Combining Type-Based Analysis and Model Checking for Finding Counterexamples against Non-Interference

Hiroshi Unno
University of Tokyo
uhiro@yl.is.s.u-tokyo.ac.jp

Naoki Kobayashi
Tohoku University
koba@ecei.tohoku.ac.jp

Akinori Yonezawa
University of Tokyo
yonezawa@yl.is.s.u-tokyo.ac.jp

Abstract

Type systems for secure information flow are useful for efficiently checking that programs have secure information flow. They are, however, conservative, so that they often reject safe programs as ill-typed. Accordingly, users have to check whether the rejected programs indeed have insecure flows. To remedy this problem, we propose a method for automatically finding a counterexample of secure information flow (input states that actually lead to leakage of secret information). Our method is a novel combination of type-based analysis and model checking; Suspicious execution paths (that may cause insecure information flow) are first found by using the result of a type-based information flow analysis, and then a model checker is used to check whether the paths are indeed unsafe. We have formalized and implemented the method. The result of preliminary experiments shows that our method can often find counterexamples faster than a method using a model checker alone.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls; D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Security, Languages, Theory, Verification

Keywords non-interference, type system, model checking

1. Introduction

A number of type systems for secure information flow have recently been proposed [8, 9, 14, 15, 20, 21]. Those type systems guarantee that well-typed programs never leak secret information, so that the problem of checking secure information flow is reduced to the problem of type checking. While the type-based information flow analysis is fast and scalable, it is sometimes too conservative and rejects innocent programs as ill-typed (i.e., false alarms may be generated). For example, let h be a variable holding a high-security value and l a variable that may be read by a low-security observer. The assignment $l := h - h$ does not leak secret information, but it is rejected by type systems for secure information flow [20][1]. In such a case, users have to manually inspect the program to check whether the ill-typed program is indeed unsafe. Such manual inspection is error-prone; users may overlook real violation of secure information flow.

To remedy the problem above, we propose a method for automatically finding input states that actually lead to leakage of secret information. More precisely, our method finds a counterexample against the *non-interference* property [3], a standard criterion for secure information flow. The counterexample is a pair of input states which only differ in the values of high-security variables and lead to output states that differ in the values of low-security variables. For example, a counterexample for the program:

$$\texttt{if}\ h\ \texttt{then}\ l := 1\ \texttt{else}\ l := 0$$

is the pair $(\{h = \texttt{true}, l = 0\}, \{h = \texttt{false}, l = 0\})$. The counterexample can convince the user that the program is indeed unsafe.

Our method is a novel combination of type-based analysis and model checking. The result of a type-based information flow analysis is first used to find suspicious execution paths that may cause insecure information flow, and then a model checker is used to check whether those paths are indeed unsafe (and if so, output a pair of input states that go through the paths). To illustrate our approach, we ask you to consider the following program (which is ill-typed in the usual type system for secure information flow):

$$\begin{aligned} c_{ex1} \quad = \quad & x_0 := h; \\ & \texttt{if}\ b_1\ \texttt{then}\ x_1 := x_0\ \texttt{else}\ \ldots \\ & \qquad\qquad \vdots \\ & \texttt{if}\ b_n\ \texttt{then}\ x_n := x_{n-1}\ \texttt{else}\ \ldots \\ & l := x_n. \end{aligned}$$

Throughout this paper, we assume that h is a high-security variable, and l is a low-security variable. The program above leaks information about h if all the conditional expressions execute the then-branch. Our method consists of the following two main phases.

A. Finding a pair of execution paths that may violate the non-interference property Let ρ_x be the security level of a variable x. From Volpano and Smith's type system, we obtain the following constraints:

$$\rho_h(= \mathbf{H}) \leq \rho_{x_0} \leq \rho_{x_1} \leq \cdots \leq \rho_{x_{n-1}} \leq \rho_{x_n} \leq \rho_l(= \mathbf{L}).$$

From the constraints, an inconsistent constraint $\mathbf{H} \leq \mathbf{L}$ is obtained. Thus, we can conclude that the program is ill-typed. Actually, we

[1] Of course, this example is too simple and unlikely to appear in real programs. In general, however, as the example shows, type-based analysis is not good at analyzing value-dependent behavior of programs.

PLAS'06 June 10, 2006, Ottawa, Ontario, Canada.

can obtain more information from the constraints: Notice that the existence of a constraint $\rho_x \leq \rho_y$ implies that information about the value of x may flow to the value of y. Thus, the above constraints tell us that information about the value stored in h may flow to variable l through variables $x_0, x_1, \ldots, x_{n-1}, x_n$ in this order. From this reasoning, we can find the following execution path as a suspicious execution path that may violate secure information flow:

$$\begin{array}{rcl} \pi_{ex1} & = & x_0 := h; \\ & & \texttt{assume}(b_1);\ x_1 := x_0; \\ & & \vdots \\ & & \texttt{assume}(b_n);\ x_n := x_{n-1}; \\ & & l := x_n. \end{array}$$

Here, the path is represented as a program with `assume` statement. The statement $\texttt{assume}(b_1)$ above means that the then-branch of the original conditional expression is chosen.

Let us consider another ill-typed program, which contains indirect information flow:

$$c_{ex2} \quad = \quad \texttt{if}\ h\ \texttt{then}\ (l := 1)^{pc_1}\ \texttt{else}\ (l := 0)^{pc_2}.$$

From Volpano and Smith's type system, we obtain the following unsatisfiable constraints:

$$\rho_h (= \mathbf{H}) \leq \rho_{pc_1} \leq \rho_l (= \mathbf{L}).$$

(We also get the constraint $\rho_h \leq \rho_{pc_2} \leq \rho_l$, but we only need to focus on a subset of inconsistent constraints.) Here, ρ_{pc} is the security level of information about whether the program point *pc* is executed. From the constraints, we know that information about the value stored in h may flow to pc_1, and then to l, so that the following pair of execution paths (one of which visits pc_1, and the other does not) may violate the non-interference property:

$$\Pi_{ex2} = ((\texttt{assume}(h);\ l := 1), (\texttt{assume}(!h);\ l := 0)).$$

Here, the path in the lefthand side takes the then-branch and the path in the righthand side takes the else-branch.

B. Checking whether the pair of execution paths are unsafe The next phase is to check whether the suspicious execution paths are indeed unsafe. For this purpose, we use the idea of model-checking-based information flow analysis [1, 18]. Let (π_1, π_2) be a pair of suspicious execution paths that may violate the non-interference property. Let π_2' be the program obtained from π_2 by renaming each variable x in π_2 with x'. Then, the two execution paths violate the non-interference property if and only if the program $\pi_1; \pi_2'$ with some initial state satisfying $x = x'$ for every variable other than high-security variables ends up in a state where $l \neq l'$ for some low security variable l[2]. We can use a model checker to check whether the latter is the case, and if so, find an initial state. For example, we can check whether the second program c_{ex2} violates secure information flow by checking whether the assertion in the following program is violated:

$$\begin{array}{l} \texttt{assume}(l = l'); \\ (\texttt{assume}(h);\ l := 1);\ (\texttt{assume}(!h');\ l' := 0); \\ \texttt{assert}(l = l'). \end{array}$$

A model checker can find that the assertion is indeed violated, and generate a counterexample $h = \texttt{true}, h' = \texttt{false}, l = l' = 0$, which is also a counterexample against the non-interference of the original program c_{ex2}.

Instead of combining type-based analysis and model checking as described above, it is possible to use model checking alone [1] to find a counterexample against the non-interference property. We, however, expect that our combination of type-based analysis and model checking can often find counterexamples faster than the method using model checking alone [1], since the type-based analysis allows us to prune innocent execution paths first, and to focus on particular pairs of execution paths in the model checking phase. For example, the program c_{ex1} contains 2^n execution paths (so the number of pairs of execution paths is 2^{2n}), but after type-based analysis, we only need to inspect a single pair of execution paths. The expectation that our method often works faster is supported by the result of preliminary experiments.

The rest of this paper is structured as follows. Section 2 introduces a simple imperative language and a type system for information flow analysis. The language and the type system are essentially the same as those of Volpano and Smith [20], except that minor modification has been made for the formalization of our method. Section 3 formalizes our method for finding a counterexample against non-interference. Section 4 and Section 5 report implementation of our method and experiments respectively. We have used the BLAST model checker [5] and implemented our method for a subset of C language. Section 6 compares our method with related work. Section 7 discusses future work and Section 8 concludes.

2. Language and Type System

This section introduces a simple imperative language and a type system for information flow analysis.

2.1 Language

The abstract syntax of the language is defined as follows:

$$\begin{array}{rrcl} \text{(Expression)} & a & ::= & x \mid n \mid a_1 + a_2 \mid\ !a \\ \text{(Labeled Expression)} & e & ::= & a^{\eta} \\ \text{(Command)} & c & ::= & (\texttt{skip})^{pc} \\ & & \mid & (x := e)^{pc} \\ & & \mid & (c_1;\ c_2)^{pc} \\ & & \mid & (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)^{pc} \\ & & \mid & (\texttt{while}\ e\ \texttt{do}\ c)^{pc} \end{array}$$

Here, the meta-variables x and n range over program variables and integers respectively. Labels ranged over by the meta-variables *pc* and η are attached to commands and expressions respectively. We assume that each command has a unique label *pc* and that each labeled expression has a unique label η. We write $\eta 2pc(\eta)$ for the label *pc* of the command in which an expression labeled by η occurs.

We write $(c, \sigma) \downarrow \sigma'$ for the evaluation relation, meaning that the program c with the initial state σ ends up in the state σ'. The formal definition of the relation is given in Figure 1. In the rules, $\sigma[x \mapsto n](y)$ has value n if $x = y$, and $\sigma(y)$ otherwise.

2.2 Type System for Secure Information Flow

Our type system for secure information flow is essentially the same as the standard type system of Volpano and Smith [20]. For the purpose of formalizing our method, however, we present the type system in a slightly different way:

1. Instead of explicitly considering a lattice of security levels, we just use variable names and labels (*pc*, η) as security levels. For example, a variable name x also stands for the security level of values stored in x.
2. Because of the difference above, we have no usual type environment that maps each variable to its type (i.e., its security level since we do not have compound data structures). Instead, we have a set of constraints of the form $\tau_1 \leq \tau_2$ (where τ_1, τ_2 are

[2] Actually, part of π_1 and π_2' can be shared by using the optimization proposed by Terauchi and Aiken [18].

$$\frac{}{(\mathtt{skip}, \sigma) \downarrow \sigma} \qquad \frac{(e, \sigma) \downarrow n}{(x := e, \sigma) \downarrow \sigma[x \mapsto n]}$$

$$\frac{}{(x, \sigma) \downarrow \sigma(x)} \qquad \frac{(c_1, \sigma) \downarrow \sigma'' \qquad (c_2, \sigma'') \downarrow \sigma'}{(c_1;\ c_2, \sigma) \downarrow \sigma'}$$

$$\frac{}{(n, \sigma) \downarrow n}$$

$$\frac{(a_i, \sigma) \downarrow n_i \qquad (\text{for } i = 1, 2)}{(a_1 + a_2, \sigma) \downarrow n_1 + n_2} \qquad \frac{(e, \sigma) \downarrow n(\neq 0) \qquad (c_1, \sigma) \downarrow \sigma'}{(\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \sigma) \downarrow \sigma'}$$

$$\frac{(a, \sigma) \downarrow n(\neq 0)}{(!a, \sigma) \downarrow 0} \qquad \frac{(e, \sigma) \downarrow 0 \qquad (c_2, \sigma) \downarrow \sigma'}{(\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \sigma) \downarrow \sigma'}$$

$$\frac{(a, \sigma) \downarrow 0}{(!a, \sigma) \downarrow 1} \qquad \frac{(e, \sigma) \downarrow n(\neq 0) \qquad (c, \sigma) \downarrow \sigma'' \quad (\mathtt{while}\ e\ \mathtt{do}\ c, \sigma'') \downarrow \sigma'}{(\mathtt{while}\ e\ \mathtt{do}\ c, \sigma) \downarrow \sigma'}$$

$$\frac{(e, \sigma) \downarrow 0}{(\mathtt{while}\ e\ \mathtt{do}\ c, \sigma) \downarrow \sigma}$$

Figure 1. The Evaluation Rules: $\boxed{(e, \sigma) \downarrow e'}$ and $\boxed{(c, \sigma) \downarrow \sigma'}$

variables or labels), which expresses the order of security levels (or, in which direction information may flow) as an assumption of a type judgment.

The syntax of types and type constraints are defined as follows:

$$\begin{array}{rrcl} \text{(Data Type)} & \tau & ::= & x \mid \eta \mid pc \\ \text{(Constraints)} & C & ::= & \{\tau_1 \leq \tau_1', \ldots, \tau_n \leq \tau_n\} \end{array}$$

A type judgment is of the form $C \vdash c : pc\ \mathtt{cmd}$ (for a command) or $C \vdash e : \tau$ (for a labeled expression). Intuitively, a judgment $C \vdash c : pc\ \mathtt{cmd}$ means that the command c conforms to the security policy C, which describes in which direction information may flow. $\tau_1 \leq \tau_2$ means that the security level τ_1 is lower than τ_2; in other words, information on a value of level τ_1 may flow to τ_2. For example, $\{y \leq x, pc \leq x\} \vdash (x := y)^{pc} : pc\ \mathtt{cmd}$ is a valid judgment, while $\{x \leq y, pc \leq x\} \vdash (x := y)^{pc} : pc\ \mathtt{cmd}$ is not. The label pc in a judgment $C \vdash c : pc\ \mathtt{cmd}$ expresses the security level of information about whether c is executed (which is actually the same as the label of c in our type system).

The typing rules are shown in Figure 2. In rule T-EXP, $\mathbf{FV}(a)$ denotes the set of variables occurring in a. We explain a few rules below. In rule T-ASSIGN, the constraint $pc \leq x$ captures the fact that information about whether the command is executed may flow to the value of variable x (since an observer may be able to guess whether the command was executed by looking at the final value of x). The constraint $pc \leq \tau_i$ in rule T-SEQ captures the fact that information about whether the command $c_1; c_2$ is executed flows to information about whether the command c_1 or c_2 is executed (since if an observer knows that c_1 has been executed, he or she can infer that $c_1; c_2$ has been executed). We write $C \models \tau_0 \leq \tau_n$ if there exists $\tau_1, \ldots, \tau_{n-1}$ such that $\tau_i \leq \tau_{i+1} \in C$ for each $i (0 \leq i < n)$.

In order to discuss the correctness of the type system, we formally define (T_1, T_2)-*non-interference* and its counterexample. (T_1, T_2)-*non-interference* states that the final values of variables in T_2 never depend on the initial values of variables in T_1.

Definition 2.1. A pair of states (σ_1, σ_2) is a counterexample against (T_1, T_2)-*non-interference* of a program c if: (i) $\sigma_1 =_{\overline{T_1}} \sigma_2$, (ii) $(c, \sigma_1) \downarrow \sigma_1'$ and $(c, \sigma_2) \downarrow \sigma_2'$ for some σ_1' and σ_2', and

$$\frac{x \leq \eta \in C \text{ for each } x \in \mathbf{FV}(a)}{C \vdash a^{\eta} : \eta} \quad \text{(T-EXP)}$$

$$\frac{}{C \vdash (\mathtt{skip})^{pc} : pc\ \mathtt{cmd}} \quad \text{(T-SKIP)}$$

$$\frac{C \vdash e : \tau \qquad pc \leq x \in C \qquad \tau \leq x \in C}{C \vdash (x := e)^{pc} : pc\ \mathtt{cmd}} \quad \text{(T-ASSIGN)}$$

$$\frac{C \vdash c_i : \tau_i\ \mathtt{cmd} \qquad pc \leq \tau_i \in C \qquad (\text{for } i = 1, 2)}{C \vdash (c_1;\ c_2)^{pc} : pc\ \mathtt{cmd}} \quad \text{(T-SEQ)}$$

$$\frac{\begin{array}{c} C \vdash e : \tau \qquad C \vdash c_i : \tau_i\ \mathtt{cmd} \\ pc \leq \tau_i \in C \qquad \tau \leq \tau_i \in C \qquad (\text{for } i = 1, 2) \end{array}}{C \vdash (\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2)^{pc} : pc\ \mathtt{cmd}} \quad \text{(T-IF)}$$

$$\frac{C \vdash e : \tau \qquad C \vdash c : \tau'\ \mathtt{cmd} \qquad pc \leq \tau' \in C \qquad \tau \leq \tau' \in C}{C \vdash (\mathtt{while}\ e\ \mathtt{do}\ c)^{pc} : pc\ \mathtt{cmd}} \quad \text{(T-WHILE)}$$

Figure 2. The Typing Rules: $\boxed{C \vdash e : \tau}$ and $\boxed{C \vdash c : pc\ \mathtt{cmd}}$

(iii) $\sigma_1' \neq_{T_2} \sigma_2'$. If there is no counterexample, we say that c satisfies (T_1, T_2)-*non-interference*. Here, $\sigma_1 =_T \sigma_2$ means that $\sigma_1(z) = \sigma_2(z)$ holds for every $z \in T$. $\overline{T}$ is the complement of T.

The following is a standard type soundness theorem:

Theorem 2.1 (soundness). *Let H be the set of high security variables and L be the set of low security variables. Suppose $C \not\models h \leq l$ for every $h \in H$ and $l \in L$. If $C \vdash c : pc\ \mathtt{cmd}$ holds, then c satisfies (H, L)-non-interference.*

3. Algorithm of Finding Counterexample

As mentioned in Section 1, our method finds a pair of execution paths that may violate the non-interference property (phase A), and then checks whether the pair of execution paths are unsafe (phase B). Each of (A) and (B) consists of the following sub-steps.

(A.1) Finding Suspicious Flow Perform type inference to find a suspicious flow of information from a high-security variable to a low-security variable that may be caused by a program.

(A.2) Path Pruning Generate a pair of execution paths that may cause the suspicious flow of information found in A.1.

(B.1) Self-Composition Express the condition that the pair of execution paths (generated in A.2) satisfies the non-interference property as a safety property, by using the idea of self-composition [1, 18].

(B.2) Finding Counterexample Use a model checker to find a counterexample (if any) against the safety property generated in B.1. That counterexample serves as a counterexample against the non-interference property of the original program.

We explain each step in Subsections 3.1-3.4 by using the following program c_{ex3}:

$$\left(\begin{array}{l} \left(\begin{array}{ll} \mathtt{if}\ b^{\eta_1} & \mathtt{then}\ (x := h^{\eta_2})^{pc_2} \\ & \mathtt{else}\ (x := 0^{\eta_3})^{pc_3} \end{array} \right)^{pc_1}; \\ \left(\mathtt{if}\ (x = 1)^{\eta_4}\ \begin{array}{l} \mathtt{then}\ (l := 1^{\eta_5})^{pc_5} \\ \mathtt{else}\ (l := 0^{\eta_6})^{pc_6} \end{array} \right)^{pc_4} \end{array} \right)^{pc_0}$$

3.1 Finding Suspicious Flow

Given a program c, we first obtain the least set C of type constraints such that $C \vdash c : pc\ \mathtt{cmd}$; Such an algorithm is easily obtained by

$$\frac{(e,\sigma)\downarrow n(\neq 0)}{(\texttt{assume}(e),\sigma)\downarrow\sigma}\qquad \frac{(\pi_1,\sigma)\downarrow\sigma'}{(\texttt{if}\ *\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2,\sigma)\downarrow\sigma'}$$

$$\frac{(e,\sigma)\downarrow n(\neq 0)}{(\texttt{assert}(e),\sigma)\downarrow\sigma}\qquad \frac{(\pi_2,\sigma)\downarrow\sigma'}{(\texttt{if}\ *\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2,\sigma)\downarrow\sigma'}$$

$$\frac{(e,\sigma)\downarrow 0}{(\texttt{assert}(e),\sigma)\downarrow \textbf{error}}\qquad \frac{(\pi,\sigma)\downarrow\sigma''\quad(\texttt{while}\ *\ \texttt{do}\ \pi,\sigma'')\downarrow\sigma'}{(\texttt{while}\ *\ \texttt{do}\ \pi,\sigma)\downarrow\sigma'}$$

$$\frac{}{(\texttt{while}\ *\ \texttt{do}\ \pi,\sigma)\downarrow\sigma}$$

Figure 3. The Evaluation Rules for New Commands: $\boxed{(\pi,\sigma)\downarrow\sigma'}$

$$\frac{(\pi,\sigma_1)\downarrow\sigma_1'\qquad(\pi,\sigma_2)\downarrow\sigma_2'}{(\langle\pi\rangle,\sigma_1,\sigma_2)\downarrow(\sigma_1',\sigma_2')}\qquad \frac{(\pi_2,\sigma_1)\downarrow\sigma_1'\qquad(\pi_1,\sigma_2)\downarrow\sigma_2'}{(\pi_1\oplus\pi_2,\sigma_1,\sigma_2)\downarrow(\sigma_1',\sigma_2')}$$

$$\frac{(\Pi_1,\sigma_1,\sigma_2)\downarrow(\sigma_1'',\sigma_2'')\quad(\Pi_2,\sigma_1'',\sigma_2'')\downarrow(\sigma_1',\sigma_2')}{(\Pi_1;\ \Pi_2,\sigma_1,\sigma_2)\downarrow(\sigma_1',\sigma_2')}\qquad \frac{(\pi,\sigma_1)\downarrow\sigma_1''\qquad(\pi,\sigma_2)\downarrow\sigma_2''\quad(\texttt{repeat}\langle\pi\rangle,\sigma 1'',\sigma 2'')\downarrow(\sigma_1',\sigma_2')}{(\texttt{repeat}\langle\pi\rangle,\sigma_1,\sigma_2)\downarrow(\sigma_1',\sigma_2')}$$

$$\frac{(\pi_1,\sigma_1)\downarrow\sigma_1'\qquad(\pi_2,\sigma_2)\downarrow\sigma_2'}{(\pi_1\oplus\pi_2,\sigma_1,\sigma_2)\downarrow(\sigma_1',\sigma_2')}\qquad \frac{}{(\texttt{repeat}\langle\pi\rangle,\sigma_1,\sigma_2)\downarrow(\sigma_1,\sigma_2)}$$

Figure 4. The Evaluation Rules for Doubled Programs: $\boxed{(\Pi,\sigma_1,\sigma_2)\downarrow(\sigma_1',\sigma_2')}$

reading the typing rules in Figure 2 in a bottom-up manner. For example, the following constraints are obtained for c_{ex3}:

$$\begin{aligned} C \;=\; \{&pc_0\le pc_1, pc_0\le pc_4,\\ &b\le\eta_1,\eta_1\le pc_2,\eta_1\le pc_3, pc_1\le pc_2, pc_1\le pc_3,\\ &h\le\eta_2,\eta_2\le x, pc_2\le x,\eta_3\le x, pc_3\le x,\\ &x\le\eta_4,\eta_4\le pc_5,\eta_4\le pc_6, pc_4\le pc_5, pc_4\le pc_6,\\ &\eta_5\le l, pc_5\le l,\eta_6\le l, pc_6\le l\}. \end{aligned}$$

Then, a suspicious flow (path) of information is constructed from the set of constraints. Let C be the least C' such that $C'\vdash c$: *pc* cmd. A sequence of data types $\tau_1\cdot\ldots\cdot\tau_n$ is called an *information flow path* from τ_1 to τ_2 of a program c, if $\tau_i\le\tau_{i+1}\in C$ for each $i(1\le i\le n-1)$, and no variable occurs twice in the sequence. Let $\mathit{flows}(c,T_1,T_2)$ denotes the set of information flow paths of c from an element of T_1 to an element of T_2. We call an information flow path $\tau_1\cdot\ldots\cdot\tau_n$ suspicious if $\tau_1\cdot\ldots\cdot\tau_n\in\mathit{flows}(c,H,L)$, where H and L are sets of high- and low-security variables respectively. If c has no suspicious flow, c has secure information flow. For example, for the program c_{ex3}, $\mathit{flows}(c_{ex3},H,L)$ is:

$$\{h\cdot\eta_2\cdot x\cdot\eta_4\cdot pc_5\cdot l,\ h\cdot\eta_2\cdot x\cdot\eta_4\cdot pc_6\cdot l\}.$$

Here, we use the first information flow path $h\cdot\eta_2\cdot x\cdot\eta_4\cdot pc_5\cdot l$ to explain the succeeding steps.

3.2 Path Pruning

The next step is to construct a pair of execution paths that may cause the suspicious information flow path obtained in the previous step (in other words, to prune paths that are irrelevant to the information flow path).

We first convert the information flow path to a sequence of *pc* labels, which represents control flow information, by using the function *f2pc* given below. (Recall that η*2pc*(η) denotes the label of the command in which η occurs.)

$$\begin{aligned} \mathit{f2pc}(x) &= \epsilon\\ \mathit{f2pc}(x\cdot\eta\cdot y\cdot\vec{\tau}) &= \eta\mathit{2pc}(\eta)\cdot\mathit{f2pc}(y\cdot\vec{\tau})\\ \mathit{f2pc}(x\cdot\eta\cdot p\vec{c}\cdot pc\cdot y\cdot\vec{\tau}) &= \eta\mathit{2pc}(\eta)\cdot pc\cdot\mathit{f2pc}(y\cdot\vec{\tau}) \end{aligned}$$

The second clause of the definition of *f2pc* captures the flow caused by an assignment, and the third clause captures the flow caused by an if- or a while-statement.

The sequence generated by *f2pc* consists of the labels of if-statements, while-statements and assignments. Intuitively, the label of an if-statement occurring in the sequence expresses that two execution paths must split into the then-branch and the else-branch. Similarly, the label of a while-statement means that the two execution paths must go through the while-loop the same number of times first, then the two must split: one of the paths must exit the loop and the other must not. The label of an assignment expresses that one of the two execution paths must execute the assignment if the two have splited, otherwise both of the two must execute the assignment. For example,

$$\mathit{f2pc}(h\cdot\eta_2\cdot x\cdot\eta_4\cdot pc_5\cdot l)=pc_2\cdot pc_4\cdot pc_5.$$

This sequence intuitively expresses that two execution paths should first go through the assignment labeled by pc_2, split into the then- and the else-branch of the if-statement labeled by pc_4, and then one of the paths must go through the assignment labeled by pc_5.

Next, we construct the set of pairs of execution paths expressed by the sequence of *pc* labels. The set of pairs of execution paths is expressed as a doubled program given below.

Definition 3.1. The sets of *extended commands* and *doubled programs* are given by the following syntax:

$$\begin{array}{rrcl} \text{(extended commands)} & \pi & ::= & \texttt{skip}\\ & & \mid & x:=e\\ & & \mid & \texttt{assume}(e)\\ & & \mid & \texttt{assert}(e)\\ & & \mid & \pi_1;\ \pi_2\\ & & \mid & \texttt{if}\ e\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2\\ & & \mid & \texttt{while}\ e\ \texttt{do}\ \pi\\ & & \mid & \texttt{if}\ *\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2\\ & & \mid & \texttt{while}\ *\ \texttt{do}\ \pi\\ \text{(doubled programs)} & \Pi & ::= & \langle\pi\rangle\\ & & \mid & \Pi_1;\ \Pi_2\\ & & \mid & \pi_1\oplus\pi_2\\ & & \mid & \texttt{repeat}\langle\pi\rangle \end{array}$$

An extended command expresses a set of execution paths. The new command $\texttt{assume}(e)$ checks that the value of e is non-zero, and if so proceeds to the next command; If the value of e is 0, the evaluation gets stuck (so that the command never terminates). The command $\texttt{assert}(e)$ is the same, except that it terminates with a special state **error** if the value of e is 0. The assert command is only used in the next step, for converting the non-interference property into a safety property (that the state **error** is never reached). The command $\texttt{if}\ *\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2$ executes π_1 or π_2 in a non-deterministic manner. The command $\texttt{while}\ *\ \texttt{do}\ \pi$ repeats π an arbitrary number of times. The evaluation rules for the new commands are given in Figure 3.

A doubled program expresses a set of pairs of execution paths. $\langle\pi\rangle$ expresses that both of the two execution paths belong to (the set of execution paths expressed by) π. The doubled program $\pi_1\oplus\pi_2$ means that one of the execution paths belongs to π_1, while the other belongs to π_2. $\Pi_1;\Pi_2$ expresses the sequential composition of path pairs. For example, $(x:=1\oplus x:=2);(y:=1\oplus y:=2)$ expresses the set of path pairs $\{((x:=1;y:=1),(x:=2;y:=2)),((x:=1;y:=2),(x:=2;y:=1))\}$. $\texttt{repeat}\langle\pi\rangle$ expresses that both of the two execution paths repeat paths which belong to π

the same number of times. We write $(\Pi, \sigma_1, \sigma_2) \downarrow (\sigma_1', \sigma_2')$ for the evaluation relation, meaning that the doubled program Π with the initial states σ_1 and σ_2 ends up in the states σ_1' and σ_2'. Intuitively, the relation expresses that Π contains a pair of execution paths, one of which starts with state σ_1 and ends in σ_1' and the other starts with state σ_2 and ends in σ_2'. The formal definition of the relation is given in Figure 4.

We formalize a (non-deterministic) algorithm for constructing a doubled program from a command c and a sequence s of *pc* labels (generated by *f2pc*) by using a 4-tuple relation $s \vdash c \longrightarrow \Pi \dashv s'$. Intuitively, the relation means that Π is the set of pairs of two execution paths of the command c that conform to control flow expressed by $s \backslash s'$ (the sequence obtained by removing the postfix s' from s).

The rules for $s \vdash c \longrightarrow \Pi \dashv s'$ is given in Figure 5. In the rules, $\mathbf{PCs}(c)$ returns the set of *pc* labels occurring in the command c, $erase(c)$ erases all *pc* labels occurring in c, and $length(s)$ returns the length of the sequence s of *pc* labels. Let $\tau_1 \cdot \ldots \cdot \tau_n$ be a suspicious information flow path obtained in the previous step. By reading the rules in a bottom-up manner, we can obtain Π such that $\mathit{f2pc}(\tau_1 \cdot \ldots \cdot \tau_n) \vdash c \longrightarrow \Pi \dashv \epsilon$.[3] Then, the doubled program Π expresses a set of execution paths that may cause the suspicious information flow.

We explain some rules below. In rule P-ASSIGN, the first label *pc* of the sequence indicates that both of the two execution paths must execute $(x := e)^{pc}$. Consequently, the algorithm consumes *pc*, and generates $\langle x := e \rangle$. In rule P-IF-THEN, the first label *pc*′ of the sequence (together with the condition $pc' \in \mathbf{PCs}(c_1)$ indicates that both of the two execution paths must execute the then-branch. Thus, the algorithm prunes the else-branch, and generates a doubled program which goes through the then-branch. In rule P-IF-THEN-IMP, the notation $!e$ expresses a logical negation of the labeled expression e. The first label *pc* of the sequence indicates that the two execution paths must split into the then-branch c_1 and the else-branch c_2. Suppose that the second label *pc*′ occurs in c_1. The algorithm consumes *pc*, and generates $(\texttt{assume}(e);\ \pi) \oplus (\texttt{assume}(!e);\ c_2)$. Here, the extended command π is generated from c_1 by using the relation $s \vdash c \dashrightarrow \pi \dashv s'$. The relation $s \vdash c \dashrightarrow \pi \dashv s'$ is used to generate an extended command π that expresses a set of *single* execution paths that conform to the control flow expressed by $s \backslash s'$.

There are two rules P-WHILE and P-WHILE-IMP for while-statements. Rule P-WHILE-IMP covers the case where an indirect information flow (caused by the two execution paths going through the while-loop different numbers of times) occurs from the condition of the while-loop. The part $\texttt{repeat}\langle \texttt{assume}(e);\ erase(c) \rangle$ in the rule's output means that the two execution paths must first go through the while-loop the same number of times, and the part $\pi \oplus \texttt{assume}(!e)$ means that the paths must then split: one path further goes through the while-loop, while the other path exits. Rule P-WHILE handles the remaining case, where an information flow first occurs inside the body of the while-loop. The first part $\texttt{repeat}\langle \texttt{assume}(e);\ erase(c) \rangle$ of the rule's output means that the two execution paths must go through the while-loop the same number of times, and the second part $\langle \texttt{assume}(e) \rangle;\ \Pi$ means that the flow specified by *pc*′ must occur inside the body of the while-loop. For example, the following doubled program $\{\Pi_{\mathit{ex3}}\}$ is obtained from c_{ex3}.

$$\langle \texttt{assume}(b);\ x := h \rangle;$$
$$((\texttt{assume}(x = 1);\ l := 1) \oplus (\texttt{assume}(x \neq 1);\ l := 0)).$$

[3] There are finitely many such Π since the rules are non-deterministic. A deterministic, complete algorithm for generating a *single* doubled command for each pair of c and $\tau_1 \cdot \ldots \cdot \tau_n$ seems to be hard to construct. We discuss a deterministic, incomplete version of the algorithm in Section 4.1.

In order to discuss the correctness of the path pruning, we formally define (T_1, T_2)-*non-interference* of a doubled program and its counterexample.

Definition 3.2. A pair of states (σ_1, σ_2) is a counterexample against (T_1, T_2)-*non-interference* of a doubled program Π if: (i) $\sigma_1 =_{\overline{T_1}} \sigma_2$, (ii) $(\Pi, \sigma_1, \sigma_2) \downarrow (\sigma_1', \sigma_2')$ for some σ_1' and σ_2', and (iii) $\sigma_1' \neq_{T_2} \sigma_2'$. If there is no counterexample, we say that Π satisfies (T_1, T_2)-*non-interference*.

The soundness and completeness of path pruning are stated as the following theorems. Proofs are given in the full version of this paper [19].

Theorem 3.1 (soundness). *For any doubled program Π such that $s \vdash c \longrightarrow \Pi \dashv s'$ is derivable for some s and s', if a pair of states (σ_1, σ_2) is a counterexample against (H, L)-non-interference of Π, then (σ_1, σ_2) is a counterexample against (H, L)-non-interference of c.*

Theorem 3.2 (completeness). *If a pair of states (σ_1, σ_2) is a counterexample against (H, L)-non-interference of a program c, then there exist a doubled program Π and a sequence $\tau_1 \cdots \tau_n$ that satisfy the following conditions:*

(i) $\tau_1 \cdot \ldots \cdot \tau_n \in \mathit{flows}(c, H, L)$,
(ii) $\mathit{f2pc}(\tau_1 \cdot \ldots \cdot \tau_n) \vdash c \longrightarrow \Pi \dashv \epsilon$, *and*
(iii) (σ_1, σ_2) *is a counterexample against* $(H, \{\tau_n\})$*-non-interference of* Π.

Intuitively, the soundness means that if there exists a counterexample for a doubled program Π which is obtained from a program c, then it is a counterexample for c. The completeness means that any counterexample for c is a counterexample for some doubled program Π which is obtained from c.

3.3 Self-Composition

The next step is to turn the non-interference property of the doubled program obtained in the previous step into a safety property, by using the idea of self-composition [1]. As observed by Barthe et al. [1], a program c satisfies (T_1, T_2)-non-interference if and only if the following self-composition of c never reach the state **error**.

$$\texttt{assume}(\vec{u} = \xi(\vec{u}));\ c;\ \xi(c);\ \texttt{assert}(\vec{v} = \xi(\vec{v})),$$

Here, $\xi(\vec{u})$ and $\xi(c)$ are the sequence of variables and the command obtained from $\vec{u}$ and c by renaming each variable x with a fresh variable x'. $\vec{u}$ is the sequence consisting of all the variables which is not in T_1, and $\vec{v}$ is the sequence consisting of all the variables in T_2.

Similarly, a doubled program Π satisfies (T_1, T_2)-non-interference if and only if the following self-composition never reach the state **error**.

$$\texttt{assume}(\vec{u} = \xi(\vec{u}));\ [\![\Pi]\!];\ \texttt{assert}(\vec{v} = \xi(\vec{v})),$$

Here, $[\![\cdot]\!]$ is defined in Figure 6. For a double program Π obtained using a suspicious flow $\tau_1 \cdot \ldots \cdot \tau_n$, we use $T_1 = H$ and $T_2 = \{\tau_n\}$.

Actually, we adopt an optimized self-composition proposed by Terauchi and Aiken [18] for statements whose effects never depend on the values of the variables in T_1. A predicate $bd(e)$ holds if the value of e may depend on the values of the variables in T_1:

$$bd(a^\eta) \equiv \mathit{flows}(c, T_1, \{\eta\}) \neq \emptyset.$$

For example, an optimized self-composition of Π_{ex3} yields:

$$\texttt{assume}(l = l' \text{ and } x = x' \text{ and } b = b');$$
$$\texttt{assume}(b);\ x := h;\ x' := h';$$
$$\texttt{assume}(x = 1);\ l := 1;\ \texttt{assume}(x' \neq 1);\ l' := 0;$$
$$\texttt{assert}(l = l').$$

$$\frac{}{s \vdash c \longrightarrow \langle erase(c)\rangle \dashv s} \quad \text{(P-EPSILON)}$$

$$\frac{}{pc \cdot s \vdash (x := e)^{pc} \longrightarrow \langle x := e\rangle \dashv s} \quad \text{(P-ASSIGN)}$$

$$\frac{s \vdash c_1 \longrightarrow \Pi_1 \dashv s'' \qquad s'' \vdash c_2 \longrightarrow \Pi_2 \dashv s'}{s \vdash c_1;\ c_2 \longrightarrow \Pi_1;\ \Pi_2 \dashv s'} \quad \text{(P-SEQ)}$$

$$\frac{pc' \in \mathbf{PCs}(c_1) \qquad pc' \cdot s \vdash c_1 \longrightarrow \Pi \dashv s'}{pc' \cdot s \vdash (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)^{pc} \longrightarrow \langle \texttt{assume}(e)\rangle;\ \Pi \dashv s'} \quad \text{(P-IF-THEN)}$$

$$\frac{pc' \in \mathbf{PCs}(c_2) \qquad pc' \cdot s \vdash c_2 \longrightarrow \Pi \dashv s'}{pc' \cdot s \vdash (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)^{pc} \longrightarrow \langle \texttt{assume}(!e)\rangle;\ \Pi \dashv s'} \quad \text{(P-IF-ELSE)}$$

$$\frac{pc' \in \mathbf{PCs}(c_1) \qquad pc' \cdot s \vdash c_1 \dashrightarrow \pi \dashv s'}{pc \cdot pc' \cdot s \vdash (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)^{pc} \longrightarrow (\texttt{assume}(e);\ \pi) \oplus (\texttt{assume}(!e);\ erase(c_2)) \dashv s'} \quad \text{(P-IF-THEN-IMP)}$$

$$\frac{pc' \in \mathbf{PCs}(c_2) \qquad pc' \cdot s \vdash c_2 \dashrightarrow \pi \dashv s'}{pc \cdot pc' \cdot s \vdash (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)^{pc} \longrightarrow (\texttt{assume}(e);\ erase(c_1)) \oplus (\texttt{assume}(!e);\ \pi) \dashv s'} \quad \text{(P-IF-ELSE-IMP)}$$

$$\frac{pc' \in \mathbf{PCs}(c) \qquad length(pc' \cdot s) > length(s'') \qquad pc' \cdot s \vdash c \longrightarrow \Pi \dashv s'' \qquad s'' \vdash (\texttt{while}\ e\ \texttt{do}\ c)^{pc} \longrightarrow \Pi' \dashv s'}{pc' \cdot s \vdash (\texttt{while}\ e\ \texttt{do}\ c)^{pc} \longrightarrow \texttt{repeat}\langle \texttt{assume}(e);\ erase(c)\rangle;\ \langle \texttt{assume}(e)\rangle;\ \Pi;\ \Pi' \dashv s'} \quad \text{(P-WHILE)}$$

$$\frac{pc' \in \mathbf{PCs}(c) \qquad pc' \cdot s \vdash (\texttt{while}\ e\ \texttt{do}\ c)^{pc} \dashrightarrow \pi \dashv s'}{pc \cdot pc' \cdot s \vdash (\texttt{while}\ e\ \texttt{do}\ c)^{pc} \longrightarrow \texttt{repeat}\langle \texttt{assume}(e);\ erase(c)\rangle;\ (\pi \oplus \texttt{assume}(!e)) \dashv s'} \quad \text{(P-WHILE-IMP)}$$

$$\frac{}{s \vdash c \dashrightarrow erase(c) \dashv s} \quad \text{(Q-EPSILON)}$$

$$\frac{}{pc \cdot s \vdash (x := e)^{pc} \dashrightarrow x := e \dashv s} \quad \text{(Q-ASSIGN)}$$

$$\frac{s \vdash c_1 \dashrightarrow \pi_1 \dashv s'' \qquad s'' \vdash c_2 \dashrightarrow \pi_2 \dashv s'}{s \vdash c_1;\ c_2 \dashrightarrow \pi_1;\ \pi_2 \dashv s'} \quad \text{(Q-SEQ)}$$

$$\frac{pc' \in \mathbf{PCs}(c_1) \qquad pc' \cdot s \vdash c_1 \dashrightarrow \pi \dashv s'}{pc' \cdot s \vdash (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)^{pc} \dashrightarrow \texttt{assume}(e);\ \pi \dashv s'} \quad \text{(Q-IF-THEN)}$$

$$\frac{pc' \in \mathbf{PCs}(c_2) \qquad pc' \cdot s \vdash c_2 \dashrightarrow \pi \dashv s'}{pc' \cdot s \vdash (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)^{pc} \dashrightarrow \texttt{assume}(!e);\ \pi \dashv s'} \quad \text{(Q-IF-ELSE)}$$

$$\frac{pc' \in \mathbf{PCs}(c) \qquad length(pc' \cdot s) > length(s'') \qquad pc' \cdot s \vdash c \dashrightarrow \pi \dashv s'' \qquad s'' \vdash (\texttt{while}\ e\ \texttt{do}\ c)^{pc} \dashrightarrow \pi' \dashv s'}{pc' \cdot s \vdash (\texttt{while}\ e\ \texttt{do}\ c)^{pc} \dashrightarrow \texttt{while}\ *\ \texttt{do}\ (\texttt{assume}(e);\ erase(c));\ \texttt{assume}(e);\ \pi;\ \pi' \dashv s'} \quad \text{(Q-WHILE)}$$

Figure 5. The Path Pruning Rules: $\boxed{s \vdash c \longrightarrow \Pi \dashv s'}$ and $\boxed{s \vdash c \dashrightarrow \pi \dashv s'}$

$$\begin{array}{rcll}
[\![\langle \texttt{skip}\rangle]\!] & = & \texttt{skip} & \\
[\![\langle x := e\rangle]\!] & = & x := e;\ \xi(x) := x & (\neg bd(e)) \\
[\![\langle x := e\rangle]\!] & = & x := e;\ \xi(x) := \xi(e) & (bd(e)) \\
[\![\langle \texttt{assume}(e)\rangle]\!] & = & \texttt{assume}(e) & (\neg bd(e)) \\
[\![\langle \texttt{assume}(e)\rangle]\!] & = & \texttt{assume}(e);\ \texttt{assume}(\xi(e)) & (bd(e)) \\
[\![\langle \pi_1;\ \pi_2\rangle]\!] & = & [\![\langle \pi_1\rangle]\!];\ [\![\langle \pi_2\rangle]\!] & \\
[\![\langle \texttt{if}\ e\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2\rangle]\!] & = & \texttt{if}\ e\ \texttt{then}\ [\![\langle \pi_1\rangle]\!]\ \texttt{else}\ [\![\langle \pi_2\rangle]\!] & (\neg bd(e)) \\
[\![\langle \texttt{if}\ e\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2\rangle]\!] & = & \texttt{if}\ e\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2; & (bd(e)) \\
 & & \texttt{if}\ \xi(e)\ \texttt{then}\ \xi(\pi_1)\ \texttt{else}\ \xi(\pi_2) & \\
[\![\langle \texttt{while}\ e\ \texttt{do}\ \pi\rangle]\!] & = & \texttt{while}\ e\ \texttt{do}\ [\![\langle \pi\rangle]\!] & (\neg bd(e)) \\
[\![\langle \texttt{while}\ e\ \texttt{do}\ \pi\rangle]\!] & = & \texttt{while}\ e\ \texttt{do}\ \pi;\ \texttt{while}\ \xi(e)\ \texttt{do}\ \xi(\pi) & (bd(e)) \\
[\![\langle \texttt{if}\ *\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2\rangle]\!] & = & \texttt{if}\ *\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2; & \\
 & & \texttt{if}\ *\ \texttt{then}\ \xi(\pi_1)\ \texttt{else}\ \xi(\pi_2) & \\
[\![\langle \texttt{while}\ *\ \texttt{do}\ \pi\rangle]\!] & = & \texttt{while}\ *\ \texttt{do}\ \pi;\ \texttt{while}\ *\ \texttt{do}\ \xi(\pi) & \\
[\![\Pi_1;\ \Pi_2]\!] & = & [\![\Pi_1]\!];\ [\![\Pi_2]\!] & \\
[\![\pi_1 \oplus \pi_2]\!] & = & \texttt{if}\ *\ \texttt{then}\ \pi_1;\ \xi(\pi_2)\ \texttt{else}\ \pi_2;\ \xi(\pi_1) & \\
[\![\texttt{repeat}\langle \pi\rangle]\!] & = & \texttt{while}\ *\ \texttt{do}\ [\![\langle \pi\rangle]\!] &
\end{array}$$

Figure 6. The Self-Composition of Path Pairs

3.4 Finding Counterexample

The last step is to check whether the self-composition obtained in the previous step reaches an error state, and generate a counterexample if so. In the experiments reported in Section 5, we used the model checker BLAST [5] for this purpose. For example, the following counterexample is obtained for c_{ex3}:

$$l = l' = 0, x = x' = 0, b = b' = \texttt{true}, h = 1, h' = 0.$$

4. Implementation

We have implemented our algorithm for a subset of C. We used CIL [11] for parsing programs, and used BLAST model checker [5] for the fourth step of the algorithm described in Section 3. There are some discrepancies between the theory described in Section 3 and the actual implementation. First, we have traded the completeness for the efficiency. Second, we have extended the target language and its type system for information flow to handle C-specific features (pointers and jumps). We discuss the first point in Subsection 4.1 and then discuss the second point in Subsection 4.2.

4.1 Optimization of the Algorithm

Our implementation applies a few optimizations to speed up our algorithm for finding a counterexample. Some of them sacrifice the completeness of the algorithm. We think that losing the completeness is not so big a problem, since we can always revert to the usual self-composition approach [18] (or just give up) when our algorithm fails to find a counterexample.

- Prioritization of suspicious information flow paths
  In the first step of our algorithm, more than one suspicious information flow paths may be found. Our system inspects those paths one by one (until a counterexample is found), giving a higher priority to information flow path which involves fewer indirect flows.
- Deterministic algorithm for constructing doubled programs
  In the second step, more than one doubled programs may be generated for each suspicious information flow path. That is because the rules in Figure 5 are non-deterministic; Note that the rule P-EPSILON is applicable to any command c. In the actual implementation, rules P-EPSILON and Q-EPSILON are replaced with the following rules.

$$\frac{pc \notin PCs(c)}{pc \cdot s \vdash c \longrightarrow \langle erase(c)\rangle \dashv pc \cdot s} \quad \text{(P-EPSILON-DET)}$$

$$\frac{pc \notin PCs(c)}{pc \cdot s \vdash c \dashrightarrow erase(c) \dashv pc \cdot s} \quad \text{(Q-EPSILON-DET)}$$

  Because of this change, the completeness (Theorem 3.2) is lost for some tricky programs (see the full version [19]). However, this change makes the execution path generation algorithm deterministic (so that only one doubled program is generated for each information flow path), which helps the system to avoid performing duplicated computation for similar paths. As a result, a counterexample is often found much faster.
- A user can optionally instruct the system to replace the while-loop with a loop-free program which simulates the while-loop only a finite number of times. Again, this loses the completeness.
- Given a suspicious flow $\tau_1 \cdot \ldots \cdot \tau_n$ and a doubled program Π which is constructed from the flow, the system tries to prove $(\{\tau_1\}, \{\tau_n\})$-non-interference of Π instead of $(H, \{\tau_n\})$-non-interference. That is partially justified by the fact that, if a pair of initial states $(\{h_1 \mapsto n_1, h_2 \mapsto n_2\}, \{h_1 \mapsto n_1', h_2 \mapsto n_2'\})$ is a counterexample, then $(\{h_1 \mapsto n_1, h_2 \mapsto n_2\}, \{h_1 \mapsto n_1', h_2 \mapsto n_2\})$ or $(\{h_1 \mapsto n_1', h_2 \mapsto n_2\}, \{h_1 \mapsto n_1', h_2 \mapsto n_2'\})$ is also a counterexample unless c diverges for the initial state $\{h_1 \mapsto n_1', h_2 \mapsto n_2\}$.

  The optimization above has two kinds of effects. First, Terauchi and Aiken's optimized self-composition becomes more effective since we can set $T_1 = \{\tau_1\}$ in the definition of $bd(e)$ in Section 3.3. Second, the search space explored by the model checker is reduced by adding the assumption $h_1 = h_1' \wedge \cdots \wedge h_n = h_n'$ about the input states, where $\{h_1, \ldots, h_n\} = H \setminus \{\tau_1\}$.

4.2 Extensions for C Programming Language

Our system supports various features of C programming language, including pointers, arrays, structures, unions, and a restricted form of jumps. Our system does not support function pointers and recursive functions. The current implementation of type-based analysis is intraprocedural. Thus, path pruning may only exploit local information within a function. Insecure information flows via global variables are overlooked. However, the system may perform interprocedural analysis by inlining functions before a type-based analysis as we do in experiments in Section 5.

The type system and the algorithm for finding a counterexample have been extended accordingly. We briefly explain those extensions below.

4.2.1 Handling pointers, arrays, structures, and unions

Operations on arrays, structures, and unions are translated to pointer operations. In order to track information flow caused by pointer operations, we introduce extended data types θ:

$$\begin{array}{rcl} \theta & ::= & \texttt{int}^{\tau} \mid \cdots \mid \theta\, \texttt{ptr}^{\tau} \\ \tau & ::= & x \mid \eta \mid pc. \end{array}$$

The typing rules for creating, dereferencing, and updating pointers ($\&e$, $*e$, and $*x = e$) are similar to those of the type system for secure information flow in ML [15].

To handle pointer arithmetic, we assume that programs are compiled by using a C compiler ensuring the memory safety like CCured [2, 12] and Fail-safe C [13], which guarantees that any access to invalid memory regions (like an access beyond an array boundary) is caught at a run-time. (Without that assumption, we have to assume that a pointer $p + n$ may point to any part of memory.) Moreover, we force every element stored in the same memory region (like an array, a structure, or a union) to have the same security level. On that assumption, we use the following typing rule for pointer arithmetic:

$$\frac{\Gamma; C \vdash e : \theta\, \texttt{ptr}^{\tau_1} \qquad \Gamma; C \vdash n : \texttt{int}^{\tau_2} \qquad \tau_1 \leq \tau_3, \tau_2 \leq \tau_3 \in C}{\Gamma; C \vdash e + n : \theta\, \texttt{ptr}^{\tau_3}}$$

4.2.2 Handling restricted control structures

C has jump instructions such as goto, break, and continue. Our system supports only jumps into the outward and forward direction. Two non-trivial extensions are required to handle the restricted jumps.

First, we need to extend the type system so that it can handle indirect flows caused by the restricted jumps. For example, the following program has indirect flow from h to l:

$$\begin{array}{l} l := 0; \\ \texttt{while}\ 1\ \texttt{do}\ (\texttt{if}\ h\ \texttt{then goto}\ BL);\ l := 1;\ \cdots \\ BL:\ \cdots \end{array}$$

For such a program, the type system must impose constraints $\{pc \leq pc_1, \ldots, pc \leq pc_n\}$, where pc is the label of the goto-statement and $pc_1, \ldots, pc_n$ are the labels of the statements that may be skipped by the jump.

Second, we need to modify the algorithm for making the self-composition (the third step of the algorithm described in Section 3). In fact, the following self-composition of the program above is incorrect:

$$
\begin{array}{l}
l := 0;\ l' := l; \\
\texttt{while}\ 1\ \texttt{do} \\
\quad (\texttt{if}\ h\ \texttt{then goto}\ BL);\ (\texttt{if}\ h'\ \texttt{then goto}\ BL); \\
\quad l := 1;\ l' := l;\ \cdots \\
BL:\ \cdots
\end{array}
$$

Note that even if h is true, $\texttt{if}\ h'\ \texttt{then}\ \cdots; l' := 1$ should be executed since h' may be false, in which case $l' := 1$ should be executed. For the program above, the current implementation duplicates the while-statement as follows:

$$
\begin{array}{l}
l := 0;\ l' := l; \\
\texttt{while}\ 1\ \texttt{do}\ (\texttt{if}\ h\ \texttt{then goto}\ BL);\ l := 1;\ \cdots \\
BL: \\
\texttt{while}\ 1\ \texttt{do}\ (\texttt{if}\ h'\ \texttt{then goto}\ BL');\ l' := 1;\ \cdots \\
BL':\ \cdots
\end{array}
$$

5. Experiments

The main purpose of the experiments is to show that our method of combining type-based analysis and model checking is effective for showing that certain programs rejected by type-based analysis are indeed unsafe (by finding a counter-example against non-interference). *In principle*, our method should be effective also for certifying that certain programs rejected by type-based analysis are actually secure. The current implementation of our method is, however, not so effective for the latter purpose: See the discussion in Section 7.2.

To evaluate our method, we have also implemented the original self-composition method of Barthe et al. [1] and Terauchi and Aiken's optimized self-composition method [18], and compared the running times for several programs. The result is summarized in Table 1.

Programs `expr1.c` to `expr4.c` are artificial examples. We examined several artificial examples. The program `expr1.c` is an instance of c_{ex1} discussed in Section 1. It has 2^{16} paths, and only one path leaks information of `high` to `low` through x_0 to x_{16}. The result shows that our method is much faster than the others.

`expr2.c` leaks information of `high` to `low` indirectly through x_0 to x_{15}. It has 2^{16} paths, but non-interference is violated only when two execution paths take different branches at every if-statement. Previous methods [1, 18] have to inspect $2^{16} \times 2^{16} = 2^{32}$ pairs of execution paths in the worst case to find a counterexample, while our method only needs to inspect 2^{16} pairs of execution paths. The table shows that our method is much faster than the others. The speed-up depends also on how BLAST search execution paths, and `expr2.c` is an ideal case, where the execution path that was first inspected by BLAST was a counterexample. To make fairer comparison, we also measured the running times for a variant `expr2'.c`, which is obtained by swapping bodies of then- and else-clauses of every if-statements and reducing the number of if-statements to 8. The result shows that our method is still faster than the others.

`expr3.c` is a secure program, although the type system reports a false alarm by detecting superfluous flows from `high` to `x` by the if-statement. The running times show how long it takes for each system to report that programs are actually secure. The result shows that our method is slightly slower than Terauchi and Aiken's method. It is because our current method generates and checks two suspicious information flows: `high` to pc_1 to x to `low`, and `high` to pc_2 to x to `low`, where pc_1 and pc_2 are labels of then- and else-clauses of the if-statement.

```
foo(intH high)
{
  int b1, b2, b3, ..., b16;
  int x0, x1, x2, ..., x16;
  intL low;
  x0 = high;
  if (b1 ) { x1  = x0; }
  if (b2 ) { x2  = x1; }
  if (b3 ) { x3  = x2; }
  ...
  if (b16) { x16 = x15; }
  low = x16;
}
```

Figure 7. A Program for Experiment (expr1.c)

```
foo(intH high)
{
  int x0, x1, x2, ..., x16;
  intL low;
  x0 = high;
  if (x0 ) { x1  = 1; }
  if (x1 ) { x2  = 1; }
  if (x2 ) { x3  = 1; }
  ...
  if (x15) { x16 = 1; }
  low = x16;
}
```

Figure 8. A Program for Experiment (expr2.c)

In general, our current method is not good at proving that a program satisfies non-interference. That is because our method may generate multiple doubled programs, and runs a model checker repeatedly for each of them. The doubled programs may share part of the original program, which result in duplicated computation by the model checker. If combined with the compositional analysis discussed in Section 7, however, our method would also be effective for verifying that a program is secure.

`expr4.c` uses 20 high-security variables and 20 low-security variables. There is an insecure flow from $\texttt{high}_{10}$ to $\texttt{low}_{10}$. From the result of type inference, our system first infers that the only possibility of information leakage is from $\texttt{high}_i$ to $\texttt{low}_i$ (where $i \in \{1, \ldots, 20\}$), rather than an arbitrary combination of $\texttt{high}_i$ and $\texttt{low}_j$. As the table shows, our system runs faster than the other methods, by taking advantage of that information .

In addition to the small artificial programs above, we have also used a real program to compare our method with the others. The program `mod_imagemap.c` is an Apache httpd server module `mod_imagemap.c`, which realizes server side image maps. This program is known to have a Cross-Site Scripting vulnerability due to an omission of sanitizing `HTTP_REFERER` (fixed in Apache httpd 2.2.1-dev). The system found suspicious flows from return values of a function `imap_url`, which may be tainted by `HTTP_REFERER` into arguments of a function `ap_rvputs()`,

Program	LOC	Naive Self-Composition (sec.)	Optimized Self-Composition (sec.)	Our Method (sec.)
expr1.c	25	2000.765	75.657	1.493
expr2.c	24	307.892	305.890	2.317
expr2'.c	16	117.349	84.111	33.027
expr3.c	19	fail	1.774	2.181
expr4.c	28	145.841	31.857	8.848
mod_imagemap.c	895	658.357	150.356	117.849

Table 1. Experimental Results (Intel Pentium III 500MHz 256MB RAM)

```
foo (intH high)
{
  int n, f1, f2, x, i;
  intL low;
  while (0 < n) {
    f1 = f1 + f2; f2 = f1 - f2; n  = n - 1;
  }
  if (high) { x = 1; } else { x = 1; }
  while (i < f1) {
    low = low + x; i = i + 1;
  }
}
```

Figure 9. A Program for Experiment (expr3.c)

```
foo(intH high1, ..., intH high20)
{
  int b1, b2, ..., b20;
  intL low1, low2, ..., low20;
  if(b1 ) { low1  = high1 - high1; }
  else if(b2 ) { low2  = high2 - high2; }
  ...
  else if(b9 ) { low9  = high9 - high9; }
  else if(b10) { low10 = high10; }
  else if(b11) { low11 = high11 - high11; }
  ...
  else if(b20) { low20 = high20 - high20; }
}
```

Figure 10. A Program for Experiment (expr4.c)

which sends the arguments to the web browsers of clients. As the table shows, our method could find a counterexample faster than the other methods, although ours was not significantly faster than Terauchi and Aiken's method. That seems to be because our system duplicated the body of while-loops by applying rule P-WHILE, P-WHILE-IMP, and Q-WHILE, and the overhead caused by that duplication canceled the gain obtained by the path pruning. If closer cooperation with a model checker discussed in Section 7 were available, our method would be more effective for while-statements.

6. Related Work

Most closely related is the work by Terauchi and Aiken [18]. They use type information to optimize the model-checking-based information flow analysis [1], but in a more limited manner than ours. The idea of their optimization is to find expressions whose values do not depend on high-security values using type information, and avoid duplication of the expressions when making the self-composition. For example, the naive self-composition [1] of `while` e `do` c yields `while` e `do` c; `while` $\xi(e)$ `do` $\xi(c)$ (where ξ renames each variable x with x'), while the optimized self-composition yields `while` e `do` $(c; \xi(c))$ when e does not depend on high-security values. This optimization avoids the duplication of the expression e and (more importantly) the while-loop. Unlike ours, however, their method does not use flow information (the suspicious information flow paths discussed in Section 3.1) obtained by type inference.

Type-based analysis for secure information flow is applied to various languages such as simple imperative languages [20, 21] and functional languages [4, 14, 15], object-oriented languages [9], and concurrent languages [6, 7, 17]. They are useful for conservatively proving that a program has secure information flow, but not at all for *disproving* it. Some systems based on such work [10, 16] provide some useful error messages, showing how secret data may possibly be leaked when a program is not well-typed. Unlike ours, however, they cannot confirm that a certain program is indeed unsafe, and generate a counterexample to show the violation of non-interference.

7. Future Work

We discuss some future work in this section.

7.1 Compositional Analysis

We expect that we can make our algorithm more efficient by introducing compositional analysis. Recall that a suspicious information flow path $h \cdot \tau_1 \cdot \ldots \cdot \tau_n \cdot l$ indicates that information about the variable h may flow to l through $\tau_1, \ldots, \tau_n$. That implies, if a flow from τ_i to τ_{i+1} does not actually occur, then the whole information flow path $h \cdot \tau_1 \cdot \ldots \cdot \tau_n \cdot l$ is superfluous. This observation suggests us to inspect a suspicious information flow path in a compositional manner: to find a counterexample corresponding to the information flow path $p_1 p_2$, we can first search counterexamples corresponding to *partial paths* p_1 and p_2, and combine them (and apply this recursively). This compositional analysis is, for example, effective for the program `expr3.c` discussed in Section 5. From type inference, we obtain a suspicious information flow path $h \cdot x \cdot l$ (*pc* labels are omitted). In order to check whether there is indeed a flow from h to x, we only need to inspect the if-statement. Since the if-statement actually does not leak information of h to x, we can immediately conclude that `expr3.c` is safe, without ever analyzing the two while-loops.

Actually, we have already implemented a restricted form of compositional analysis sketched below (although that feature has not been used in the experiment described in Section 5). Consider the following program:

$$x := h - h; \cdots; l := x$$

From type inference, we obtain a suspicious information flow path $h \cdot x \cdot l$. Since the possible flow from h to x is inferred from the assignment $x := h - h$, if there is indeed such a flow, the values of x and x' in the self-composition should differ after the

assignment. Thus, we can insert additional `assume` statement into the self-composition:

$$\begin{array}{l}\texttt{assume}(x = x' \text{ and } l = l' \text{ and } \cdots);\\ x := h - h;\ x' := h' - h';\ \texttt{assume}(x \neq x');\\ \quad\vdots\\ l := x;\ l' := x';\\ \texttt{assert}(l = l')\end{array}$$

The insertion of $\texttt{assume}(x \neq x')$ enables a model checker to conclude that no assertion violation occurs, without looking at the code after $\texttt{assume}(x \neq x')$. (Note that the `assert` statement is never reached since $x \neq x'$ does not hold.)

7.2 Closer Cooperation with a Model Checker

As the experiment in Section 5 shows, our method is sometimes less efficient than the previous method [18]. The main source of the overhead of our current system is that the system generates a doubled program and invokes a model checker for it repeatedly, once for each suspicious information flow path, until a counterexample is found. Moreover, each doubled program may contain duplicated copies of while-loops. Thus, some part of the original program is analyzed by a model checker repeatedly. It seems possible to avoid the duplicated computation to some extent by generating one doubled program for multiple suspicious flow paths, but closer cooperation with a model checker seems necessary to avoid the duplicated computation completely. More cooperation with a model checker seems also necessary for realizing the compositional analysis discussed above.

8. Conclusion

We have formalized and implemented a novel method of combining type-based analysis and model checking to construct counterexamples against non-interference. The result of preliminary experiments shows that our method can often find counterexamples faster than previous methods based on model checking.

Acknowledgments

We thank Tachio Terauchi and Jun Furuse for many useful discussions. We will also like to thank anonymous referees for useful comments.

References

[1] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW '04: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 100–114, Washington, DC, USA, 2004. IEEE Computer Society.

[2] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 232–244, New York, NY, USA, 2003. ACM Press.

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

[4] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[5] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM Press.

[6] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–92, New York, NY, USA, 2002. ACM Press.

[7] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.

[8] P. Li and S. Zdancewic. Practical information flow control in web-based information systems. In *CSFW '05: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 2–15, June 2005. 20-22.

[9] A. 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, New York, NY, USA, 1999. ACM Press.

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

[11] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[12] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, New York, NY, USA, 2002. ACM Press.

[13] Y. Oiwa, S. Tatsurou, S. Eijiro, and Y. Akinori. Fail-safe ANSI-C compiler: An approach to making C programs secure: Progress report. In *In International Symposium on Software Security*, number 2609 in LNCS, pages 133–153. Springer-Verlag, 2002.

[14] F. Pottier and S. Conchon. Information flow inference for free. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 46–57, New York, NY, USA, 2000. ACM Press.

[15] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, New York, NY, USA, 2002. ACM Press.

[16] V. Simonet. Flow Caml in a nutshell. In G. Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, March 2003.

[17] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 355–364, New York, NY, USA, 1998. ACM Press.

[18] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *In Proceedings of the 12th International Static Analysis Symposium*, September 2005.

[19] H. Unno, N. Kobayashi, and A. Yonezawa. Combining type-based analysis and model checking for finding counterexamples against non-interference (Full version), February 2006. Available from `http://web.yl.is.s.u-tokyo.ac.jp/~uhiro/`.

[20] D. M. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[21] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK, 1997. Springer-Verlag.