Security-Typed Languages for Implementation of Cryptographic Protocols: A Case Study

Aslan Askarov and Andrei Sabelfeld

Dept. of Computer Science, Chalmers University of Technology, 41296 Göteborg, Sweden
{aaskarov, andrei}@cs.chalmers.se

Abstract. Security protocols are critical for protecting modern communication infrastructures and are therefore subject to thorough analysis. However practical implementations of these protocols lack the same level of attention and thus may be more exposed to attacks. This paper discusses security assurance provided by security-typed languages when implementing cryptographic protocols. Our results are based on a case study using Jif, a Java-based security-typed language, for implementing a non-trivial cryptographic protocol that allows playing online poker without a trusted third party. The case study deploys the largest program written in a security-typed language to date and identifies insights ranging from security guarantees to useful patterns of secure programming.

1 Introduction

Networked commerce, health, and military applications critically depend on underlying security protocols. Malicious attacks on these systems target vulnerabilities of two basic kinds—vulnerabilities of protocols and vulnerabilities of protocol implementations. Although the first kind of vulnerabilities is dangerous, the descriptions of security protocols are often open to public and are subject to thorough analysis by security experts. As a result, discovering and exploiting protocol-level weaknesses is significantly more daunting than attacking flaws in protocol implementations (cf. the need for a paradigm shift in cryptology [4]). This is also confirmed by CERT® incident reports where most of exploited flaws come from inadequate implementations. In the context of protocols, recent discoveries of multiple vulnerabilities in different implementations of the well studied SSL/TLS protocols [1] uncover insecure implementations that can be exploited to allow a remote attacker to execute arbitrary code.

To defend against implementation-level attacks, modern security technology relies on common principles for building secure software [47], including prudent techniques for deploying cryptographic software [23]. Moreover, since the paradigm shift in cryptology, much work has been done on timing, cache, power-consumption, and other implementation-level attacks. Nevertheless, the resulting principles and techniques are somewhat ad-hoc; they provide no end-to-end guarantees that systems preserve the confidentiality of secret data. For example, these principles and techniques provide little help in preventing an accidental leak of a secret key to a public field, or revealing a secret bid in an online auction before all participants have committed their bids. These are examples of undesired information flows that compromise confidentiality.

S. De Capitani di Vimercati et al. (Eds.): ESORICS 2005, LNCS 3679, pp. 197–221, 2005.

Security-typed languages have emerged over the past decade as an attractive approach to preventing insecure information flows (see [39] for a survey). These languages allow labeling sensitive data with security levels (naturally extending conventional types to *security types*). Security type systems regulate flows between data at different security levels, providing tight control over information flow.

Recent developments [33, 5, 37] raise hope for the possibility of regulating the propagation of sensitive information by security type systems in realistic languages. Furthermore, compilers for these languages such as Jif [36] (based on Java) and Flow-Caml [45] (based on Caml) have been developed. Nevertheless, "despite this large body of literature and considerable, ongoing attention from the research community, information-flow based enforcement mechanisms have not been widely (or even narrowly!) used" [48].

The challenge is whether security-typed languages scale up to real systems. In particular:

- *How helpful are security types for identifying potential insecurities in security-critical code?*
- *How laborious is the process of security typing? Does it force unnecessary restrictions on code?*
- *Is the security assurance provided by security types transparent enough?*
- *What is the general balance of benefits and drawbacks when using security-typed languages?*

Addressing these challenges seems impossible without practical experience in deploying security-typed languages. Motivated by this, we have performed an in-depth case study of securing an implementation of a non-trivial cryptographic protocol in the security-typed language Jif. To the best of our knowledge, this implementation is the largest program written in a security-typed language so far.

The focus of the case study is a protocol for online poker without a trusted third party (also known as *mental poker* [44]). This protocol has direct application in e-gambling, but it is also generally interesting because its security goals are similar to those of many other protocols. These goals include confidentiality in an environment of mutual distrust (in the absence of a trusted third party), auditability, fairness, and detection of cheating with high probability. This gives us a range of security properties that are useful in security-critical applications. For example, in online voting, it is important that every vote remains confidential yet the result (such as the number of votes for each candidate) becomes known to the public after the election is over. Besides confidentiality, a form of auditability is a desired security property here—it should be possible to recount the results. Another example with similar goals is an online auction protocol with mutual distrust. Participants reveal their secret bids only when the bidding phase has been completed. That the participants cannot alter their bids in the verification phase is also a form of auditability.

It is worth mentioning that the threat model adopted in this paper does not include covert channels that are due to probabilistic, timing, power-consumption, and cache behavior. Neither integrity nor availability issues are treated in our setting. While these restrictions are inherited from Jif's threat model, they are not fundamental to security-typed languages. Indeed, there are such languages capable of treating various covert channels [3, 41], as well as integrity [25, 49] and availability [51].

The case study has been conducted in three steps. First, we have implemented a baseline implementation in a conventional programming language (Java). Second, we have lifted this implementation to Jif. Finally, we have distributed the Jif implementation in order to simulate a realistic scenario where players run their parts of the protocol on their respective machines.

The case study has resulted in a range of insights into the challenges above (whose summary we defer to the conclusion). Further, the case study has suggested the need for richer mechanisms of information release (currently lacking not only in Jif but in most available information flow analyses). Additionally, we have developed patterns for secure programming that help streamline the process of security typing. Furthermore, we have uncovered some vulnerabilities and problems in Jif that lead to interesting directions for improvements.

The rest of the paper is organized as follows. Section 2 provides some background about the protocol for mental poker and the Jif language. Section 3 discusses the three different implementations. The lessons learned from the case study are reported in Section 4. Section 5 presents some programming patterns that have emerged from our experience in security-typed programming. Section 6 comments on related work. Section 7 concludes the paper.

2 Background

This section contains necessary background on protocols for mental poker and an introduction to security-typed languages and Jif.

2.1 Protocols for Mental Poker

Mental poker. In the popular card game of poker players with fully or partially concealed cards make wagers into a central pot. After several rounds of betting the pot is awarded to the remaining player or players with the best combination of cards. Mental poker is a well-known problem in cryptography on how to "play a fair game of poker [...] over the phone" [44] or how to play poker without a trusted third party (TTP). This problem continues to attract researchers and many solutions have been proposed [22, 12, 13, 29, 28, 43, 8, 6].

Crépeau has outlined some objectives for mental poker [12], summarized as follows:

1. Uniqueness of cards: every card must appear exactly once—either in the deck or in the hand of one player. This property can only be broken as a result of detectable cheating.
2. Uniform distribution of cards: the hand of each player must be possible with equal probability and must depend on decisions made by every player.
3. Absence of TTP: players trust neither each other nor any third party.
4. Cheating detection with high probability: the probability that a player may cheat without being detected must decrease fast (exponentially) with respect to some

security parameter that the players must agree on before the game. Also, the amount of work to accomplish the protocol should increase reasonably (polynomially) with respect to this parameter.
5. Complete confidentiality of cards: no information about any card from the deck may be obtained without the approval of every opponent. Also, no information may be obtained from a player's hand without his or her approval.
6. Minimal effect of coalitions: when more than two players are involved, some players could establish secret communication and exchange all their knowledge about the game. Nonetheless they should not be able to learn more than what they can deduce from the cards in their coalition.
7. Confidentiality of strategy: losing players may keep their cards secret at the end of a game.

Although protocols that claim to achieve all these properties have been proposed [13, 29, 28], they are demanding to computation time and are unacceptable in practice [18, 24]. For our case study, we have adopted a protocol by Castellà-Roca et al. [8] that achieves these properties (with the exception of the last one) and is practical in terms of computational requirements.

Castellà-Roca et al. TTP-free protocol. In this protocol all players cooperate in shuffling, so that no player coalition can force a particular outcome. Every player generates a random permutation of the card deck and keeps it secret; the player then commits this permutation using a bit commitment protocol. The shuffled deck is formed by the composition of all players' permutations.

Turning a physical card face down corresponds to encryption. Shuffling a card corresponds to a mathematical operation over the card's representation. The protocol uses an additive and multiplicative homomorphic cryptosystem, such as [17], to shuffle a deck of cards and maintain the privacy of the cards. The outcome of permuting an encrypted card and decrypting it is the same as if the card had been permuted without prior encryption.

When the game is over, the players reveal their encryption keys and permutations for validation. Requiring the disclosure of players' strategies after the game is a limitation of this protocol. On the other hand, it raises an interesting security goal of preventing hand revelation earlier in the game.

2.2 Security-Typed Languages and Jif

Secure information flow. Information flow from object x to object y occurs whenever the value of y is affected by the value of x. *Explicit flows* are results of assignment statements (e.g., `y=x`), I/O statements, and value returns by functions. The flow in these cases is caused by the operation explicitly; whether the operation is reached during execution does not necessarily depend on the value of x. By contrast, *implicit flows* [16] occur whenever x affects y through control flow, i.e., the execution of a statement that updates y depends on x. For example, in the fragment `y = 1; if (x == 0) y = 0`, the `if` statement causes an implicit flow from variable x to y.

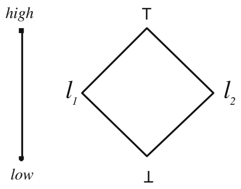


Fig. 1. Examples of lattices

The problem of information flow is relevant for security if, for instance, x stores sensitive information and y is a public system output. In this case, the control of how sensitive information propagates in the program is crucial for protecting confidentiality. Generally, program data can be associated with *security levels*, which constitute a *security lattice* [15]. The higher the security level is located in the lattice the more sensitive information is associated with this level. Figure 1 presents two examples of lattices: a two-element lattice with *high* and *low* levels corresponding to secret and public information; and a four-element lattice with a public element $\bot$, a top secret element $\top$, and two mutually incomparable intermediate elements ℓ_1 and ℓ_2. Information flow is considered *secure* if the level of the flow target is higher than (or the same as) the level of the flow origin.

Decentralized label model. The decentralized label model (DLM) [34] is a security model in which *principals* express their privacy concerns via *labels*. Principals in DLM (e.g., users, groups, and roles) may own, update, and release information. Labels are used to guarantee confidentiality. Every label consists of a set of *policies* that express privacy requirements. A privacy policy has two parts: an owner and a set of readers; and is written in the form *owner:readers*. By definition, an owner is implicitly contained in its readers set. A principal is allowed to read data if and only if it is contained in the reader sets of all policies of the label attached to the data.

Jif. Jif [33, 36] is an extension of the Java language with DLM labels. In Jif, methods can be granted *authority* to act for some set of principals. Authority regulates the ability of a method to *declassify* (or downgrade) the data: weaken or remove a policy in a label. This is possible if a policy is owned by a principal that is a part of the process authority.

An example of a label written in Jif syntax is `{Alice:Bob,Carol;}`. This label contains a single policy in which Alice is the owner; and Alice, Bob, and Carol are the readers. The label `{Bob:Alice;Alice:Carol;}` contains two policies. In this label Alice is the only principal present among readers of both policies. Hence, only Alice can read the data.

Variable types in Jif are composed of two parts: a regular Java type, such as `boolean`, and a security label, indicating how the value stored in this variable may propagate. For instance, the type `boolean{Alice:Bob}` represents a boolean that Alice owns and Alice and Bob can read. The bottom security level corresponding to public data has label `{}` (with the empty list of policies).

In security-typed languages implicit flows are often controlled by the *program-counter label* (*$\underline{pc}$*). This label tracks dependencies of the program counter. Recall an example given earlier (displayed in Listing 1) with some variable definitions. Here, the $\underline{\text{pc}}$ in the branch of the `if` statement captures the dependency on x and, thus, has label `{Alice:}`. The assignment statement is rejected by the compiler because the variable y is less secure than the $\underline{\text{pc}}$.

```
int {Alice:} x;
int {} y = 1;
...
if (x == 0) y = 0;
```

Listing 1. Implicit flow

Method declarations and constraints. Method declarations in Jif may be annotated with two optional labels, called *begin-label* and *end-label*. Begin-label is a lower bound on the side effects of a method. That is, Jif prevents calling a method if $\underline{pc}$ at the invocation point is higher than the begin-label of the method being invoked. By default, if no begin-label is specified, it is assumed that the method has no side effects and can be called regardless of $\underline{pc}$ of the caller (i.e., from any context). A method's end-label carries information about how much can be learned by observing if the method terminates normally or raises an exception. After the method invocation, $\underline{pc}$ of the caller is affected by the end-label of the method that has been called.

Arguments and return values can also be labeled with their security levels. An example of a method declaration is:

```
public boolean{Alice:Bob} validate{Alice:}(String{} s, int{} hash):{Alice:}
```

In this example, the function `validate` takes two arguments both of which are of the bottom security level. The return value has label `{Alice:Bob}`. Both the begin- and end-labels are `{Alice:}`.

Jif methods may contain a list of *constraints* prefixed by the keyword `where`. Two kinds of constraints are useful for our purposes: (i) `authority`($p_1, \ldots, p_n$), listing principals that this method is authorized to act for, and (ii) `caller`($p_1, \ldots, p_n$), listing principals whose authority the caller of the method is required to possess in order to run this method. We return to some security implications of these constraints in Section 5.4.

Exceptions. In contrast to Java, Jif disallows unchecked runtime exceptions. Consider the program (which is rejected by the Jif compiler) in Listing 2. If variable `secret` is zero, `ArithemeticException` is thrown by the method `div`. Observing whether this exception takes place would expose some information about the value of `secret`.

```
public class IntegerLeak {
    private int {Alice:} secret;
    public int{Alice:} div(int{} a) {
        return a/secret;
    }
}
```

Listing 2. Flow via exception

Parameterized classes. Jif classes and interfaces can be parameterized over labels and principals. This is useful for building reusable data structures. For instance, instead of writing two separate `Player` classes for Alice and Bob, which would only differentiate in the labels of the corresponding variables, one can write a single class `Player[P]` parameterized over principal variable `P`. Later in the instantiation, this parameter is replaced by the actual principal (e.g., Alice or Bob).

```
class X[label L] {
  private int {L} p;
  public int{L} getP() {return this.p;}
  public void setP{L} (int {L} n) {
     this.p = n;
  }
}
```

Listing 3. Parameterized class

Listing 3 displays an example of a Jif class parameterized over label `L`. This label is used in the declaration of class fields, such as `p`, and methods, such as `getP()` and `setP()`. An example of how this class can be instantiated with label `{Alice:}` is `X[{Alice:}]{Alice:} x = new X[{Alice:}]()`. Note the two labels that appear in the declaration of the variable `x`. The first one is a parameter of the class, while the second is the label of the referring variable.

Array labels. Being mutable data containers, arrays have two labels: one for the elements of an array, and the other for the array itself and its length. A single label for arrays would allow *laundering attacks* (i.e., code that exploits a vulnerability in a protection mechanism in order to leak more information than intended). Assume that arrays only had a single label. A variable `indeck` of type `int[]{}` with a single label `{}` could be assigned to a variable `hand` typed `int[]{Alice:}`. Then it would be safe to assign a variable `cardvalue` labeled `{Alice:}` to an element of array `hand`. However, this value would become visible through the variable `indeck`. This provides an illustration of a laundering attack.

An example array declaration is `int{Alice:}[]{} hand`. This array denotes Alice's hand of cards. The length of the array has the bottom label (`{}`); indeed, it is publicly known *how many* cards a player has. In contrast, the *values* of the actual cards are secret for others. Therefore, the elements of the array are labeled as `{Alice:}`.

Declassification. Many secrets have their lifetimes, after which they are not secrets anymore. Controlled information release or *declassification* is an important aspect of security-typed languages. It is safe to move data to a higher position in the security lattice. However, declassification relabels program variables so that the resulting label can be *less* restrictive than the original. Declassification in Jif is expressed via `declassify` statements. The process is required to have sufficient authority to declassify data. For example, to declassify a variable `x` of type `int{Alice:}` to `int{}` a process is required to have the authority of Alice. An example of a declassification statement is `y = declassify(x,{})`. Here, the `declassify` statement returns the value of x relabeled to `{}`, which is assigned to y.

3 Implementation

This section discusses the three different implementations we have developed [2]. The baseline implementation is in Java, the two remaining ones are in Jif. For both Jif implementations we assume the presence of two principals Alice and Bob (without loss of generality we assume two players).

3.1 Java Baseline Implementation

One motivation for an implementation in Java is to set a baseline implementation that would have been produced by ordinary Java programmers. Another reason is that debugging Jif programs often becomes burdensome. The baseline implementation follows Castellà-Roca et al.'s protocol for two players (Alice and Bob). It can be straightforwardly extended to multiple players. In this implementation, we have developed the main functional part of the program. A player is represented by a class `Player`. This class contains the player's data (such as the name, the hand of cards, cryptographic keys, and the game log) and methods that implement initialization, card drawing, and ending protocols. These methods are called by the game coordination routines.

3.2 Jif Implementation

The second implementation lifts the Java version to Jif (as realized by the current distribution Jif 1.1.1). Following the security objectives, we have adopted the security lattice in Figure 2. The sensitive information of the players carries the labels `{Alice:}` and `{Bob:}`. The data passed between the players is downgraded to the bottom level (`{}`).

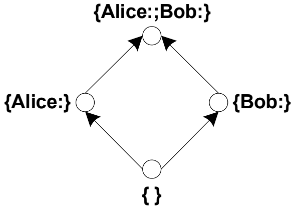


Fig. 2. Security lattice for Jif implementation

Lifting Java programs to Jif involves the following steps: (i) writing signatures for necessary Java API classes, (ii) changing some of the classes to Jif analogues, (iii) parameterizing classes over labels and principals, (iv) assigning labels to class fields, (v) assigning begin- and end-labels to functions and arguments, (vi) handling runtime exceptions, and (vii) writing helper functions for declassification of large data structures. Note that, there is no linear dependency in performing these steps—the process of lifting may (and is likely to) consist of a number of iterations and repetitive refactoring. Below we discuss these steps in detail.

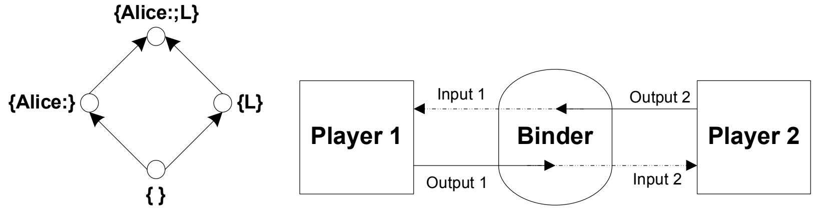
Writing signatures. To compile against existing Java API classes, the Jif implementation needs Jif signatures for these classes. Although writing signatures is a relatively simple task compared to Jif programs, this should be done with care. It is possible to misuse this feature (see Section 4.4 on problems and vulnerabilities related to signatures).

Changing to Jif analogues. Writing class signatures for Java classes can be avoided if there is a Jif analogue providing the same functionality. For example, the Jif implementation uses `jif.util.ArrayList` instead of `java.util.Vector`. The former is written completely in Jif, which makes its usage both safer and more convenient.

Parameterizing classes. Class parameterization is heavily used by the Jif implementation. Most of the classes are parameterized over an invariant label `L`, which stands for the security level of the information stored in instances of these classes. The main class of the implementation `Player[principal P, label L]` is parameterized over the player principal `P`, and the label of the output channel `L`. Therefore, in this class the label `{P:;L}` corresponds to the high label and `{L}` to the low one.

Assigning labels to class fields. It is important to identify which variables contain sensitive information and how restrictive their labels should be. It is sometimes convenient to use high labels for low data, for example, when a low variable is only used in a high context.

Assigning labels to functions and arguments. Recall that begin- and end-labels in method declarations are related to side effects in the program: in this implementation we identify side effects in Jif programs by the following events: (i) assignment to a non-final member variable, (ii) assignment to a mutable data structure such as an array or a class, and (iii) calling a method with side effects.



(a) Lattice in distributed implementation

(b) Distributed implementation architecture

Fig. 3. Distributed implementation

Catching/throwing runtime exceptions. Because unchecked exceptions may lead to information leaks, Jif requires runtime exceptions to be handled. Our Jif implementation uses the following disciplines:

- Declare and throw: an exception is declared in the method header and the responsibility to handle it is passed to the caller. However, this exception still has to be handled by the caller; moreover, its origin becomes obscured for the caller since it may be thrown at different points in the method.
- Avoiding exceptions: Jif compiler has a simple, yet useful, dataflow analysis that may detect if a local variable is null at a particular program point.
- Catch and ignore: there are two scenarios when a programmer might want to ignore an exception:
  - there are sufficient guarantees in the code that the exception may not be thrown, and
  - the programmer deliberately hides the presence of the exception.
- Catch and handle: exception may be handled, and a custom application exception is thrown by the method that contains information about the error.

The ArgCheck pattern (Section 5.1) describes how `NullPointerExceptions` caused by method arguments can be handled.

Modularizing declassification. See the Declassifier pattern in Section 5.2.

3.3 Distributed Jif Implementation

The third implementation has been developed to provide a "real-world" application of Jif. In this implementation players run as different processes and standard input/output is used as a communication medium.

Figure 3(a) displays the security lattice for one of the players (Alice). Here `L` is the label of the run-time environment. Sensitive variables in the program are labeled by `{Alice:;L}`. System outputs have the label `{L}`. The lattice for the other player's process is similar.

Figure 3(b) illustrates how the distribution works: two player processes communicate through a binding process so that I/O pipes of both processes are connected to that process.

The introduction of distribution to the Jif implementation involves these steps: (i) changing the logic of the coordinating process to take care of the distribution, (ii) writing helper classes for serialization of objects into strings and visa versa (strings can be easily exchanged by the processes), and (iii) writing a synchronization program that interconnects two processes via pipes.

4 Evaluation

This section reports on lessons we have learned from this case study. We compare the three implementations, evaluate security assurance provided by the security-typed implementations, discuss the role of declassification, and report some problems we have uncovered in Jif.

4.1 Comparison of the Three Implementations

Java vs. Jif implementation. Lifting Java programs to Jif is not straightforward. Section 5 presents useful programming patterns we have developed over the process of lifting. The Jif implementation is a result of successive refactoring iterations over the initial Java version. The main impact is caused by the security label annotations of program variables. These labels propagate further into the begin- and end-labels of the methods. Following this propagation, we have rewritten Java methods in such a way that the Jif version either repairs a discovered flow or declares it explicitly either via a declassification statement or via a method header. This technique increases assurance that the program protects the confidentiality of sensitive variables.

Explicit declassification helps specifying exactly which data is downgraded, who authorizes downgrading, and where in code it is downgraded. Thanks to declassification statements, intended leaks in the program are reduced to declassification points in the code. A detailed discussion of the declassification points in our implementation is presented later in this section.

Jif has helped uncover some insecurities in the Java implementation. Although it is not obvious how these particular insecurities can be exploited, they still represent potential vulnerabilities. One of the interesting insecurities we have discovered in the Java implementation is due to exceptions occurring at high security levels. Listing 4 illustrates this insecurity:

```
public class ExceptionLeak[label L] {
    private int{} readInput{}() { ... }
    public void exceptionLeak{}()  throws Exception{
        while (true) {
            int{} x = readInput();
            highMethod(x);
        }
    }
    private int{L} highDenominator;
    private int{L} highCounter;
    private int{L}[]{L} highArray;
    private void highMethod{}(int{} x)
    throws ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException {
            highArray[highCounter++] = x/highDenominator;
    }
}
```

Listing 4. Example of leaks via exceptions

Here, any of exceptions ArrayIndexOutOfBoundsException, ArithmeticException, or NullPointerException can be thrown in a high context reflecting problems in the high variables highDenominator, highArray, and highCounter. Therefore, the caller of highMethod on line 6 obtains information not only about the successful termination of the method but also details on what kind of exception has occurred and, in more complicated scenarios, the stack trace back to the origin of the exception! The SuccessFlag pattern (Section 5.5) describes how one can prevent such leaks.

Jif vs. distributed Jif implementation. While the second implementation reveals important security issues, the third one is more interesting from a practical point of view. From this perspective, the second implementation may be considered as an intermediate step toward the final distributed version. The second and third implementations have both been developed in Jif. Most of the code from the second implementation is reused in the third. Thus, the third implementation benefits from the security guarantees that are achieved in the second version and, in addition, is a simple, yet fully functional, example of a distributed program written in a security-typed language.

Because the distributed Jif implementation encompasses the features of the intermediate Jif implementation, the rest of the discussion refers to the distributed Jif implementation as the Jif implementation, unless specified otherwise.

4.2 Security Assurance

Recall from Section 2 the security objectives provided by the underlying protocol: (1) uniqueness of cards, (2) uniform distribution of cards, (3) absence of TTP, (4) cheating detection with high probability, (5) complete confidentiality of cards, and (6) minimal effect of coalitions. Although these objectives are addressed by protocol design, our goal is to ensure that the protocol implementation may not violate these objectives. Let us discuss how these objectives are addressed by the Jif implementation.

The first and the second properties rely on the random number generators, supported by Java API. The third property (absence of TTP), which is crucial to the protocol, is not broken by our implementation. Indeed, the implementation does not introduce a TTP as there are only two principals—the players. Cheating detection is supported in our implementation via logging the messages that players exchange. In the verification phase of the game this log is used to check if either of the players has cheated. Note that the sixth objective does not apply to a two-player implementation. For a multi-player extension, control over the propagation of sensitive information (as provided by the security-type system) is essential for minimizing the effect of coalitions.

The most interesting objective is the fifth one (complete confidentiality of cards). It is for this objective that we capitalize on using security types. Tight control of confidentiality is guaranteed by assigning high-security labels to the following program variables:

- instance of the cryptosystem that stores the secret key for homomorphic encryption;
- signature key used for signing messages;
- player’s hand: decrypted values of the player’s cards; and
- player’s secret permutation (for commutative shuffling) and variables related to this permutation.

Table 1. Declassification points

GR	POINT	WHAT	WHO	WHERE	WHEN
I	1	Public key for signature	Anyone	Initialization	Before game start, before seal is open
	2	Public security param	Player	Initialization	Before game start, before seal is open
II	3	Message signature	Player	Sending msg	Any time
	4–7	Protocol init data	Player	Initialization	Before game start, before seal is open
	8–10	Encrypted permuted card	Player	Card drawing	During game, before seal is open
III	11	Decryption flag	Player	Card drawing	During game, before seal is open, after player obtains card
IV	12	Player's secret encryption key	Player	Verification	After game end, after seal is open
	13–14	Player's secret permutation	Player	Verification	After game end, after seal is open

Labeling these variables as high restricts the flow of sensitive data to public outputs. Jif's type system prevents unintended flows of sensitive information unless otherwise specified by `declassify` statements. This reduces the manual security analysis of the system down to inspection and justification of `declassify` statements in code. This is the subject of the following section.

4.3 Authority and Declassification

Label assignment reduces possibilities for information leaks to the program points where declassifications occur. Declassification is possible if the running process has enough authority to relabel the data. The ability to grant a class or a method authority is useful but also a potentially dangerous feature since this authority may be misused for inappropriate declassification of confidential information. The Jif implementation grants the authority of the player to the following two functions: (i) function that returns the public key of the player and (ii) function that coordinates the game process.

There are 14 declassification points in the main class of the Jif implementation. For each declassification Table 1 states *what* is declassified, *who* declassifies data, *where* in the program, and *when* declassification may occur (the last column uses the notion of seal defined in group IV below). These aspects correspond to dimensions of information release [42]. Accordingly, the declassification points can be naturally grouped as follows.

I Declassification of naturally public data (points 1–2). Functions for generating signature keys return `KeyPair` data structure. It contains both a private and a public component and, as a whole, is labeled as high. In order to return the public key, a separate high copy of the key has to be obtained and declassified. Similarly, the public parameter of the homomorphic cryptosystem is extracted from an instance of the cryptosystem. Again, we obtain a high copy of this parameter and declassify it separately. These declassifications are safe, since they affect neither the sensitive part of the key nor the secret parameter of the cryptosystem.

II Declassification following signatures and encryptions in the underlying protocol (points 3–10). This is the largest group of declassification points. The first such point is related to the digital signature of the messages. Because computation of the signature involves a private key, the result gets tainted by the high label of the private key and becomes high. Here we rely on the cryptographic properties of the calculated signature and assume it is safe to declassify the computed result. The obtained declassified signature is attached to the message.

The rest of the declassifications in this group are applied to encrypted values created in the context with a high $\underline{pc}$ label. The motivation for these declassifications is similar to the one for signatures. These declassifications are safe as far as we trust the underlying cryptographic protocol.

III Declassification of the success flag in `finishCardDraw()` (point 11). See the SuccessFlag pattern (Section 5.5) for motivation and details.

IV Declassification of sensitive information for verification (points 12–14): After a game completes, the protocol requires players to exchange their private keys and secret permutations in order to verify the fairness of each other. This is a common scenario for security protocols that rely on bit commitment. It is important that these functions are not used earlier than they are supposed to, i.e., it is important *when* information is downgraded. Jif's declassification mechanism is not powerful enough to support such temporal properties. Therefore, we introduce a so called *seal*, a boolean flag that changes its value at most once after initialization. The seal is initialized in the constructor of the `Player` class. Its integrity is checked in the methods that implement game protocols. The seal is *opened* once the sensitive information that it protects is released. This is done in the methods that declassify the keys and secret permutation. Next time there is a call to a method that assumes the seal's integrity, a runtime exception is thrown indicating that this call violates the security properties of the protocol. That is, one is not allowed to declassify data prematurely. Implementation details of the seal technique are presented in the section on patterns.

The artifact of four different categories of declassification (with independent reasons for justifying each of them) opens up the question of an adequate treatment of the multifaceted nature of declassification. In particular, there is need for enforcing temporal information release policies. This provides a basis for our future work (cf. Section 7).

4.4 Jif's Vulnerabilities and Problems

Many of the insights we have gathered in the case study are not Jif-specific. However, as Jif is the most ambitious security-typed language to date, it is useful to highlight some vulnerabilities and problems discovered in the Jif compiler.

Signature misuse. Jif uses existing Java libraries by means of class signatures. A signature is a file with Jif-style method declarations. Jif programs are type-checked against these headers, but pre-compiled Java binaries are used at runtime. An example of where we use Java API with Jif signatures is DSA signature scheme, which has internal random values that need to remain secret.

However, a flow can be easily introduced if labels declared in the method header do not correspond to the code in the library. An example of such a weakness is a signature of `System.arraycopy` function from the current Jif distribution.

```
public static native void arraycopy(Object{dst} src, int{dst} src_position, Object dst,
                                    int{dst} dst_position, int{dst} length)
throws (IndexOutOfBoundsException, ArrayStoreException, NullPointerException);
```

This method has no begin-label, which implies the absence of side effects. However, the copy of an array in the memory is an obvious side effect that should be reflected in the begin-label of the method. Listing 5 is an example of how this weakness can be exploited.

```
public class TestLeak[label L] {
    private int{L}[] secret;
    private int{}[]  output;
    public void leak() {
        try { System.arraycopy(secret, 0, output, 0, secret.length);
        } catch (Exception ignored) { }
    }
}
```

Listing 5. Leakage via invalid method signature

In this example, function `leak()` calls `System.arraycopy` to copy data from the high array `secret` into the low array `output`. Nevertheless, this code is accepted by the Jif compiler since it trusts the provided method signature.

Parameterized signatures. Recall class `X` from Section 2.2 with label annotations erased to obtain a Java class. Listing 6 is an example of a vulnerable signature that can be written for such class. Here, `{this}` is a label of the current instance. This signature is exposed to the attack similar to the array-laundering attack from Section 2.2. However, this attack can be prevented by parameterizing the signature as it is shown in Listing 7.

```
class X {
  public native int {this} getP();
  public native void setP{this}(int{this}n);
}
```

Listing 6. Vulnerable signature for class

Generally, a class should be parameterized if a variable of that class can be modified after the instantiation. While such flows are captured in pure Jif programs by the type system, they are not prevented in class signatures. It is the author of a signature who is responsible for its correctness.

```
class X[label L]{
  public native int {L} getP();
  public native void setP{L}(int{L} n);
}
```

Listing 7. Correct signature for class

Relabeling mutable data containers. Assume there is an array `x` of type `int{}[]{}` which we want to relabel to type `int{Alice:}[]{Alice:}`. The assignment statement `int{Alice:}[]{Alice:} y = x` is rejected by the Jif type system as it is exposed to the laundering attack similar to the one described in Section 2.2. A possible solution is to create a separate copy of an array, upgrading elements one-by-one.

```
int{Alice:}[]{Alice:} x = new int[y.length];
for (int i = 0; i < y.length, i++) y[i] = x[i];
```

Similar code has to be written if one wants to relabel an instance of some parameterized class. Declassification is another example of relabeling, so the same argument applies when there is need to downgrade an array or a class instance. As a consequence, the programmer is forced to write relabeling (and declassification) code for every complex data structure that is used at different security levels. In our Jif implementation, we resolve this problem with the Declassifier pattern (Section 5.2).

Missing Java features. Although Jif includes a large subset of Java, some useful features of Java, such as inner classes and `super` calls, are missing. The lack of inner classes and `super` calls have not proven a substantial obstacle. Most important in the context of this case study is the lack of support for serializability for parameterized classes. As a result, serialization routines used in the distributed implementation need to be written manually for every class.

5 Programming Patterns

As discussed in Section 4, enriching Java code with security types is not straightforward. To help streamline this process we have developed patterns for secure programming in Jif. These patterns help resolve insecurities in baseline code in a uniform and transparent fashion. Appendix A presents examples and code listings for each pattern sketched below.

5.1 ArgCheck: Checking Arguments in Jif

This pattern suggests raising `IllegalArgumentException` if an argument provided to a method is null. This exception type is more informative than `NullPointerException`. Also, because an exception is raised, Jif's built-in `NullPointerException` analysis ensures that `NullPointerException` no longer needs to be handled for this argument, which results in transparent code for the rest of the method.

5.2 Declassifier: Declassification of Large Data Structures

Because arrays and parameterized classes are mutable data containers they cannot be completely declassified with a single `declassify` statement. Each field of such a class or element of an array needs to be relabeled separately. This pattern uses a single class `Declassifier` that contains static declassification and upgrade methods for every data type used in the program. This class is parameterized over a principal `P` whose authority is used for declassification and a label `L` to which the data is downgraded. Thus, declassification methods in this class accept arguments of the level `{P:;L}` and return low copies of them relabeled to `{L}`. Similarly, upgrade routines return high copies of their arguments relabeled from `{L}` to `{P:;L}`.

In this pattern, the class `Declassifier` has no authority, and all declassification methods in the class require the caller to have the authority of `P` to declassify data. One can also imagine an alternative version of this class which combines encryption and declassification transparently for the caller. Then, the caller of the method receives an encrypted (and declassified) value without having the authority necessary for declassification. We can assume that this kind of declassification is safe as the caller of the method receives an encrypted value. While helpful, this scenario is not used in our implementation since it is the player who owns, encrypts, and declassifies the data.

5.3 EffectOrder: Ordering Effects

Some declassifications in a program may be avoided if the code is rearranged so that low operations (such as input) precede operations that affect the $\underline{pc}$ label.

5.4 ReqAuth: Requiring Authority vs. Granting it

Jif's `authority` clause in method declarations can be easily misused. This clause grants the authority of a principal opening up possibilities for declassification by any caller. Often, it is safer to use a `caller` clause. This avoids granting authority and prevents the method from being called when the calling process does not have the required authority.

5.5 SuccessFlag: Declassification of a Success Flag

This pattern prevents the propagation of exceptions thrown at a high security level to callers at a lower level. A caller is still notified about the failure; however, no detailed data, such as the call stack, is passed to the caller. Instead, the high code that can generate an exception is enclosed by a `try...catch` block; a boolean flag tracks whether the `try` block has terminated with an exception. This variable is then declassified immediately after the `try...catch` block; a low exception is thrown depending on the flag's value.

5.6 Seal: Seal Class

Sealing is used to enforce temporal properties such as preventing secret-key declassification from happening earlier in the game. Unlike the other patterns, this is a combination of conventional programming techniques and security features of Jif.

The class has two parameters: the owning principal `P` and the label `L`. This label stands for the lowest security level at which the seal is visible. The value of the seal is stored in the boolean variable `open`, which is only accessible to the owner `P`. The `caller` constraint in the class constructor requires that the calling process should have the authority of `P`. Initially, the value of the variable `open` is `false`. It may only change to `true` in the method `unseal()`. Similar to the constructor, this method has the `caller` constraint. This prevents calling `unseal()` from program contexts that do not have the authority of the seal's owner. Note that the actual validation occurs at run-time.

The method `isOpen()` returns the value of the seal. It grants the authority of the owning principal to the process in order to declassify the current value to the visible level `L`. The method `assertIntegrity()` is similar to `isOpen()` and is a suggested way of checking whether the seal has been opened.

5.7 KeySignature: Signature for Key Generation

This pattern shows how the signature of a class that generates encryption keys can be specified to avoid declassifying public keys (declassification group I in Section 4). This signature is parameterized over two labels corresponding to the labels of public and private keys. Then, method headers can be written in such a way that `getPublic()` returns a low value and `getPrivate()` returns a high one. Thus, declassification is avoided. This, however, does not eliminate the flow but makes it invisible to Jif.

6 Related Work

Although the theoretical area of information flow security is rather mature [21, 19, 38, 39], there is little evidence for the scalability of information flow controls in practice. Below we discuss some latest progress in this area.

On the security assurance side, this work fits into a recent classification of declassification [42] that, however, considers only information release policies (not declassification mechanisms). Nevertheless, the what, who, where, and when columns of Table 1 correspond to the dimensions of declassification from [42].

A related recent development that investigates the practical use of declassification policies is Li and Zdancewic's work on web scripting languages. With the target of enforcing *relaxed noninterference* [30] they develop a type system for web programming [31]. To what extent this language addresses challenges for practical security has not so far been reported, however.

Giambiagi and Dam have investigated how *admissibility* justifies the security of a simple payment protocol [14]. In subsequent work [20], they separate protocol specification from its implementation such that implementation is guaranteed to reveal no more information than the specification of a protocol. The implementation language, however, is rather distant from a realistic language like Jif. Recently, Chong and Myers have considered temporal release policies in the context of an ML-like language [9, 10]. Their *noninterference "until"* policies are intended to guarantee that secrets are released after a certain statically-enforceable condition becomes true. This approach, however, abstracts away from how the release conditions are enforced. An intriguing direction for future work is exploring the sealing technique further in order to enforce conditional release policies similar to admissibility and noninterference "until."

Heldal et al. [26, 27] show how UML can be integrated with Jif in order to introduce declassification early in the design process. This line of work is promising for modularizing declassification and can lead to a way of combining declassification-free Java code with security-critical Jif code in such a way that declassification statements agree with declassification at the modeling level.

Jif/split [49, 50] performs systematic partitioning of Jif programs into distributed components. Unfortunately, Jif/split does not support parameterized classes (due to compatibility issues with Java's serialization) which would be an obstacle for splitting our Jif implementation of the protocol.

As for the largest implementations in security-typed languages reported so far, we are aware of a battleship game protocol implemented in Jif/split [49, 50] and an evaluation of an earlier version of Jif on a library of cryptographic primitives [46]. However, these implementations are relatively light (500 and 800 lines of code, respectively, vs. 4500 lines in this study).

7 Conclusion

As a proof of concept, we have implemented a non-trivial cryptographic protocol in a security-typed language. The implementation has resulted in the largest program written in a security-typed language to date. The case study has given useful evidence on challenges for practical information flow security (cf. Section 1). We discuss insights into these challenges in turn:

- *How helpful are security types for identifying potential insecurities in security-critical code?*
  We have found security types useful for preventing explicit and implicit insecure flows. We have also uncovered insecurities in the baseline implementation due to liberal handling of exceptions and mutable data structures (cf. Section 4.1 and 4.4).
- *How laborious is the process of security typing? Does it force unnecessary restrictions on code?*
  All three implementations have been coded by a graduate student (the first author). The baseline implementation consumed around 60 man-hours of development work. The Jif implementation and distributed Jif implementation consumed 150 and 80 man-hours respectively, excluding the time to learn Jif. The case study indicates that although lifting Java code to Jif takes some experience to master, the security-typed result is not significantly distant from the original code. Furthermore, we have developed patterns for secure programming (cf. Section 5) to make programming with security types clearer and more convenient.
- *Is the security assurance provided by security types transparent enough?*
  This is the territory of the most interesting findings. Jif's mechanism for declassification has proven to be useful for localizing information release to certain well-marked parts of the program (`declassify` points). The case study, however, suggests that there are various reasons for declassifying at different points. Not only does one need to control *what* information is released, by *whom* and *where* in the system, but also *when* it is safe to release information (cf. Section 4.3). For example, the declassification of the result of encryption (as in the card shuffling phase) and the declassification of the secret key (as in the commitment verification phase) have distinct reasons and hence need to be protected in distinct ways. One disadvantage of Jif (as many other information release mechanisms [20, 40, 35, 32, 30]) is that its treatment of declassification disregards the multifaceted nature of declassification.
- *What is the general balance of benefits and drawbacks when using security-typed languages?*
  Apart from the (dis)advantages we have already discussed, modularity (due to the compositionality of the type system) and selective type annotation (due to security type inference) have proven particularly helpful. On the other hand, debugging tools and extensive documentation currently lack for security-typed languages, which forces programmers to debug code in baseline implementations written in conventional languages. Another unaddressed issue is the connection between high-level

security policies for information release and declassification statements in the code. One can argue that manual inspection of declassification points can, in some cases, be acceptable but, generally, there is need for expressing security policies in high-level languages (perhaps modeling languages) ensuring that these policies are enforced in code. For recent steps in this direction, see [26, 27, 7].

Future work. The case study has strongly suggested that existing information release mechanisms need further improvement. A lesson we have learned is that different kinds of declassifications need to be treated differently.

In order to alleviate this problem, we plan to generalize the seal-like data construction to represent declassification that may only happen once a certain condition has been satisfied (as, e.g., the mental poker protocol has reached its verification phase, or all bids are placed in an online auction protocol). This would lead to enforcing stronger (and more intuitive) security guarantees statically.

At the next level of ambition, we intend to connect language-based declassification to security assurance that is provided with respect to information release policies. For example, it remains to be seen how this approach can be connected to conditional release policies such as admissibility [14, 20] and noninterference "until" policies [9, 10]. A long-term goal is to provide a toolbox of declassification mechanisms for each of the *what*, *who*, *where*, and *when* axes of information release [42].

Another strand of worthwhile future work is improving Jif's shortcomings (cf. Section 4.4). We plan to explore automated refactoring techniques for pattern design (e.g., [11]) in order to facilitate program transformations that result in security typed programs. Another interesting problem is connected to relabeling mutable data structures. We believe it can be improved by introducing an operation that would combine declassification and object cloning so that a relabeled separate copy of an object would be created. This would prevent laundering attacks via object aliases. Also, this would make programming in Jif easier because traversing mutable data structures for the sole purpose of declassification would no longer be needed. A structured way of doing this is by defining a Jif interface `Declassifiable` that would allow an operation `declassifyAndClone` to be performed on the classes that implement the interface.

Although the case study is the largest of the kind, it is not large enough to be an example of real production code. In order to investigate additional subtleties that come with such code, we plan to run a student project to extend the case study to a fully fledged application for mobile devices.

Acknowledgments. Thanks are due to Michael Hicks, Boniface Hicks, Stephen Chong, and anonymous reviewers for helpful comments.

References

1. CERT® advisory CA-2003-26: Multiple vulnerabilities in SSL/TLS implementations, October 2003. `http://www.cert.org/advisories/CA-2003-26.html`.
2. Jif source code for the mental poker protocol, March 2005. `http://www.cs.chalmers.se/∼aaskarov/jifpoker`.

3. J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, January 2000.
4. R. Anderson. Why cryptosystems fail. In *ACM Conference on Computer and Communications Security*, pages 215–227, November 1993.
5. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 253–267, June 2002.
6. A. Barnett and N. P. Smart. Mental poker revisited. In *Proc. Cryptography and Coding IMA International Conference*, volume 2898 of *LNCS*, pages 370–383. Springer-Verlag, December 2003.
7. L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. ACM Conf. on Programming Language Design and Implementation*, June 2005. To appear.
8. J. Castellà-Roca, J. Domingo-Ferrer, A. Riera, and J. Borrell. Practical mental poker without a TTP based on homomorphic encryption. In *Progress in Cryptology-Indocrypt*, volume 2904 of *LNCS*, pages 280–294. Springer-Verlag, December 2003.
9. S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, October 2004.
10. S. Chong and A. C. Myers. Language-based information erasure. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005. To appear.
11. M. Ó. Cinnéide. Automated refactoring to introduce design patterns. In *Proc. ACM International Conference on Software Engineering*, pages 722–724, 2000.
12. C. Crépeau. A secure poker protocol that minimizes the effect of players coalitions. In *Advances in Cryptology: Crypto'85*, volume 218 of *LNCS*, pages 73–86. Springer-Verlag, 1986.
13. C. Crépeau. A zero-knowledge poker protocol that achieves confidentiality of the players' strategy or how to achieve an electronic poker face. In *Advances in Cryptology: Crypto'86*, volume 263 of *LNCS*, pages 239–247. Springer-Verlag, 1987.
14. M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proc. IEEE Computer Security Foundations Workshop*, pages 233–244, July 2000.
15. D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.
16. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
17. J. Domingo-Ferrer. A new privacy homomorphism and applications. *Information Processing Letters*, 60(5):277–282, 1996.
18. J. Edwards. Implementing electronic poker: A practical exercise in zero-knowledge interactive proofs. Master's thesis, Dept. of Computer Science, University of Kentucky, 1994.
19. R. Focardi and R. Gorrieri. Classification of security properties (part I: Information flow). In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 331–396. Springer-Verlag, 2001.
20. P. Giambiagi and M. Dam. On the secure implementation of security protocols. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 144–158. Springer-Verlag, April 2003.
21. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.
22. S. Goldwasser and S. Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *Proc. ACM Symp. Theory of Computing*, pages 365–377, 1982.
23. P. Gutmann. Lessons learned in implementing and deploying crypto software. In *Proc. USENIX Security Symp.*, pages 315–325, August 2002.

24. R. Hanna, A. Rideout, and D. Ziegler. Secure peer-to-peer texas hold'em. Course project, MIT. `http://web.mit.edu/ardonite/6.857/`, 2003.
25. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, January 1998.
26. R. Heldal and F. Hultin. Bridging model-based and language-based security. In *Proc. European Symp. on Research in Computer Security*, volume 2808 of *LNCS*, pages 235–252. Springer-Verlag, October 2003.
27. R. Heldal, S. Schlager, and J. Bende. Supporting confidentiality in UML: A profile for the Decentralized Label Model. In *Proc. International Workshop on Critical Systems Development with UML*, pages 56–70, 2004.
28. K. Kurosawa, K. Katayama, and W. Ogata. Reshufflable and laziness tolerant mental card game protocol. *IEICE Transactions*, E80-A(1):72–78, 1997.
29. K. Kurosawa, Y. Katayama, W. Ogata, and S. Tsujii. General public key residue cryptosystems and mental poker protocols. In *Advances in Cryptology: EuroCrypto'90*, volume 473 of *LNCS*, pages 374–388. Springer-Verlag, 1991.
30. P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 158–170, January 2005.
31. P. Li and S. Zdancewic. Practical information-flow control in web-based information systems. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005. To appear.
32. H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, November 2004.
33. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.
34. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.
35. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004.
36. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2004.
37. F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, January 2003.
38. P. Ryan. Mathematical models of computer security—tutorial lectures. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 1–62. Springer-Verlag, 2001.
39. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
40. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.
41. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.
42. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005. To appear.
43. C. Schindelhauer. A toolbox for mental card games, 1998. `http://citeseer.ist.psu.edu/schindelhauer98toolbox.html`.
44. A. Shamir, R. Rivest, and L. Adleman. Mental poker. *Mathematical Gardner*, pages 37–43, 1981.
45. V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/∼simonet/soft/flowcaml/`, July 2003.

46. S. Tse and G. Washburn. Cryptographic programming in Jif. Course project, 2003. `http://www.cis.upenn.edu/∼stse/bank/main.pdf`.
47. J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.
48. S. Zdancewic. Challenges for information-flow security. In *Proc. Programming Language Interference and Dependence (PLID)*, August 2004.
49. S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. ACM Symp. on Operating System Principles*, pages 1–14, October 2001.
50. L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 236–250, May 2003.
51. L. Zheng and A. C. Myers. End-to-end availability policies and noninterference. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005. To appear.

A Programming Patterns

We describe further details on some programming patterns that one may find useful when programming in Jif, or lifting Java programs to Jif. A brief summary of these patterns appears in Section 5.

A.1 ArgCheck: Checking Arguments in Jif Functions

The obligation to handle runtime exceptions in Jif can easily make code more clunky than necessary. Consider, for example, a Java function:

```
public boolean validate(byte [] p, Matrix mx) {
    if (!mx.validate(p)) return false;
}
```

If the argument `mx` is null, `NullPointerException` is thrown. Straightforward porting of this function may lead to the following Jif code:

```
public boolean validate{L}(byte{L}[]{L}p, Matrix[L]{L} mx):{L} throws NullPointerException{
    if (!mx.validate(p)) return false;
}
```

This code declares the exception in the method header. Listing 8 illustrates how the above function can be modified so that `IllegalArgumentException` is thrown whenever an argument is `null`. Due to Jif's `NullPointerException` analysis it is not necessary to handle `NullPointerException` for `mx`.

```
public boolean validate{L}(byte{L}[]{L} p, Matrix[L]{L} mx):{L}
throws IllegalArgumentException {
    if (mx == null) throw new IllegalArgumentException();
    if (!mx.validate(p)) return false;
}
```

Listing 8. Checking arguments in Jif functions

This pattern achieves more transparent code as all arguments are checked in advance before they are used; also, the type of a declared exception is more specific to the error.

A.2 Declassifier: Declassification of Large Data Structures

Consider the class `IntPair` in Listing 9. Listing 10 is an example of a method in `Declassifier` that downgrades an object of the type `IntPair[{P:;L}]{P:;L}` to the type `IntPair[L]{L}`. The first declassification on line 2 downgrades the reference to the object. Line 4 declassifies the fields of the object `a` and constructs a new object at the low security level that has the same value as `a`.

```
public class IntPair[label L] {
    private int x, y;
    public IntPair(int{L} _x, int{L} _y) {
        this.x = _x; this.y = _y;
    }
    public int{L} getX() { return this.x; }
    public int{L} getY() { return this.y; }
}
```

Listing 9. IntPair

```
  public static IntPair[L]{L} declassifyIntPair(IntPair[{P:;L}]{P:;L}a1) where caller(P) {
2     IntPair[{P:;L}] a = declassify(a1, {L});
      if (a == null) return null;
4     return new IntPair[L](declassify(a.getX(),{L}),  declassify(a.getY(),L));
  }
```

Listing 10. Method

A similar approach needs to be applied for upgrading a mutable object from one security level to another. A disadvantage of this approach is that it requires the constructors of the relabeled classes to have all class fields as arguments. This is not always desirable since the values of private variables, (e.g., the internal state of an object) are not always supposed to be instantiated in constructors.

A.3 EffectOrder: Ordering Effects

Some program statements in Jif may affect the pc label. If pc is high, low side effects are not allowed without prior declassification. Sometimes, it is possible to avoid such a declassification by ordering program statements so that low side effects precede the statements that taint pc with a higher label. Consider an example where `readData` and `update` functions are defined as follows:

```
public String{L} readData{L}():{L} { ... }
public void update{L;H}(String{L} x):{L;H} throws NullPointerException { ... }
```

Now, consider a code snippet where low input is interleaved with high `update` calls and which is rejected by the compiler (unless pc is declassified before the second read).

```
1   String{L} a = readData(); // low side effect
2   update(a);                // high statement, pc becomes high
3   String{L} b = readData(); // low side effect, rejected because pc is high
4   update(b);                //
```

After the call to `update` function on line 2, pc becomes high, and the call to the function `readData` on line 3 is rejected. The revision below demonstrates how this can be repaired by ensuring that the low function calls precede the high ones.

```
String{L} a = readData(); // low side effect
String{L} b = readData(); // low side effect, ok
update(a); // high statement effect, ok
update(b); // high statement effect, ok
```

A.4 ReqAuth: Requiring Authority vs. Granting it

Consider examples of two functions that reveal the secret permutation of the player, declassifying it to a lower label:

```
public byte{L}[]{L} revealPermutation{L}() where authority (P) { ... }
```

```
public byte{L}[]{L} revealPermutation{L}() where caller (P)    { ... }
```

The first example grants the authority of the player to the method. This implies that the method can be called from any context. Such a declaration is dangerous because secret information can be easily leaked. In contrast, the second example grants no authority but requires the caller to have necessary authority for declassification: calling this method from a context where the calling process does not have the authority of the principal P is rejected by the compiler.

A.5 SuccessFlag: Declassification of a Success Flag

Listing 4 in Section 4.1 is an example of code that needs to release information about the termination of the method without leaking the details on why the method failed to terminate normally. Listing 11 shows how one can use a boolean flag variable to track the termination path of high methods.

```
1  public void foo{L}():{L} throws Exception where caller(P) {
2      boolean ok = false;
3      try { ... // code that can throw high exception
4            ok = true;
5      } catch (Exception ex) { ... } // handling high exception
6      if (declassify (!ok, {L})) { throw new Exception(); }
7  }
```

Listing 11. Declassification of success flag

In this example, the boolean flag ok is initialized on line 2. High code that can generate exceptions is enclosed by a try ... catch statement, so that possible exceptions are caught and handled on line 5. The assignment ok=true on line 4 may generate no exception and is the last one within the try block. Line 6 declassifies the value of this variable and, depending on this value, it may generate a low exception that will propagate to the caller.

A.6 Seal: Seal Class

Listing 12 presents the structure of the seal class that is described in Section 5.6.

```
/* Seal belongs to a principal P, and is visible at the level L */
public class Seal[principal P, label L]  authority(P) {
    private boolean{P:;L} open; /*actual value of the seal */
    /* require the principal to create this */
    public Seal{P:;L}() where caller(P) {  this.open = false;    }
    /* require the principal to unseal it */
    public void unseal{P:;L}() where caller (P) {  this.open = true;  }
    /* anyone at the level L can check it */
    public boolean{this;L} isOpen():{L} where authority (P) {
        return declassify (open, {this;L});
    }
    /* similar to previous */
    public void assertIntegrity():{L} throws SecurityException {
        if (this.isOpen()) throw new SecurityException();
    }
}
```

Listing 12. Seal class

Listing 13 shows how sealing is used. Line 1 declares a seal that belongs to Alice and is observable by everyone. It is initialized in the method `init()`. The `work()` method checks if the seal has been opened before it is called. The seal is opened in `revealSecret()` method. If `work()` is called after the seal is opened, the exception `SecurityException` is thrown.

```
1  private Seal[Alice,{}] seal; // declaration
2  public void init() where caller(Alice) { // initialization
     this.seal = new Seal[Alice, {}]();
     ...
   }
   public void work()  throws SecurityException, NullPointerException{
     this.seal.assertIntegrity();   // check the integrity in the beginning of the method
     ...
   }
   public void revealSecret() where caller (Alice)  throws NullPointerException{
     this.seal.unseal();
     ... // declassification goes next
   }
```

Listing 13. Usage of seal

A.7 KeySignature: Signature for Key Generation

Listings 14 and 15 are two signatures for Java's `java.security.KeyPair` class. The usage of the first one requires an instance of the `KeyPair` class to be high, because it contains a sensitive private key. Declassification is applied when information about the public key is needed. This declassification is safe because the released information is naturally public.

```
public final class KeyPair{
  public KeyPair(PublicKey{this} publicKey,
                 PrivateKey{this} privateKey) {}
  public native PublicKey{this} getPublic();
  public native PrivateKey{this} getPrivate();
}
```

Listing 14. Non-parameterized signature for KeyPair

The declassification-free version is parameterized over two labels—for the private and public keys. In this case, a parameterized signature avoids declassification by labeling method headers appropriately. Both of the approaches are acceptable, providing a trade-off between explicit flow control and elegance.

```
public final class KeyPair[label L, label H]{
  public KeyPair(PublicKey{L} publicKey,
                 PrivateKey{L;H} privateKey) {}
  public native PublicKey{L} getPublic();
  public native PrivateKey{L;H} getPrivate();
}
```

Listing 15. Parameterized signature for KeyPair