler. Hence the program is secure by Lemma 6. □

5 Related Work

Past research that most directly relates to our research is the study static of information flow and data integrity analysis. Most information flow models are use a lattice model that originates from work by Bell and LaPadula [4] and Denning [6]. The earliest work on static information flow dates back to Denning and Denning [7].

Palsberg and Ørbæk examined trust and integrity with a simple extension of the λ-calculus [18]. Heintze and Riecke's formalized information-flow and integrity in a typed λ-calculus with references, the SLam calculus [10], and proved a number of soundness and noninterference results.

Volpano et al. [27] showed how to formulated Dennings work as type system and proved its soundness with respect to noninterference. Volpano and Smith later developed type inference system with for information flow with noninterference [25]. Smith and Volpano showed how to eliminate covert information flows resulting from exceptions and nontermination [24] and threads [22] in the context of an imperative language with two security classifications, "high" and "low".

Recognizing that strict noninterference is too strong for practical programming, Myers and Liskov build upon previous work in information flow with a decentralized declassification model [14, 15, 13, 16] that allows principals to leak information in a controlled fashion. Out of this research extensions to the Java [9] language and type system were developed, first in JFlow [12], and then Jif [17]. Zdancewic et al. in the context of secure transparent distributed computing [29] extended the decentralized label model of information flow with limited integrity constraints and an operator "endorse" that mirrors that of declassification. Research on extending Jif with full integrity constraints is in progress. A recent version of Jif served as the basis for our study.

Zdancewic and Myers [28] gave a precise characterization of the information leaks possible declassification. Volpano and Smith have also examined how to characterize the rate at information leaked as result of declassification in sequential and synchronous concurrent programs [26].

Less related, there is recent work on formal languages with cryptographic primitives. Sumii and Pierce [19, 23] and consider an extension to the λ-calculus

with cryptographic primitives, but the their focus was primarily to study the relationship between cryptography and parametricity.

Abadi and Gordon developed the spi-calculus [1, 2, 3], an extension of Milner's π-calculus [11] with cryptographic primitives, for studying cryptographic protocols. The spi-calculus has been used as part of the Cryptyc project by Gordon and Jeffrey [8] where a type and effect system for protocols has been developed to check for secrecy and authenticity violations. Secrecy violations are akin to covert information flows, and authenticity is similar to integrity. This work could conceivably formed an alternate foundation for our study, but they are more concerned with formal protocol development than practical programming, nor do they have a general purpose language at this time.

Our study makes use of the RSA [20] algorithm for public-key and AES (Rijndael) [5] for symmetric cryptography, but other algorithms would have worked equally well. Our work assumes that whatever algorithms are used, that they are capable of perfectly maintaining noninterference.

6 Evaluation of the Jif System

Developing and implementing the information flow policies we have described using the Jif system proved to be significantly more difficult than we initially anticipated. Firstly, while they are already recognized limitations to programming in Jif, it is worth mentioning that its lack of many of Java's features such as threads and inner classes make translating Java code to Jif challenging. Some features of the Java language, such as inner classes, are presently missing from Jif simply because the engineering time hasn't been devoted to them, while others such as threading will require advances in the design of the type system to ensure that covert and implicit flows do not occur.

We were able to address the lack of threads in Jif by converting the banking simulation code we started with to a single transaction queue implementation. This however, will not scale well in practice, because for large ATM systems it is very likely that the queue will become long, resulting in lengthy wait times for customers. This could be partially addressed by having several front-end servers, but as only one of these front-end servers will be able to communicate with the bank database and transaction logs at a time, the bottleneck has only been pushed around.

Translating existing Java code into Jif is complicated because it is very difficult to introduce label annotations in an incremental fashion. It is common that introducing a label in one location, will break unrelated code in non-obvious ways. The problem becomes even more difficult when faced with the rich design space that the owners-lattice and readers model allows. There are usually a great number of seemingly viable choices for annotations that will enforce the desired confidentiality policies. However, certain combinations of assignments to different subsystems may not interact as expected. It would be worthwhile to document a common set of information flow idioms ("patterns"?) that could be referred to in the design of larger systems. It might even be possible to design

a set of standard abstract base classes or interfaces implementing these idioms. It might also be worthwhile to consider the development of tools for visually explaining and editing information flows within a program.

Choosing annotations can become even more difficult when the novice Jif programmer becomes aware that static, dynamic, and parameter principals may not be used interchangeably in a number of contexts. It would be advisable to revise and extend the theory of principals to make the different types of principals more symmetric, and provide language constructs to work around the the differences that cannot be eliminated. For example, if nothing else, it would be helpful to allow for more closely relating dynamic and static principals in a program, perhaps by bounding the type of dynamic principal by a static one.

Jif suffers quite heavily from the problem of confusing error messages that plague many languages that perform constraint solving as part of type checking. The problem is that as the compiler solves constraints, a failure may occur due to a constraint that was introduced in a context distant from where the error is actually reported. The constraint failure could also be the result of a number of interacting annotations, making the cause of the problem difficult to pin down. There is a bit of literature already in existence on addressing this problem for Hindly-Milner type inference that could be relevant to Jif. Some solutions include calculating the minimum set of constraints responsible for the failure, and then displaying only those program slices that contributed to those constraints. There has also been work on type inference error explanation in terms of English, as well as searching for type isomorphisms in a program in order to provide suggestions to the user as to how they might go about fixing their code.

However Jif makes matters worse in some cases by silently joining the labels the programmer specified with the external program counter, leading to errors that seem to contradict the annotations. More appropriate behavior would to report that the programmer's annotation does not correctly respect the program counter rather than silently extending the label.

It might be interesting to consider an extension to the Jif language that allows for meta-programming. For example, it is most natural to describe a network protocol in a straightforward fashion with only the necessary communications. However, for some information flow policies the absence of a communication in many cases could result in an implicit flow. The solution is to insert dummy communications so that all cases look the same. However, this unnecessarily complicates the protocol description and implementation. One possible solution would be to generate the appropriate boilerplate code from a description of the protocol either via macros or run-time code generation.

While perhaps not particularly interesting technical problems, the Jif language's syntax could use some refinement to be more consistent (for example `authority` vs `where authority`) and less burdensome. Variable labels while perhaps convenient, are confusing and introduce additional dependency into the type system. A better solution would perhaps to introduce static label definitions for abbreviations.

Another significant problem, which likely to be resolved as development proceeds, is that it is presently very difficult to build the Jif system. Some of the authors of this paper have never actually succeeded in building a fully working version of the Jif compiler and runtime. In particular a single unified build system should be chosen, and some effort applied to building a regression test suite. Given the relative portability of the Java language, some effort should be made so that it is possible to build the Jif system on most platforms that have a JDK.

References

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

[2] Martín Abadi. Secrecy in programming language semantics. *Electronic Notes in Theoretical Computer Science*, 20, 1999. `http://www.elsevier.nl/locate/entcs/volume20.html`.

[3] Martín Abadi and Andrew Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[4] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as NTIS AD-A023 588.

[5] J. Daemen and V. Rijmen. Aes proposal: Rijndael, 1998.

[6] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

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

[8] Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. In *Preliminary Proceedings of the 17th Conference on the Mathematical Foundations of Programming Semantics (MFPS 17)*, Aarhus, May 2001. BRICS Notes Series NS-01-2, May 2001, pages 99–120.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996. ISBN 0-201-63451-1.

[10] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, January 1998.

[11] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

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

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

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

[15] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, CA, USA, May 1998.

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

[17] Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[18] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, September 1995.

[19] Benjamin Pierce and Eijiro Sumii. Relating cryptography and polymorphism, July 2000.

[20] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[21] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 2001. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

[22] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 355–364, San Diego, California, January 1998.

[23] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. In *Computer Security Foundations Workshop*, June 2001. To appear in *Journal of Computer Security*.

[24] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *10th IEEE Computer Security Foundations Workshop*, pages 156–168. IEEE Computer Society Press, June 1997.

[25] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of TAPSOFT '97, Colloquium on Formal Approaches to Software Engineering*, Lille, France, April 1997.

[26] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 268–276. ACM Press, January 2000.

[27] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[28] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Canada, June 2001.

[29] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *Transactions on Computer Systems*, 20(3):283–328, 2002.