Typing Computationally Secure Information Flow in Jif

Liisi Haav[1] Peeter Laud[1,2]
[1]Tartu University [2]Cybernetica AS
{liisi222|peeter.laud}@ut.ee

Abstract

We investigate how to model type systems for computationally secure information flow within the limits of the type system of Jif — an extension of Java with types for tracking the flow of information. In particular, we consider a type system proposed by Laud and Vene which can handle encryption keys as first-class data. We show how the typing decisions of Laud-Vene type system can be captured using the declassification mechanism of Jif, and present a Jif class for "keys" that encapsulates all necessary information releases. The rules that a user of the defined class has to follow, in order to be consistent with the Laud-Vene type system, can be syntactically checked in a straightforward manner.

1 Introduction

The question of *secure information flow* in a program (or a larger system) arises if the program has inputs and outputs of different *security levels*. A common way of specifying secure information flow is *non-interference* [10] stating that inputs of higher security levels must not affect the outputs of lower security levels at all.

There exists various means to statically check whether a program is non-interferent. *Type systems* are one of such means. They are mature enough for being included in software development tools, e.g. Jif [16], an extension of the Java programming language with security levels for values and locations. The Jif compiler statically checks the validity of security annotations, thus ensuring that a program compiles only if it is non-interferent. However, non-interference is often a too strong property for realistic programs. Hence Jif also contains means to *declassify* information; to assign to it a lower security level than would be mandated by the type system.

For programs containing cryptographic operations, non-interference is obviously not the correct formalization for secure information flow. Indeed, a ciphertext depends on the plaintext, and an attacker with an unbounded amount of resources would even be able to find the plaintext from a ciphertext, but nevertheless one would like to consider a program releasing encrypted secrets as secure, because a reasonable attacker cannot deduce anything about the secret inputs from the ciphertexts it sees. A notion of *computational non-interference* is suitable here, demanding only computational, not absolute independence of a program's secret inputs and public outputs. There also exist type systems enforcing such a property for programs containing cryptographic operations; they are more lax than the type systems for non-interference.

The type systems for computational non-interference have not yet found their way to development tools. Some of the simpler systems (e.g. the type system of [19]) can be readily modeled in Jif, using its declassification mechanism to lower the security level of ciphertexts. But these simple type systems put serious restrictions on the manipulation of cryptographic material — they do not consider the encryption keys as the first-class data that can be manipulated by programs. In this paper we investigate how a type system enforcing computationally secure information flow, not putting restrictions on the programs [14], can be modeled in Jif.

Again, we have to use the declassification mechanism during the encryption, but we also have to track the flow of keys, and consider what happens when information from several sources is combined. As a result of this paper we show that the type system [14] can be modeled in Jif with an exception of encryption cycles.

2 Related Work

Static analysis for verifying the programs for secure information flow was started by Denning [8], a suitable type system for a simple imperative language, together with solid semantical underpinnings were first proposed by Volpano et al. [21]. The extensions for the Java type system to track the information flow — the *decentralized label model* — were proposed by Myers [16], subsequently implemented in the Jif compiler. Later, similar extension has appeared for Caml [17]. The enforcement of information flow policies can also be added to a programming language through a suitable library / sublanguage [15]. Among those extensions, Jif is certainly the most mature one, having been used for larger projects, e.g. an e-voting application [6] and a secure e-mail client [11]. A semi-recent overview of language-based information flow security and the static methods to enforce it is given by Sabelfeld and Myers [18].

Language-based analysis of computations containing cryptographic primitives was started by Abadi et al. [2, 1] and extended to a full imperative programming language by Laud [12, 13]. In these papers, a data flow analysis was proposed. A type system similar in expressiveness was proposed by Laud and Vene [14]. Later, several type systems for computationally secure information flow have been proposed. Smith and Alpizar [19] gave a simple type system that handled encryption (but also decryption) with a single key that could not be accessed otherwise. Courant et al. [7] modified this type system to handle deterministic encryption (i.e. pseudorandom permutations). Askarov et al. [3] have proposed an abstraction of the encryption primitive and devised a type system for it. The type system, similarly to [14], does not constrain the operations performed with cryptographic data. Fournet and Rezk [9] give a similar type system to a language containing public-key encryption and signatures and use this language to soundly implement information flow policies for accessing shared memory. Vaughan and Zdancewic [20] have introduced an *information-packing* primitive (which provides confidentiality) and integrated it into the decentralized label model. We are not aware of any previous attempts to model these type systems in Jif.

3 Details of the Jif Type System

Jif is a security-typed language that extends Java programming language by adding types to define different security policies. These policies are expressed by using a decentralized label model. A principal is an entity that can express security requirements.

A principal can also delegate authority to another principal. If principal p delegates authority to principal q then the principal q is said to act for principal p, written $q \succeq p$. Principal $\top$ may act for all other principals whereas principal $\bot$ permits all other principals to act for it. Jif also allows to form principal conjunctions and disjunctions. Principal $p\&q$, a conjunction of principals p and q, allows $p\&q$ to act for both q and p. Principal "p, q", a disjunction of principals p and q, permits both q and p to act for it.

The label consists of two policies: the confidentiality and integrity policy. The confidentiality policy allows the owner to specify which principals may read certain information. It is formed by conjunctions and disjunctions of reader policies $o \to r$, where o is the owner of the policy and r is the specified reader. The policy says that the owner o allows certain information to be read

by principal q only if q is the owner o or if q can act for r. Also, reader policy conjunctions and disjunctions can be formed. The conjunction $c \sqcup d$ enforces both c and d. Thus the information may be read by principal $c \sqcup d$ only if both c and d allow it. The disjunction $c \sqcap d$ says that the information may be read by principal $c \sqcap d$ if one of the two c and d allow it. The least restrictive confidentiality policy is $\bot \rightarrow \bot$, as all principals know that the information may be read by all principals. The most restrictive one is $\top \rightarrow \top$, because only $\top$ can read the information.

The integrity policies are defined dually to confidentiality policies. These policies allow the owner to specify who may have influenced certain information. The writer policy $o \leftarrow w$ says that according to owner o principal q may have influenced the information only if q is the owner o, or may act for w. Similarly to reader policies conjunctions and disjunctions of writer policies can be formed. The most restrictive writer policy is $\bot \leftarrow \bot$, as all principals know that any principal may have influenced the information. The least restrictive writer policy is $\top \leftarrow \top$, because all principals know that only $\top$ may have influenced the information.

Labels are pairs that consist of a confidentiality policy and integrity policy, written $\{c; d\}$, where c is a confidentiality policy and d is an integrity policy. Each variable (whether local, or instance, or class) has a label associated with it; the label of a variable `x` is denoted by `{x}`. These labels must generally be specified by the programmer (otherwise a default label is used), but for local variables, the Jif compiler may be able to infer them automatically. The label of an expression is the join of the labels of its subexpressions; the Jif compiler makes sure that information with a more restrictive label is never stored in a variable with a less restrictive label. To allow such stores, the programmer may *declassify* or *endorse* some expression, relaxing either its associated confidentiality or integrity policy. Whenever some principal's policy is relaxed this way, the Jif compiler verifies that the code currently executing has the *authority* of this principal.

The variables are not the only objects to have labels. Each program point has a label (usually inferred) characterizing the level of the information that has affected the program's control flow whenever it reaches this program point. Only those variables can be assigned to whose label is at least as restrictive as the label of the program point containing the assignment. Jif also relies on explicit label annotations while arguing about interprocedural flow of information. The programmer has to annotate the arguments of methods, as well as their return values. Similarly, all exceptions that a method can throw have labels characterizing the flow of information that lead to the throwing of this exception. To prevent implicit flows through assignments made inside the method, the method also has the *begin-label* that lower-bounds the label of the program points where this method is called from.

The methods in the Jif language may be label-polymorphic. For such methods, the programmer can restrict the polymorphism by declaring that the labels of parameters must satisfy certain constraints. These constraints are verified by the Jif compiler at each program point where this method may be called; these constraints can also be relied on inside the method body. Jif also allows classes to be parameterized by labels or principals; this mechanism is similar to the generics of Java. Each time an object is created, these label and principal parameters are instantiated.

The above description of Jif covers only the features our implementation uses. For a complete overview we refer to [5].

4 Laud-Vene Type System

The type system [14] has been designed to check for the computational security of information flow (CSIF) in a simple imperative language (the WHILE-language) where the set of operations has been extended with key generation (nullary) and encryption (binary). In defining CSIF, let

us assume that each variable of the program has either *high* or *low* security level. A program has CSIF if the initial values of its high-security variables are *computationally independent* of the final values of its low-security variables. Two random variables $\mathbf{X}$ and $\mathbf{Y}$ are computationally independent if no efficient algorithm, given the values of $\mathbf{X}$ and $\mathbf{Y}$ can tell whether these values come from the same experiment (a run of the program) or different experiments.

A *typing* assigns a type to each variable. A *type system* puts restrictions on possible typings. These restrictions make sure that the information does not flow from high-typed (or -security) variables to low-typed variables, unless it is encrypted inbetween. The set of types is quite rich, because there are various kinds of secrets whose flow has to be kept track of. Besides the secret inputs we also have to track the various keys. Let $\mathcal{G}$ be the set of program points where keys are generated. Let the set of *basic secrets* be $\mathcal{T}_0 = \{\mathsf{h}\} \cup \mathcal{G}$ where h denotes the secret inputs. The elements of $\mathcal{T}_0$ denote the various kinds of data that need protection. The set of *encrypted secrets* is $\mathcal{T}_1 = \{\{t\}_N \,|\, t \in \mathcal{T}_0, N \subseteq \mathcal{G}\}$. The type $\{t\}_N$ denotes the amount of information contained in a basic secret t that has been protected (through encryption) by at least one key from each of the key generation statements in N. The encrypted secrets are ordered by the amount of information still present in them: we have $\{t\}_N \leq \{t'\}_{N'}$ if $t = t'$ and $N \supseteq N'$. An *information type* of a variable is basically a set of encrypted secrets, meaning that the value of the variable can depend from only these encrypted secrets (the information type of a variable depending only on public data is the empty set). Given an information type $T \subseteq \mathcal{T}_1$, it may be possible to *simplify* it. First, we may drop from T all its non-maximal elements. Second, if $\{t\}_{N\cup\{i\}} \in T$ and $i \in T$ (here i denotes $\{i\}_\emptyset$) for some $i \in \mathcal{G}$ then we may replace $\{t\}_{N\cup\{i\}}$ with $\{t\}_N$. This simplification corresponds to the ability to decrypt with known keys.

Besides the information type, each variable also has the *usage type*. It is either Key_N, denoting that the variable can be used as a key and its value was generated in one of the program points in $N \subseteq \mathcal{G}$, or it is Data, denoting that the variable is not a key.

Let γ be a typing; it assigns a pair $\langle T, U\rangle$ of information and usage types to each variable. For some variable x, let $\gamma_{\mathsf{Data}}(x)$ be its information type if it is considered to be non-key: if $\gamma(x) = \langle T, \mathsf{Data}\rangle$ then $\gamma_{\mathsf{Data}}(x) = T$, and if $\gamma(x) = \langle T, \mathsf{Key}_N\rangle$ then $\gamma_{\mathsf{Data}}(x) = T \cup N$ (possibly simplified). Each statement $x := o(x_1, \ldots, x_k)$ in the program imposes certain constraints on γ. For a "non-special" o we just require that $\pi_2(\gamma(x)) = \mathsf{Data}$, $\pi_1(\gamma(x)) \geq \gamma_{\mathsf{Data}}(x_i)$ and $\pi_1(\gamma(x)) \geq T_{\mathsf{pc}}$ where T_{pc} is the current *program counter label* (the least upper bound of all $\gamma_{\mathsf{Data}}(b)$, such the execution of this program point is controlled by an *if*- or *while*-statement whose guard is b).

For a key generation statement $x := \mathcal{G}en()$ at the program point i we have the constraints $\pi_2(x) = \mathsf{Key}_N$ for some $N \supseteq \{i\}$ and $\pi_1(x) \geq T_{\mathsf{pc}}$. An assignment $x := y$ can be typed as in the previous paragraph, but if $\pi_2(\gamma(y)) = \mathsf{Key}_N$ then we may also choose to satisfy the constraints $\pi_2(\gamma(x)) = \mathsf{Key}_{N'}$ where $N \subseteq N'$, $\pi_1(\gamma(x)) \geq \pi_1(\gamma(y))$ and $\pi_1(\gamma(x)) \geq T_{\mathsf{pc}}$. Finally, for a statement $x := \mathcal{E}nc(k, y)$, where $\pi_2(\gamma(k)) = \mathsf{Key}_N$ we may also choose to satisfy the constraints $\pi_2(\gamma(x)) = \mathsf{Data}$, $\pi_1(\gamma(x)) \geq T_{\mathsf{pc}}$ and $\pi_1(\gamma(x)) \geq \{\{t\}_{M\cup\{i\}} \,|\, \{t\}_M \in \pi_1(\gamma(k)) \cup \pi_1(\gamma(y)), i \in N\}$.

As we mentioned before, the type of a secret input variable must be at least $\langle\{\mathsf{h}\}, \mathsf{Data}\rangle$. Similarly, the *least upper bound* of $\gamma_{\mathsf{Data}}(y)$ for all public outputs y must not be $\{\mathsf{h}\}$ or higher. If these conditions and all constraints from previous two paragraphs hold, then the program has computationally secure information flow. Note, however, that the security of the encryption scheme under *key-dependent messages* [4] is necessary for this to hold. In [14] the simplification rules for sets of encrypted secrets were more complex, allowing the type system to detect *encryption cycles* and deem them insecure. These rules are probably too complex to be modeled within the Jif type system. Still, even without considering encryption cycles, the modeling task remains interesting.

5 The Principles of Modeling

There is a special `Key` class defined so that only keys generated by that class can be used for encryption. The class implements the methods for generating new keys (realized in the constructor) and encryption. As we also want to allow one to access the "actual value" (as a bit-string) of the key, there will also be a method that returns it.

In [14], a type consisted of an information type and a usage type. We will obviously use Jif's labels to track the information types of values. A value has the usage type Key_N only if it is an object of class `Key`, otherwise it has the usage type `Data`. The class `Key` is parameterized with something that allows us to track the set N of possible generation points of that key.

The public and private inputs are modeled by having a fixed principal H that is allowed to read private data (the public data can be read by $\bot$). For each key generation point $g \in \mathcal{G}$ we also introduce a principal P that is allowed to read the keys generated at this point. Besides P, we also introduce the principal $\overline{P}$ that certainly *does not know* the keys generated at the point g. On can form conjunctions and disjunctions of these principals. For example, $P_1 \& \overline{P_2}$ is a principal that knows the keys generated at g_1, but certainly does not know the keys generated at g_2. The principal $P \& \overline{P}$ does not exist — i.e. it is considered to be equivalent to $\top$ and it may not occur in the program text.

If information in variable x can be read by someone who acts for the principal X and the key k was generated at g (represented by the principals P and $\overline{P}$), then the ciphertext $\mathcal{E}nc(k, x)$ may be read by someone who acts for the disjunction $X, \overline{P}$. The necessary declassification from X to $X, \overline{P}$ is performed by the encryption method of the `Key`-class. Similarly, if a key k_1 generated at g_1 is encrypted with a key k_2 generated at g_2 then the result can be read by $P_1, \overline{P_2}$. A pair consisting of $\mathcal{E}nc(k_2, k_1)$ and $\mathcal{E}nc(k_3, k_2)$ (where k_3 is generated at g_3) can be read by the principal $((P_1, \overline{P_2}) \& (P_2, \overline{P_3}))$. This means that the result may be read by either a principal who acts for $P_1 \& P_2$ (the principal who may already read both plaintexts), or principal who acts for $\overline{P_2} \& \overline{P_3}$ (the principal who surely does not know the keys to decrypt both ciphertexts), or principal who acts for $P_1 \& \overline{P_3}$ (the principal who knows the key k_1 but is unable to decrypt $\mathcal{E}nc(k_3, k_2)$. The fourth possibility $P_2 \& \overline{P_2}$ equals $\top$.

In [14], a program has computationally secure information flow if the least upper bound of the types of its public variables is not `h` or greater. While modeling this type system in Jif, the programmer has to explicitly state that least upper bound. The public variables may be read by a principal of the form $P_{i_1} \& \ldots \& P_{i_k} \& \overline{P_{j_1}} \& \ldots \& \overline{P_{j_l}}$ with $\{i_1, \ldots, i_k\} \cap \{j_1, \ldots, j_l\} = \emptyset$. The use of this label is more clearly explained in Sec. 7.

6 The `Key`-class

The class for encryption keys, given in Fig. 1, contains methods for key generation, encryption and for returning the value of the key. In the declaration of a variable that is of type `Key`, written `Key[l1,l2]{l} k`, the label `l1` must be of the form `{p->P1&...&Pn;p<-*}` and `l2` of the form `{p->NotP1&...&NotPn;p<-*}` for some principal `p` that has the authority to execute the declaration of `k`. In the syntax of Jif, `*` denotes $\top$. Jif does not check that the two labels are of the correct form (first one containing $P_1 \& \cdots \& P_n$ and the second one $\overline{P_1} \& \cdots \& \overline{P_n}$), but this syntactic check could be easily included somewhere in the development environment. The variable `k` may contain keys generated at one of the points $g_1, \ldots, g_n$. The covariance of the label parameters is used in the subtyping decisions; this allows the assignments of the form `k1 = k2;` where the keys pointed to by `k1` may have been created in at least as many program points as the keys pointed to by `k2`.

```
1  import value.Value;
2  import javax.crypto.*;
3  import javax.crypto.spec.*;
4
5  public class Key[covariant label l1, covariant label l2] {
6    final byte[]{this} key;
7
8    Key() {
9      this.key = gen();
10   }
11
12   String{pt meet l2 ; p<-*}
13     encrypt{this}(principal p, String pt)
14       where {pt}<={p->*;p<-*},{this}<={p->*;p<-*},caller(p)
15   {
16     String r = encAES(key,pt);
17     return declassify(r, {pt meet l2 ; p<-*});
18   }
19
20   String{this ; l1} value() {
21     String{this ; l1} keyValue = Value.bytesToString(key);
22     return keyValue;
23   }
24
25   private static byte[] gen() {  ...  }
26
27   private static String encAES(byte[] raw,String pt) {  ...  }
28 }
```

Figure 1: The class `Key`

The instance method `encrypt` takes a principal `p` and plaintext `pt` as arguments. The principal `p` must have authorized the call to `encrypt`, as stated by the precondition `caller(p)`. This is needed because the method uses declassification to properly model the weakening of the restrictions on information flow according to type system [14] and thus the authority of the concerned principal is required. The other two restrictions `{pt}<={p->*;p<-*}` and `{this}<={p->*;p<-*}` assure that labels `{pt}` and `{this}` for the plaintext and the key only contain policies of principal `p`. See Sec. 9 for a discussion of this restriction. As the actual encryption method `encAES` uses both `key` and `pt` then the label of the result `r` is a conjunction of the two labels: `{this;pt}`. The label of the ciphertext `r` is then declassified to also allow the text to be read by the principal who surely does not know the key used for encryption. Thus the new label of the ciphertext is a disjunction `pt meet l2`. As the label `{r}` only contains the policy of the principal `p`, there exists sufficient authority to perform the declassification.

The `Key`-class also contains the method `value` for returning the actual value of the key. The method will just convert the array of bytes `key` to a string and restrict its label. This restriction is manifested as the label of the return value of the `value`-method.

If a program makes use of the class `Key` and does not contain any declassification statements

```
1  public static final void main{p<-*}(principal{p<-*} p, String args[])
2      throws IllegalArgumentException
3        where caller(p)
4  {
5
6    final label L = new label {p->NotP1; p<-*};
7    PrintStream[{*L}] out = ...
8
9    Key[{p->P1;p<-*}, {p->NotP1;p<-*}] k;
10   k = new Key[{p->P1;p<-*}, {p->NotP1;p<-*}]();
11
12   String{p->H;p<-*} pt = "Plaintext";
13   String x = k.encrypt(p, pt);
14   out.println("x: " + x);
15 }
```

Figure 2: Example 1

outside of this class, then Jif's type system puts "the same" restrictions on it as the type system of Laud and Vene. We believe that this statement could be formalized as a theorem; however we do not currently intend to do so. It would require a formalization of the Jif type system to an extent that we are not aware of having been done. Also, our goal has been "similarity" of the behavior of type systems, not their total coincidence. Indeed, we have already seen a difference (the encryption cycles) between the two type systems. The next section shows that there are other differences as well.

7 Examples

Fig. 2 presents the most basic example of using the `Key`-class. We generate a key (the two principals associated with this key generation statement are `P1` and `NotP1`), use it to encrypt a secret (denoted by having the confidentiality policy `p->H`) plaintext and output a ciphertext. The program is secure according to the type system of [14] and the Jif compiler also accepts it.

Similarly to Java, Jif starts the execution of the program from a method named `main`, with the correct signature. In Jif's case, this signature also includes the principal `p`, under whose authority the program is executed. Jif's standard library includes the labeled versions of input and output streams; in our example, `out` is an output stream that can print values whose labels are no more restrictive than `L`. By defining the label `L`, we are stating the least upper bound of the labels of the public variables, as required in the end of Sec. 5. Jif does not verify that `L` is of the shape required in Sec. 5, but this simple syntactic check could be embedded elsewhere.

But if in addition to ciphertext `x` the value of the key `k` is also output (as shown in Fig. 3, where the method signature is no longer shown), then Jif rejects the program because the label of `k.value()` contains the policy `{p->P1}` which is not less or equal to `L`. We cannot add this policy to `L` (although the Jif compiler would not complain) because that would violate the conditions put on `L` in Sec. 5.

In the example in Fig. 4 two keys `k1` and `k2` are defined. The first is used to encrypt the plaintext, while the second is used to encrypt the first key. Both ciphertexts are output. The Jif compiler accepts this program, because the first ciphertext may be read by `NotP1` (and also

```
{
  final label L = new label {p->NotP1; p<-*};
  PrintStream[{*L}] out = ...

  Key[{p->P1;p<-*}, {p->NotP1;p<-*}] k;
  k = new Key[{p->P1;p<-*}, {p->NotP1;p<-*}]();

  String{p->H;p<-*} pt = "Plaintext";
  String x = k.encrypt(p, pt);
  out.println("x: " + x);
  out.println("k: " + k.value());
}
```

Figure 3: Example 2

```
{
  final label L = new label {p->NotP1&NotP2; p<-*};
  PrintStream[{*L}] out = ...

  Key[{p->P1;p<-*}, {p->NotP1;p<-*}] k1;
  k1 = new Key[{p->P1;p<-*}, {p->NotP1;p<-*}]();

  Key[{p->P2;p<-*}, {p->NotP2;p<-*}] k2;
  k2 = new Key[{p->P2;p<-*}, {p->NotP2;p<-*}]();

  String{p->H;p<-*} pt = "Plaintext";

  String x1 = k1.encrypt(p, pt);
  String x2 = k2.encrypt(p, k1.value());

  out.println("x1: " + x1 + " x2: " + x2);
}
```

Figure 4: Example 3

by `H`), while the second ciphertext may be read by `NotP2` (and also by `P1`). The label `L` is of the correct form.

Fig. 5 demonstrates double encryption: here `x2` is a ciphertext that may be read by each of the principals `H`, `NotP1` and `NotP2`. Hence we may output `x2` and also one of the keys (chosen statically). Note the value of the label `L` — it states that the key(s) generated at the program point g_1 are public.

The key that is used for encryption may also depend on other values as shown in figure 6. The value of the key `k3` depends on some other (public) value. The labels `l1` and `l2` in the declaration of `k3` have to contain both `P1`/`NotP1` and `P2`/`NotP2`, otherwise the assignment to `k3` is not allowed because of incompatible types. After encrypting the plaintext `pt` with the key `k3` the ciphertext `x` may be read by principal `NotP1&NotP2` (but not just `NotP1` or `NotP2`). Still,

```
{
  final label L = new label {p->P1&NotP2; p<-*};
  PrintStream[{*L}] out = ...

  Key[{p->P1;p<-*}, {p->NotP1;p<-*}] k1;
  k1 = new Key[{p->P1;p<-*}, {p->NotP1;p<-*}]();

  Key[{p->P2;p<-*}, {p->NotP2;p<-*}] k2;
  k2 = new Key[{p->P2;p<-*}, {p->NotP2;p<-*}]();

  String{p->H;p<-*} pt = "Plaintext";

  String x1 = k1.encrypt(p, pt);
  String x2 = k2.encrypt(p, x1);

  out.println("x2: " + x2 + " k1: " + k1.value());
}
```

Figure 5: Example 4

```
{
  final label L = new label {p->NotP1&NotP2; p<-*};
  PrintStream[{*L}] out = ...

  Key[{p->P1;p<-*}, {p->NotP1;p<-*}] k1;
  k1 = new Key[{p->P1;p<-*}, {p->NotP1;p<-*}]();

  Key[{p->P2;p<-*}, {p->NotP2;p<-*}] k2;
  k2 = new Key[{p->P2;p<-*}, {p->NotP2;p<-*}]();

  Key[{p->P1&P2;p<-*}, {p->NotP1&NotP2;p<-*}] k3;
  k3 = (..) ? k1 : k2;

  String{p->H;p<-*} pt = "Plaintext";
  String x = k3.encrypt(p, pt);
  out.println("x: " + x);
}
```

Figure 6: Example 5

the label `L` allows us to output `x`.

The example in Fig. 7 differs from the previous one only in the confidentiality policy on the information from which there is an implicit flow to `k3`. That policy is now `{p->P3}`, instead of `{}`, and that is also the confidentiality policy of `k3` itself. According to the type system of [14], the ciphertext `x` can now be read by either of the principals `H&P3` and `NotP1&NotP2`. Hence the program is still secure and it should suffice to define `L` as in Fig. 7.

```
{
  final label L = new label {p->NotP1&NotP2; p<-*};
  PrintStream[{*L}] out = ...

  Key[{p->P1;p<-*}, {p->NotP1;p<-*}] k1;
  k1 = new Key[{p->P1;p<-*}, {p->NotP1;p<-*}]();

  Key[{p->P2;p<-*}, {p->NotP2;p<-*}] k2;
  k2 = new Key[{p->P2;p<-*}, {p->NotP2;p<-*}]();

  Key[{p->P3;p<-*}, {p->NotP3;p<-*}] k;
  k = new Key[{p->P3;p<-*}, {p->NotP3;p<-*}]();

  Key[{p->P1&P2;p<-*}, {p->NotP1&NotP2;p<-*}] k3;
  k3 = (k.value()) ? k1 : k2;

  String{p->H;p<-*} pt = "Plaintext";
  String x = k3.encrypt(p, pt);
  out.println("x: " + x);
}
```

Figure 7: Example 6

Jif, however, acts differently. It considers there to be information flow from the target of the method call (`k3`) to the result of the method call (`x`). Hence, according to Jif, `x` can be read by either of the principals `H&P3` or `NotP1&NotP2&P3`. The program in Fig. 7 does not compile. We could add `P3` to `L`, thereby making it compile again, but then we could not output any ciphertexts created with the key `k` (this is still allowed in [14]).

Such an assumption by Jif cannot probably be overcome, as long as `encrypt` is an instance method and the occurrences of declassification are constrained to be inside the class `Key`. On the other hand, this assumption is also not a weakness of Jif, because it is necessary each time the target of the call is `nil`. The language used in [14] does not allow the possibility of a key being undefined.

8 Static Method for Encryption

The assumption made by Jif on the information flow from that target of a method call to the result of that call could be overcome if we implement encryption as a static method, as shown in Fig. 8. In addition to a principal `p` and a plaintext `pt`, the static encryption method takes a key `k` as an argument. The method saves `k.key` in `kv`, but before dereferencing `k` it declassifies it, such that the possible `NullPointerException` that may be thrown does not have a too restrictive label. After saving `k.key` in `kv`, the static method works the same way as the instance method.

This static encryption method is typed almost like in the type system [14] with the exception of handling the `NullPointerException`. Using this method instead of the instance method would allow us to compile the example in Fig. 7. But as handling the `NullPointerException` might be cumbersome then the instance encryption method is used where implicit information

```
1  static String{pt meet l2; k meet l2; p<-*}
2    encrypt_s{p<-*}(principal p,Key[l1,l2] k,String pt)
3      :{pt meet l2; k meet l2; p<-*}
4        throws NullPointerException
5          where {pt} <= {p->*;p<-*}, {k} <= {p->*;p<-*},
6            caller(p)
7  {
8    byte [] kv = declassify(k,{k meet l2; p<-*}).key;
9    String r = encAES(kv,pt);
10   return declassify(r, {pt meet l2 ;k meet l2;p<-*});
11 }
```

Figure 8: Static encryption method

flow does not occur.

9 Conclusions and Discussion

We have shown that existing tools for secure programming (in particular, Jif) are also well-suited for making sure that programs have computationally secure information flow. The next logical step would be the extension of Jif to include encryption in its policies. So far we have not really made use of the decentralized label model; in fact, we have actively tried to work around it by stating that there is a single principal `p` whose policies we are concerned with. Indeed, Laud and Vene [14] also do not consider multiple principals. Hence the extensions of this work also have to consider important theoretical problems, e.g. the integrity of keys.

References

[1] Martín Abadi and Jan Jürjens. Formal Eavesdropping and Its Computational Interpretation. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, volume 2215 of *LNCS*, pages 82–94, Sendai, Japan, October 2001. Springer-Verlag.

[2] Martín Abadi and Phillip Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *International Conference IFIP TCS 2000*, volume 1872 of *LNCS*, pages 3–22, Sendai, Japan, August 2000. Springer-Verlag.

[3] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-Masked flows. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 353–369. Springer, 2006.

[4] John Black, Phillip Rogaway, and Thomas Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In Kaisa Nyberg and Howard M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2002.

[5] Stephen Chong, Andrew C. Myers, K. Vikram, and Lantian Zheng. *Jif Reference Manual*, April 2008.

[6] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society, 2008.

[7] Judicaël Courant, Cristian Ene, and Yassine Lakhnech. Computationally sound typing for non-interference: The case of deterministic encryption. In Vikraman Arvind and Sanjiva Prasad, editors, *FSTTCS*, volume 4855 of *Lecture Notes in Computer Science*, pages 364–375. Springer, 2007.

[8] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[9] Cédric Fournet and Tamara Rezk. Cryptographically Sound Implementations for Typed Information-Flow Security. In *POPL 2008, Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, January 2008. ACM Press.

[10] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, April 1982. IEEE Computer Society Press.

[11] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick Drew McDaniel. From languages to systems: Understanding practical application development in security-typed languages. In *ACSAC*, pages 153–164. IEEE Computer Society, 2006.

[12] Peeter Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In David Sands, editor, *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001*, volume 2028 of *LNCS*, pages 77–91, Genova, Italy, April 2001. Springer-Verlag.

[13] Peeter Laud. Handling Encryption in Analyses for Secure Information Flow. In Pierpaolo Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003*, volume 2618 of *LNCS*, pages 159–173, Warsaw, Poland, April 2003. Springer-Verlag.

[14] Peeter Laud and Varmo Vene. A Type System for Computationally Secure Information Flow. In Maciej Liśkiewicz and Rüdiger Reischuk, editors, *15th International Symposium on Fundamentals of Computation Theory (FCT) 2005*, volume 3623 of *LNCS*, pages 365–377, Lübeck, Germany, August 2005. Springer-Verlag.

[15] Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *CSFW*, page 16. IEEE Computer Society, 2006.

[16] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 1999. ACM Press.

[17] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.

[18] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[19] Geoffrey Smith and Rafael Alpízar. Secure Information Flow with Random Assignment and Encryption. In *4th ACM Workshop on Formal Methods in Security Engineering*, pages 33–43, 2006.

[20] Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In Birgit Pfitzmann and Patrick McDaniel, editors, *IEEE Symposium on Security and Privacy*, pages 192–206. IEEE Computer Society, 2007.

[21] Dennis M. Volpano, Geoffrey Smith, and Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.