Refactoring Preserves Security

Florian Kammüller

Department of Computer Science
Middlesex University London
f.kammueller@mdx.ac.uk

Abstract. Refactoring allows changing a program without changing its behaviour from an observer's point of view. To what extent does this invariant of behaviour also preserve security? We show that a program remains secure under refactoring. As a foundation, we use the Decentralized Label Model (DLM) for specifying secure information flows of programs and transition system models for their observable behaviour. On this basis, we provide a bisimulation based formal definition of refactoring and show its correspondence to the formal notion of information flow security (noninterference). This permits us to show security of refactoring patterns that have already been practically explored.

1 Introduction

In distributed systems, we are interested in specifying and verifying security[1] of data values. Usually, values are labelled to indicate their confidentiality level. The labels express the owners and the readers of a value. However, a value in itself is not security critical: everyone may know the value 42 but in association with a specific usage it can become a secret, for example, if 42 is the PIN code of an online banking account. Security models, like the decentralized label model (DLM) [10] we use for the presentation of our framework, assign security labels to the input and output variables (or channels) of a computer program. This enables the analysis of flows of values through this program judging whether certain computations violate the specified secure information flows. This analysis is called Information Flow Control (IFC) [2]. Besides the easy to spot direct flows, e.g., by assignment or parameter passing, there are more subtle cases where "the information flow is disguised as control flow" [1], like in the classical if-then-else example,where the control flow copies the confidential bit $\mathtt{x}_H$ to the public $\mathtt{y}_L$.

$$\mathtt{if\ x}_H\ \mathtt{=\ 1\ then\ y}_L\ \mathtt{:=\ 1\ else\ y}_L\ \mathtt{:=\ 0\ end}$$

Refactoring [4,9] is a technique that is applied in order to improve the internal structure of a software artifact to enhance readability of the code, make it more amenable to extensions, and thus support its maintainability. Integrated Development Environments (IDE) like Eclipse support refactoring.

Our contribution is a correspondence theorem between a formal characterisation of refactoring and a formal characterization of noninterference of a program.

[1] for simplicity we concentrate on confidentiality in this paper.

Assuming that a program is initially correctly labeled, i.e., permits only the labeled information flows, then our theorem can be applied to show that a (proper) refactoring of the program code preserves the security of that program. Thereby, this paper provides a formal basis of what has been introduced by examples [5].

In this paper, we first review the concepts of the decentralized label model (DLM) [10] (Section 2). Then, we provide a formal definition of refactoring and information flow security, relating the two by a security preservation theorem (Section 3). As a proof of concept, we finally show how our theoretical framework can be applied. The refactoring mechanism "Extract method" for Java Information Flow (Jif) [5] can now be shown to be security preserving by a simple application of our theoretical framework (Section 4).

2 Decentralized Label Model — DLM

A value in the DLM model [10] always carries the label of the variable it resides in. Values become labeled when they are read from input variables by that variable's label. If the program *writes* this variable, then the old label of the value assigned to this variable is forgotten and the value becomes reassigned with the new label of the destination. This process is called *relabeling.* To preserve security, information may only flow up: the relabeling must respect the security levels in that a value that has label L_0 can be relabeled with label L_1 iff $L_0 \sqsubseteq L_1$. Writing includes assignment of a value to a variable or passing a value as a parameter to a method call but also implicit flows as described in the if-then-else example above.

2.1 Labels

Every value used or computed in a program execution has an associated label which stands for a set of allowed *flows (owner, reader)* from a principal *owner* to a a principal *reader*. There may be a range of permitted flows for a variable, therefore we accumulate all possible flows into labels. A label is a set of *label components* that summarise the allowed flows for a single owner o, i.e., a component (o, R_K) specifies that the owner o permits all readers $r \in R_K$. A label can have a list of label components. The allowed flows of a label are given as the union of all flows of all components of L and all flows (o, r) for all o for which there is *no* component (o, R_K) with $r \in R_K$ in L. The meaning of this addition to the explicitly stated components in a label is: if a principal o is *not* an owner in the label L, then L describes flows (o, r) for every principal r.

To summarise, a label $L = \{o_0 : R_0; \ldots ; o_n : R_n\}$, where $O_L \stackrel{\text{def}}{=} \{o_i \mid i \in 0..n\}$, denotes the set of flows

$$[\![L]\!] \stackrel{\text{def}}{=} \{(o_i, r) \mid o_i \in O_L \wedge r \in R_i\} \cup \{(o, r) \mid o \notin O_L\}.$$

For example, for the label $L_{ex} \stackrel{\text{def}}{=} \{\text{al} : \{\text{eve}\}; \text{bob} : \{\text{al}\}\}$ we have

$$[\![L_{ex}]\!] = \{(\text{al}, \text{al}), (\text{al}, \text{eve}), (\text{bob}, \text{bob}), (\text{bob}, \text{al}), (\text{eve}, \text{eve}), (\text{eve}, \text{al}), (\text{eve}, \text{bob})\}.$$

assuming that al, bob, and eve are all possible principals.

2.2 Label Lattice and Relabeling

Given this interpretation of labels as sets of allowed flows (o, r), the set of labels forms a complete lattice together with the following partial order on labels.

$$L_0 \sqsubseteq L_1 \overset{\text{def}}{=} [\![L_0]\!] \supseteq [\![L_1]\!]$$

The lattice operations join $\sqcup$ and meet $\sqcap$ are defined as follows.

$$\begin{aligned} L_0 \sqcup L_1 &\overset{\text{def}}{=} [\![L_0]\!] \cap [\![L_1]\!] \\ L_0 \sqcap L_1 &\overset{\text{def}}{=} [\![L_0]\!] \cup [\![L_1]\!] \end{aligned}$$

When values flow from one variable with label L_0 to another with label L_1 we call this a relabeling as discussed above; it is allowed if L_1 is equally or more restrictive than L_0, i.e., $L_0 \sqsubseteq L_1$.

The lattice operations *join* ($\sqcup$) and *meet* ($\sqcap$) allow combining labels thus supporting inference of labels for compound expressions.

$$\begin{aligned} \mathit{owners}(L_1 \sqcup L_2) &= \mathit{owners}(L_1) \cup \mathit{owners}(L_2) \\ \mathit{readers}(L_1 \sqcup L_2, O) &= \mathit{readers}(L_1, O) \cap \mathit{readers}(L_2, O) \end{aligned}$$

The dual equations hold for the operation meet ($\sqcap$).

In the following, we assume that all program variables are labelled correctly, i.e., the labels correspond to the actual flows in the programs. In practice, this assumption is enforced by a process of (static) checking.

2.3 Observation and State Transition Model

In the decentralised label model, an observation happens when values are written to output channels (variables) which have a set of readers associated to it. These are the principals who will be able to observe values written to that destination (a channel or variable). The owners assigned to an input variable are the principals whose data was observed in order to obtain that value.

For the system model we follow the classical state transition model mainly used for security modeling, e.g., [6,7,8]. A system is described by its traces of events. Since we consider a programming system, the events are changes of state variables according to inputs, outputs, and computation steps. Each step in the state transition corresponds to a step in the operational semantics of the programming language. We consider deterministic programming languages with no real parallelism, i.e., events happening in different steps lead to traces where "parallel" events are resolved using interleaving. A system trace in our model is a possibly infinite sequence $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$ of maps $s_i : \mathit{Var} \mapsto \mathit{Val}$ from program variables $\mathit{Var} \overset{\text{def}}{=} \{v_0, \ldots, v_n\}$ to their values $\mathit{Val} \overset{\text{def}}{=} \{a_0, \ldots, a_n\}$. In our system model, we assume that each state is reachable from some initial state s_{init}, i.e., $s_{init} \rightarrow^* s_0$. Each variable v_i in the program has a DLM label assigned to it and the transition relation respects the labeling.

3 Security of Refactoring

Let $Var \overset{\text{def}}{=} \{v_0, \ldots, v_n\}$ denote the labeled state variables of program P and Q.[2]

We define a map $\mathcal{L}$ that assigns to each variable its label, i.e., set of components.

$$\mathcal{L} : v_i \mapsto \{(o_j, R_j) \mid o_j \in \mathcal{P} \wedge R_j \subseteq \mathcal{P}\}, j \in 0..m, i \in 0..n$$

The indistinguishability relation $\sim_\alpha$ describes that from an observation point α (which is a label) two states $s_0, s_1 : Var \mapsto Val$ look the same, i.e., variables that are at or below α appear equal in s_0, s_1.

$$s_0 \sim_\alpha s_1 \overset{\text{def}}{=} \text{dom}(s_0) = \text{dom}(s_1) \wedge \forall v \in Var.\ \mathcal{L}(v) \sqsubseteq \alpha \Rightarrow s_0(v) = s_1(v).$$

Indistinguishability is often called "low-equivalence": only variables that are above α may differ in states that are related. Thus an attacker at level α cannot perceive a difference in different program runs that are due to variables labeled with a more restrictive label (higher in the order $\sqsubseteq$).

We use the highest observation point seeing all variables (in terms of $\sqsubseteq$) to express the program equality that defines a refactoring.

Definition 1 (Refactoring). *Let s_0, t_0 be states in P and Q respectively. Let*

$$Obs \overset{\text{def}}{=} \bigsqcup_{i \in 0..n} \mathcal{L}(v_i).$$

Q is a refactoring of P iff
$s_0 \sim_{Obs} t_0$ and $s_0 \to s'_0$ implies $t_0 \to^* t'_0$ and $s'_0 \sim_{Obs} t'_0$ for some t'_0.

For an attacker we can specify a viewpoint in order to quantify his attack powers. For the sake of the generality of the exposition, we assume a very powerful attacker that is a principal $a \in \mathcal{P}$ with observation point $Att \overset{\text{def}}{=} \bigsqcup_{i \in 0..m}(o_i, \{a, o_i\})$. The attacker a is a reader for any owner o_i, i.e., can see data of all owners. The following observation holds for this attacker and for any other choice of an attacker, since we have chosen Obs to be the least upper bound of the label lattice.

Lemma 1.

$$Att \sqsubseteq Obs$$

Lemma 2.

$$s_0 \sim_{Obs} t_0 \Rightarrow s_0 \sim_{Att} t_0$$

Definition 2 (Security (Noninterference)). *Program P is secure for attacker a with viewpoint Att iff $s_0 \sim_{Att} s_1$ and $s_0 \to s'_0$ implies $s_1 \to^* s'_1$ and $s'_0 \sim_{Att} s'_1$ for some s'_1.*

Lemma 3. *The relations $\sim_\alpha$ and 'P refactors to Q' are equivalence relations, i.e., are reflexive, transitive, and symmetric.*

[2] We should consider differently named bijective sets of variables for P and Q since renaming is also a refactoring but for the sake of simplicity we omit it here.

Lemma 4. *Security and refactoring are defined for the one step transition $s_0 \to s_0'$ but they naturally extend to the reflexive transitive closure $s_0 \to^* s_0'$.*

1. *Let P be secure for α. If $s_0 \sim_\alpha s_1$ and $s_0 \to^* s_0'$, then there exists s_1' such that $s_1 \to^* s_1'$ and $s_0' \sim_\alpha s_1'$.*
2. *Let P refactors to Q. If $s_0 \sim_{Obs} t_0$ and $s_0 \to^* s_0'$, then there exists t_0' such that $t_0 \to^* t_0'$ and $s_0' \sim_{Obs} t_0'$.*

Lemma 5. *Let Q be a refactoring of P. For any state s_0 in Q, there is a state t_0 in P with $s_0 \sim_{Obs} t_0$.*

Theorem 1 (Refactoring is secure). *Let Q be a refactoring of P and let P be secure for a. Then Q is also secure for a.*

Proof. Let P be a program that refactors to Q for Obs and let P be secure for attacker a, i.e., the observation point Att. We need to show that for any s_0, s_1 in Q with $s_0 \sim_{Att} s_1$, if $s_0 \to s_0'$ (see arrow (1) in Figure 1) then $s_1 \to^* s_1'$ for some s_1' (see arrow (4) in Figure 1) such that $s_0' \sim_{Att} s_1'$ (d). Lemma 5 shows

$$\begin{tikzcd}
 & t_1 \arrow[rrr, "(3)"] \arrow[dd, dash, "\sim_{Obs}"] & & & t_1' \arrow[dd, dash, "(c)"] \\
t_0 \arrow[ur, dash, "\sim_{Att}"] \arrow[rrr, "(2)"] \arrow[dd, dash, "\sim_{Obs}"] & & & t_0' \arrow[ur, dash, "(b)"] \arrow[dd, dash, "(a)"] & \\
 & s_1 \arrow[rrr, "(4)"] & & & s_1' \\
s_0 \arrow[ur, dash, "\sim_{Att}"'] \arrow[rrr, "(1)"] & & & s_0' \arrow[ur, dash, "(d)"'] &
\end{tikzcd}$$

Fig. 1. Proof structure for Theorem 1

that, because P refactors to Q, we have t_0 and t_1 in P such that $s_0 \sim_{Obs} t_0$ (i) and $s_1 \sim_{Obs} t_1$ (ii). Lemma 2 immediately implies that then also these states are indistinguishable from the observation point of attacker a, i.e., $s_0 \sim_{Att} t_0$ and $s_1 \sim_{Att} t_1$. (see the left of Figure 1). Since indistinguishability is symmetric and transitive according to Lemma 3, we can deduce that $t_0 \sim_{Att} t_1$ (iii).

Since $s_0 \sim_{Obs} t_0$ and $s_0 \to s_0'$, there exists t_0' such that $t_0 \to^* t_0'$ in P and $t_0' \sim_{Obs} s_0'$ because P refactors to Q ((1), (2) and (a) in Figure 1).

Since P is secure according to assumption and $t_0 \sim_{Att} t_1$ (iii), we obtain a t_1' with Lemma 4.1 such that $t_1 \to^* t_1'$ and $t_0' \sim_{Att} t_1'$ ((2), (3), and (b) in Figure 1).

Since P refactors to Q and we have that $t_1 \sim_{Obs} s_1$ (symmetry of $\sim_{Obs}$ and (i)) we obtain a s_1' such that $s_1 \to^* s_1'$ (iv) and $t_1' \sim_{Obs} s_1'$ ((3), (4), and (c) in Figure 1).

Summarizing we get $s_0' \sim_{Obs} t_0'$, $t_0' \sim_{Att} t_1'$, and $t_1' \sim_{Obs} s_1'$ ((a), (b), and (c) in Figure 1), hence with Lemmas 2 and 3, we get $s_0' \sim_{Att} s_1'$ ((d) in Figure 1) and $s_1 \to^* s_1'$ (iv) which finishes the proof.

4 Example

We can show now with our framework that a major refactoring pattern, the “Extract method” refactoring is secure. We first motivate and explain this refactoring and the resulting labeling on an example. With this preparation, we show that the labeling we propose for the refactoring is bisimilar hence secure.

The example is depicted in Figure 2 showing how Refactoring extracts shared code and puts it into a new method. The labels in the example indicate that

```
public class secure_node {
  List<<byte>> {B, {A,B}} skey
  public Integer {B, {A,B}}
    send(Integer m; R r):
      {B, {A,B}}{
k = skey.subList(0,4);
s = k^m;
skey = skey.subList(0,4).clear();
    r.put(s);
  }
  public Integer {B, {A,B}}
    receive(Integer c):
      {B, {A,B}}{
k = skey.subList(0,4);
s = k^c;
skey = skey.subList(0,4).clear();
    return s;
  }
}
```

```
public class secure_node {
  List<<byte>> {B, {A,B}} skey
  public Integer {B,{A,B}}
    send(Integer m; R r):
      {B,{A,B}}{
    s = crypt(m);
    r.put(s);
  }
  public Integer {B, {A,B}}
    receive(Integer c):
      {B, {A,B}}{
    s = crypt(c);
    return s;
  }
 public Integer {B, {A,B}}
   crypt(Integer t):
     {B, {A,B}}{
     k = skey.subList(0,4);
     s = k^t;
     skey =
       skey.subList(0,4).clear();
     return s;
     }
}
```

Fig. 2. Symmetric key encrypted messages can be sent by methods send and receive in the Java class secure_node on the left. Symmetric key encryption and decryption is implemented using exclusive or (^) on a code block of size Integer (4 Bytes). Used key-bits are eliminated with `clear()`. The class can be used for instances to principals Alice and Bob for shared key encryption. Labels are abbreviated for brevity in the code by A for alice and B for bob. Refactoring allows to extract shared code block (“xor”ing an integer and eliminating used key-bits) into new method crypt. Labels are transferred consistently.

the symmetric key `skey` is owned by bob but can be read also by alice: {B, {A, B}}. The entry and exit levels of the method send and receive are bounds for the entry and exit level of the *program counter* (*pc*). A *pc* is a common technique in information flow control originating in Fenton’s Data Mark Machine [3]. The *pc* encodes the highest security level that has been reached in all possible control flows leading to the current control state. The program counter *pc* is derived from the labels of the state variables in the static analysis process. This derivation depends on the static analysis rules of a concrete IFC language, like Jif [10].

Generalising from the example, we need to compare the traces of the program P and the refactored program Q where a common code block has been extracted as depicted in Figure 3. The markers in the figure show the program counters (pc) at the exit and entry points between two parts of the original program and the extracted code block. In the practical application of refactoring to Jif

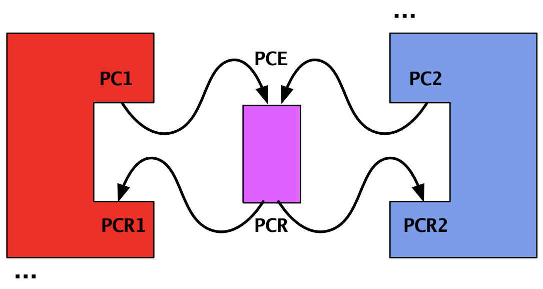

Fig. 3. Refactored program Q with extracted method

programs [5] we provided the following rule for determining the correct labels for refactoring a Jif program by extracting a common code block into a new method illustrated in Figure 3. We chose the entry and exit level of the extracted method such that the entry level is an upper bound to the entry levels of the origin and the exit level is the lower bound of the extracted code [5].

$$\mathrm{PCE} \overset{\mathrm{def}}{=} \mathrm{PC1} \sqcup \mathrm{PC2}$$
$$\mathrm{PCR} \overset{\mathrm{def}}{=} \mathrm{PCR1} \sqcap \mathrm{PCR2}$$

The labels in the example in Figure 2 are trivially consistent with the above rule since

$$\{A, \{A, B\}\} \sqcap \{A, \{A, B\}\} = \{A, \{A, B\}\} = \{A, \{A, B\}\} \sqcup \{A, \{A, B\}\}.$$

To justify the security of this rule now in the current framework, we compare the traces of program P with those of the refactored program Q. Let t_P be a trace of P and s_P be a state in that trace corresponding to the program point before the code block to be extracted. Then there is a trace t_Q of Q with an indistinguishable state s_Q before the call to the extracted method, i.e., $s_P \sim_{Obs} s_Q$. Let, in t_P the next state be s'_P, i.e., $t_P = \langle \ldots s_P \rightarrow s'_P \ldots \rangle$. The entry level of the extracted code in Q is PCE $=$ PC1 $\sqcup$ PC2 and the *pc* in the current state s_Q of t_Q is PCi $\sqsubseteq$ PCE for $i \in 1, 2$. Therefore, the execution of Q can proceed and $s_Q \rightarrow^* s'_Q$ with $s'_P \sim_{Obs} s'_Q$ (possibly more than one step is necessary due to the method call of the extracted method). The important point is that the choice of the entry levels permits the same execution paths in both programs P and Q. A similar argument shows that the same execution paths are permitted for P and Q at the exit point of the extracted method. Therefore, the programs P and Q are bisimilar with the chosen definition of PCE and PCR, i.e., according to Definition 1 they are a refactoring.

Using Theorem 1, we can thus immediately conclude that the program Q, that is refactored from P by Extract method, is secure if P is.

5 Conclusions

Refactoring [4,9] is a technique of much practical value to software engineering increasing the quality of program code while preserving properties. Therefore, different techniques to improve the quality can be applied and good features preserved.

Security is a difficult property to deal with. Information Flow Control with DLM is a technique operating at the program code level that enables giving precise specification of security. However, DLM is difficult to use for the common programmer. We propose a process of security refactoring, in which program code labelled according to a security policy by a team of programmers and security experts can then subsequently be improved by common programmers *without changing the specified security properties.*

In this paper, we have provided the theoretical foundation for this process.

References

1. G. Boudol and I. Castellani. Noninterference for concurrent programs. *ICALP'01.* LNCS **2076** Springer, 2001.
2. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 1977.
3. J. S. Fenton. *Information protection systems.* PhD thesis, Univ. Cambridge, 1973.
4. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 2004.
5. S. Helke, F. Kammüller, and C. W. Probst. Secure refactoring with java information flow. *Data Privacy Management, DPM'15.* LNCS **9481**, Spinger 2015.
6. H. Mantel. On the composition of secure systems. *Security and Privacy*, 2002.
7. H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. *IEEE CSF*, 2011.
8. J. Mclean. A general theory of composition for trace sets closed under selective interleaving functions. *Security and Privacy*, 1994.
9. T. Mens and T. Tourvé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.
10. A. C. Myers and B. Liskov. A decentralized model for information flow control. *ACM symposium on Operating systems principles, SOSP '97*, 1997.