17-3-R07

집합 제한식을 이용한 프로시저-내 정보흐름 안전성 분석

(Set Constraints Based Intra-procedural Analysis of Secure Information Flow)

임 흥 태*, 신 승 철**, 도 경 구*

*한양대학교 컴퓨터공학과, **동양대학교 컴퓨터공학부

*{htlim, doh}@cse.hanyang.ac.kr, **shin@dyu.ac.kr

요 약

이 논문은 명령형 프로그램의 프로시저 내부에 대한 정보흐름의 안전성을 집합 제한식 분석법을 사용하여 예측하는 방법을 제시한다. 지금까지 제안된 분석 기법은 정보흐름이 안전한 프로그램을 안전하지 않다고 보수적으로 판정한다는 점에서 정밀도가 떨어지는 경우가 많이 있다. 본 논문에서는 이전의 타입 시스템을 이용한 접근 방법보다는 분석 결과가 더 정밀한 새로운 분석법을 제안한다.

1. 서론

프로그램에서 정보흐름(information flow)의 안전성을 정적으로 결정하는 문제는 70년대 Denning 부부의 선구적인 논문[1,2]에서 제안된 이후 수십년 동안 연구되어 왔다. 정보흐름의 안전성이 보장되려면 불간섭 성질(non-interference property)이 성립해야 하는데, 이는 프로그램의 실행을 통해서 기밀 정보가 허가되지 않은 대상에 누출되지 않음을 의미한다. 이 문제를 풀기 위해서 다양한 정적 분석기법을 사용해왔다[3,4,5,6]. 이러한 분석 방법들은 정보흐름이 안전하지 않은(insecure) 프로그램에 대해서 "안전하다(secure)"고 틀린 분석 결과를 절대로 내주지 않는다는 측면에서 대부분 분석의 안전함(soundness)이 증명되었다. 그러나 Denning 부부의 방법[2], Volpano와 Smith의 타입시스템을 기반으로 하는 방법[5]은 너무 보수적인 경향이 있어서, 정보흐름이 안전한 프로그램을 안전하지 않다고 판정하는 경우가 많아 분석의 정밀성이 떨어진다.

이 논문에서는 명령형 프로그램의 프로시저 내부를 대상으로 집합 제한식의 표준 기법[7]을 사용하여 정보흐름의 안전성을 예측하는 문제를 풀어본다. 이 논문에서 제시한 분석방법은 위에서 열거한 보수적인 방법보다 더 정밀하도록 설계되었다.

본 논문의 구성은 다음과 같다. 2절에서는 간단한 예제를 통해서 정보흐름의 보안성 문제를 약식으로 설명하고, 3절에서는 이 논문의 대상언어인 While 언어의 구문과 의미를 정의하고, 4절에서는 흐름그래프 구축 방법과 정보누출을 탐지하는 집합 제한식의 생성과 그 제한식을 푸는 알고리즘을 제시하고, 안전성 검사의 실질적인 예를 보여준다. 마지막으로 5절에서는 결론을 맺고 추후 연구 방향을 제시한다.

2. 정보흐름의 안전성

이 논문에서 다루는 명령형 프로그램에 대한 정보흐름의 안전성 분석 문제를 다음과 같이 약식으로 설명할 수 있다. 변수가 비밀변수(높은 보안등급을 가진 변수)와 공개변수(낮은 보

안등급을 가진 변수)로 구별되어 있는 프로그램에서, 공개변수의 최종 값을 보고 비밀변수의 초기에 저장된 값에 관련된 어떤 정보를 알아낼 수 있는지 여부를 검사하는 문제를 정보흐름의 안전성 분석 문제라고 한다. 즉, 공개변수의 실행전 값과 실행후 값을 보고 비밀변수의 실행전 값에 관한 정보를 절대로 알아낼 수 없는 경우, "정보흐름은 안전이 보장된다."라고 한다.

예를 들어, x를 비밀변수, m과 n을 공개변수라고 할 때, 다음 배정문의 정보흐름은 안전하지 않다.

$$m := x$$

변수 m에 저장된 값을 보고 x에 저장되어 있는 값을 알 수 있기 때문이다. 이러한 경우 변수 x에서 변수 m으로 명시적 정보 누출(explicit information leak)이 일어났다고 한다. 반면에 다음 배정문의 정보흐름은 안전하다고 한다.

$$x := m$$

m값을 알더라도 x에 저장된 값에 관련된 어떠한 정보도 알아낼 수 없기 때문이다. 다음 조건문을 생각해보자. 각 분기에 속한 문장만 따져보면 모두 정보흐름이 안전하지만, 조건문 전체적으로 보면 안전하다고 할 수 없다.

$$\text{if } x < 1 \text{ then } m := 0 \text{ else } m := 1$$

이 조건문에서 변수 x에 저장된 값이 변수 m에 그대로 저장되지는 않지만, 변수 m에 저장된 값을 보고 변수 x에 저장된 값에 대한 정보를 유추해낼 수 있다. 이러한 경우 변수 x에서 변수 m으로 묵시적 정보 누출(implicit information leak)이 일어났다고 한다. 이 두 가지 유형의 정보 누출을 즉시 누출(immediate leak)이라고 한다. 정보는 전이적으로 누출될 수도 있다. 예를 들어, 다음 프로그램에서

$$m := x;$$
$$n := m;$$

변수 x에 저장된 값은 변수 m뿐 아니라 변수 n에 저장된 값을 보면 알 수 있다. 즉, 변수 x에서 변수 n으로의 정보 누출이 변수 m을 통하여 발생한다. 이러한 누출을 전이 누출(transitive leak)이라고 한다.

누출된 정보는 복구되기도 한다. 예를 들어, 다음 프로그램의 정보흐름은 안전하다.

$$m := x;$$
$$m := n;$$

왜냐하면, 프로그램 실행이 끝난 후 변수 m에 변수 x의 값이 저장되어 있지 않기 때문이다. 이러한 상황을 "변수 x에서 변수 m으로의 정보 누출이 전방향으로 복구되었다(forwardly recovered)"라고 한다. 다음 프로그램도 안전하다.

$$x := m;$$
$$n := x;$$

왜냐하면, 프로그램 실행 전에 변수 x에 저장되어 있던 값은 변수 m에 저장되어 있던 공개값으로 변경되었기 때문이다. 이러한 상황을 "변수 x에서 변수 n으로의 정보 누출이 후방향으로 복구되었다(backwardly recovered)"라고 한다. 따라서 정보누출이 전혀 발생하지 않거나, 발생했더라도 모두 전방향 또는 후방향으로 복구된다면, 프로그램의 정보흐름은 안전하다고 주장할 수 있다.

Denning 부부의 방법과 Volpano와 Smith의 타입시스템을 기반으로 한 방법은 위에서 언급한 정보 누출의 복구를 감지할 수 없고, 따라서 바로 위의 두 예제 프로그램에 대해서 정보흐름이 안전하지 않다고 판정한다는 점에서 분석의 정밀도가 낮다. 하지만, 이 논문에서 소개하는 집합 제한식 분석법은 마지막 두 예제 프로그램에 대해 정보 누출이 없고, 따라서 정보흐

름이 안전하다고 판정한다.

3. While 언어의 정의

간단한 명령형 언어 While의 구문구조(syntax)와 의미구조(semantics)를 정의해 보자. While 프로그램의 문장과 조건식을 인식하기 위해서 각 배정문(assignment), skip 문장, 조건식마다 유일한 번호 라벨을 부여한다.

구문 도메인(Syntax Domain) :

$a \in$ AExp 산술식(arithmetic expressions)
$b \in$ BExp 논리식(boolean expressions)
$S \in$ Stmt 문장(statements)
$x \in$ Var 변수(variables)
$l \in$ Lab 라벨(labels)

추상 구문 구조(Abstract Syntax) :

$S ::= [x := a]^l \mid [\text{skip}]^l \mid S_1; S_2 \mid$
if $[b]^l$ then S_1 else $S_2 \mid$
while $[b]^l$ do S

변이규칙(Transition) :

상태(state)는 다음과 같이 변수가 주어지면 변수에 저장되어 있는 값인 정수를 내주는 함수로 정의된다.

$\sigma \in \text{State} = \text{Var} \rightarrow \text{Z}$

의미구조는 $<S, \sigma> \rightarrow \sigma'$와 같이 변이규칙으로 표현된다. 이 변이규칙의 의미는 "상태 σ에서 문장 S가 실행하여 종료하며 결과 상태는 σ'이다"가 된다.

의미구조(Semantics) :

산술식과 논리식에 대한 의미 함수는 다음과 같이 이미 정의되어 있다고 가정한다.

$A : \text{AExp} \rightarrow (\text{State} \rightarrow \text{Z})$
$B : \text{BExp} \rightarrow (\text{State} \rightarrow \text{T})$

여기서 Z는 정수의 집합이고, T는 진리값의 집합이다.

의미 구조는 다음과 같다.

[배정문] $< [x := a]^l, \sigma > \rightarrow \sigma[x \text{ a } A[[a]]\sigma]$

[skip문] $< [skip]^l, \sigma > \rightarrow \sigma$

[문장나열] $$\frac{< S_1, \sigma > \rightarrow \sigma_1 \quad < S_2, \sigma_1 > \rightarrow \sigma_2}{< S_1; S_2, \sigma > \rightarrow \sigma_2}$$

[조건문#1] $$\frac{B[[b]]\sigma = true \quad < S_1, \sigma > \rightarrow \sigma_1}{< if\ [b]^l\ then\ S_1\ else\ S_2, \sigma > \rightarrow \sigma_1}$$

[조건문#2] $$\frac{B[[b]]\sigma = false \quad < S_2, \sigma > \rightarrow \sigma_2}{< if\ [b]^l\ then\ S_1\ else\ S_2, \sigma > \rightarrow \sigma_2}$$

[반복문#1] $$\frac{B[[b]]\sigma = true \quad < S, \sigma > \rightarrow \sigma_1 \quad < while\ [b]^l\ do\ S, \sigma_1 > \rightarrow \sigma_2}{< while\ [b]^l\ do\ S, \sigma > \rightarrow \sigma_2}$$

[반복문#2] $$\frac{B[[b]]\sigma = false}{< while\ [b]^l\ do\ S, \sigma > \rightarrow \sigma}$$

4. 정보 누출 분석

이 절에서는 집합 제한식 분석법을 사용하여 정보 누출을 찾아내는 분석법을 살펴본다. 정보 누출 분석을 위해서 While 프로그램을 흐름그래프(flow graph)로 변환한 후, 이 흐름그래프를 기초로 하여 정보 누출을 탐지하는 집합 제한식을 제시하고 생성된 집합 제한식의 해를 구하는 방법을 알아본다.

4.1. 흐름그래프

흐름그래프(flow graph)는 Nielson 부부와 Hankin의 책에서 표현한 방법과 비슷하게 정의한다[8]. 흐름그래프에는 정상적인 제어흐름(control flow)뿐만 아니라, 조건식 블록에서 조건문의 분기 또는 루프 몸체에 속해있는 각 문장 블록을 잇는 묵시적 흐름(implicit flow)도 포함되어 있다. 묵시적 흐름은 묵시적으로 일어나는 정보 누출을 감지하기 위해서 사용된다.

흐름그래프는 기본블록의 집합과 블록 사이를 잇는 (제어와 묵시적) 흐름의 집합으로 구성된다. 좀 더 정식으로 정의하면, While 문장 S

의 흐름그래프는 다음과 같이 다섯 겹으로 구성된다.

$$flowgraph(S) = (blocks(S),\ flow(S),\ flow_I(S),\ init(S),\ final(S))$$

여기에서 각 함수는 아래와 같이 정의된다.

Blocks를 $[x:=a]^l$ 또는 $[\text{skip}]^l$ 또는 $[b]^l$의 형태로 된 기본블록의 집합이라고 하자. 그러면 함수 *blocks*는 각 문장에 속해있는 기본 블록의 집합을 내주며, 다음과 같이 정의된다.

$$\begin{aligned}
&blocks : \text{Stmt} \to \mathcal{P}(\text{Blocks})\\
&blocks([x:=a]^l) = \{[x:=a]^l\}\\
&\quad blocks([\text{skip}]^l) = \{[\text{skip}]^l\}\\
&\quad blocks(S_1;S_2) = blocks(S_1) \cup blocks(S_2)\\
&\quad blocks(\text{if } [b]^l \text{ then } S_1 \text{ else } S_2) =\\
&\qquad \{[b]^l\} \cup blocks(S_1) \cup blocks(S_2)\\
&\quad blocks(\text{while } [b]^l \text{ do } S) = \{[b]^l\} \cup blocks(S)
\end{aligned}$$

흐름그래프에서 시작 블록은 항상 하나뿐이지만, 마지막 블록은 조건문 때문에 하나 이상이 될 수 있다. 함수 *init*는 문장에서 시작 블록의 라벨을 찾아주며, 다음과 같이 정의된다.

$$\begin{aligned}
&init : \text{Stmt} \to \text{Lab}\\
&init([x:=a]^l) = l\\
&\quad init([\text{skip}]^l) = l\\
&\quad init(S_1;S_2) = init(S_1)\\
&\quad init(\text{if } [b]^l \text{ then } S_1 \text{ else } S_2) = l\\
&\quad init(\text{while } [b]^l \text{ do } S) = l
\end{aligned}$$

함수 *final*은 문장에서 마지막 블록의 집합을 내주며, 다음과 같이 정의된다.

$$\begin{aligned}
&final : \text{Stmt} \to \mathcal{P}(\text{Lab})\\
&final([x:=a]^l) = \{l\}\\
&\quad final([\text{skip}]^l) = \{l\}\\
&\quad final(S_1;S_2) = final(S_2)\\
&\quad final(\text{if } [b]^l \text{ then } S_1 \text{ else } S_2) =\\
&\qquad final(S_1) \cup final(S_2)\\
&\quad final(\text{while } [b]^l \text{ do } S) = \{l\}
\end{aligned}$$

함수 *flow*는 문장에 있는 블록들을 잇는 모든 제어흐름의 집합을 내주며, 다음과 같이 정의된다.

$$\begin{aligned}
&flow : \text{Stmt} \to \mathcal{P}(\text{Lab}\times\text{Lab})\\
&flow([x:=a]^l) = \varnothing\\
&\quad flow([\text{skip}]^l) = \varnothing\\
&\quad flow(S_1;S_2) = flow(S_1) \cup flow(S_2)\\
&\qquad \cup \{(l,\ init(S_2)) \mid l \in final(S_1)\}\\
&\quad flow(\text{if } [b]^l \text{ then } S_1 \text{ else } S_2) =\\
&\qquad flow(S_1) \cup flow(S_2)\\
&\qquad \cup \{(l,\ init(S_1)),\ (l,\ init(S_2))\}\\
&\quad flow(\text{while } [b]^l \text{ do } S) =\\
&\qquad flow(S) \cup \{(l,\ init(S))\}\\
&\qquad \cup \{(l',\ l) \mid l' \in final(S)\}
\end{aligned}$$

다음 함수 $flow_I$는 문장에 있는 묵시적 흐름을 정의한다.

$$\begin{aligned}
&flow_I : \text{Stmt} \to \mathcal{P}(\text{Lab}\times\text{Lab})\\
&flow_I([x:=a]^l) = \varnothing\\
&\quad flow_I([\text{skip}]^l) = \varnothing\\
&\quad flow_I(S_1;S_2) = flow_I(S_1) \cup flow_I(S_2)\\
&\quad flow_I(\text{if } [b]^l \text{ then } S_1 \text{ else } S_2) =\\
&\qquad flow_I(S_1) \cup flow_I(S_2)\\
&\qquad \cup \{(l,\ l') \mid \mathrm{B}^{l'} \in blocks(S_1) \cup blocks(S_2)\}\\
&\quad flow_I(\text{while } [b]^l \text{ do } S) =\\
&\qquad flow_I(S) \cup \{(l,\ l') \mid \mathrm{B}^{l'} \in blocks(S)\}
\end{aligned}$$

예를 들어 다음 프로그램을 보자.

$$\begin{aligned}
S = {} & [z:=1]^1;\\
& \text{while } [x>0]^2 \text{ do}\\
& \quad ([z:=z*y]^3;\ [x:=x-1]^4)
\end{aligned}$$

위 프로그램 *S*의 흐름그래프는 다음과 같다.

$$\begin{aligned}
&flowgraph(S) = (blocks(S),\ flow(S),\ flow_I(S),\ init(S),\ final(S))\\
&= (\{[z:=1]^1,\ [x>0]^2,\ [z:=z*y]^3,\ [x:=x-1]^4\},\\
&\quad \{(1,2),\ (2,3),\ (3,4),\ (4,2)\},\\
&\quad \{(2,3),\ (2,4)\},\\
&\quad 1,\\
&\quad \{2\})
\end{aligned}$$

4.2. 집합 제한식

집합 제한식(Set constraints)은 프로그램 변수를 관심있는 값의 집합으로 다루는 것에 기반한 프로그램 분석 방법이다. 집합 제한식을 이용한 프로그램 분석 방법은 주어진 프로그램의 실행 의미를 근사적으로 표현하는 집합 제한식을 생성해 낸 후, 이 제한식의 해를 구하여 프로그램의 분석 결과를 도출해내는 것이다.

본 논문에서는 이러한 집합 제한식 분석법을 사용하여 명령형 프로그램의 핵심 부분에 대한 정보흐름의 안전성 분석 문제를 해결한다. 정보흐름의 안전성 분석을 위하여 필요한 프로그램의 실행 의미는 각 기본블록에서 나타나는 모든 변수들의 보안등급이며, 집합변수는 보안등급의 집합으로 나타낸다. 따라서 프로그램의 각 기본블록에서 변수들의 보안등급 상태를 집합 변수들간의 집합 제한식으로 생성하고, 이 제한식의 해를 구하여 정보흐름의 안전성 여부를 검사할 수 있다.

4.3. 집합 제한식의 생성

프로그램 내의 모든 변수들은 실행 전에 이미 보안등급이 정해져 있다고 가정한다. 보안등급은 부분적으로 순서있는 집합(partially ordered set)인 Sec = ({L, H}, ≤)을 이룬다. 여기서 L은 낮은 보안등급을, H는 높은 보안등급을 나타내고, 이들 사이에 L < H의 순서가 형성되며, 이는 H가 L보다 보안등급이 절대적으로 높음을 의미한다.

집합 변수(Set variable)는 프로그램 각 지점에서 각 변수에 저장될 수 있는 값의 보안등급을 모아놓은 것이다. 블록 l에서 변수 x의 보안등급은 집합 변수 x_l로 나타내고, 이것은 블록 l의 실행 직후 변수 x의 가능한 보안등급을 모두 모아놓은 집합이다. 프로그램 실행 전 각 변수 x의 초기 보안등급은 집합변수 x_0로 표시하며 $x_0 \supseteq \{L\}$ 또는 $x_0 \supseteq \{H\}$의 보안등급을 갖는다.

집합 제한식은 집합 변수간의 관계를 나타내며, 기본블록 $[x:=a]^l$ 또는 $[skip]^l$ 또는 $[b]^l$의 형태로 된 프로그램 각 지점에서 $x_l \supseteq y_k$의 형태로 생성된다. 이것의 의미는 "블록 l을 실행한 후 변수 x의 보안등급이 블록 k를 실행한 후의 변수 y의 보안등급에 영향을 받는다"가 된다.

While 프로그램의 각 기본블록에서의 집합 제한식 생성 규칙은 (그림 1)과 같다. 여기서 $vars(S)$는 문장 S에 포함된 모든 변수들의 집합을 나타내며, $FV(e)$는 표현식 $e(\in AExp \cup BExp)$에 속한 변수들의 집합이다. 또한 $flow(S)$에는 프로그램 입구 지점에서의 집합 제한식을 생성하기 위하여 기존 제어흐름에 추가로 $(0, init(S))$의 제어흐름이 포함되어 있다고 가정한다.

$$[Skip]\ \frac{[skip]^l \in blocks(S) \quad z \in vars(S) \quad (k,l) \in flow(S)}{z_l \supseteq z_k}$$

$$[Ass_{exp}]\ \frac{[x := a]^l \in blocks(S) \quad y \in FV(a) \quad z \in (vars(S) \setminus \{x\}) \quad (k,l) \in flow(S)}{x_l \supseteq y_k \ , \ z_l \supseteq z_k}$$

$$[Ass_{imp}]\ \frac{[x := a]^l \in blocks(S) \quad [b]^k \in block(S) \quad y \in FV(b) \quad (k,l) \in flow_I(S)}{x_l \supseteq y_k}$$

$$[Cond]\ \frac{[b]^l \in blocks(S) \quad z \in vars(S) \quad (k,l) \in flow(S)}{z_l \supseteq z_k}$$

(그림 1) 집합 제한식 생성 규칙

Rule(Skip) : skip문장을 실행한 후 변수들의 보안등급은 변하지 않으므로, 실행 후 집합 변수(z_l)는 실행 전 집합 변수(z_k)의 보안등급에 그대로 영향을 받는다.

Rule(Ass_{exp}) : 배정문의 명시적 정보흐름에 대한 집합 제한식을 생성한다. 배정문을 실행한 후 좌변의 변수(x)는 우변의 모든 변수들($FV(a)$)의 보안등급에 영향을 받으며, 그 외의 변수들(x를 제외한 변수)은 보안등급이 변하지 않는다.

Rule(Ass$_{imp}$) : 배정문의 묵시적 정보흐름에 대한 집합 제한식을 생성한다. 배정문이 if나 while문 조건식의 묵시적 흐름에 영향을 받는다면 좌변의 변수(x)는 조건식에 포함된 모든 변수들($FV(b)$)의 보안등급에 영향을 받는다.

Rule(Cond) : 조건식을 실행한 후 변수들의 보안등급은 변하지 않으므로, 실행 후 집합 변수는 실행 전 집합 변수의 보안등급에 그대로 영향을 받는다.

집합 제한식의 생성 예로서 While 프로그램의 다음 예를 생각해보자. 변수의 초기 보안등급은 $x_0 \supseteq \{H\}$, $m_0 \supseteq \{L\}$, $n_0 \supseteq \{L\}$ 라고 가정한다.

$$S \equiv [m := x]^1; [m := n]^2$$

먼저, 프로그램 S의 실행 후 변수 m의 정보 누출 여부를 직관적으로 판단해보자. 블록 1을 실행한 후에는 x에서 m으로의 명시적 누출이 발생한다. 하지만 블록 2의 실행 후에 m의 보안등급은 n의 보안등급인 L로 바뀌므로 m은 더이상 정보누출이 발생하지 않는다. 따라서 문장 S를 실행한 후에 변수 m의 정보흐름은 안전성이 보장된다고 할 수 있다.

이제 (그림 1)에 따라 문장 S의 집합 제한식 생성 과정을 살펴보자. 블록 1에서는 *Rule (Ass$_{exp}$)*에 따라 다음과 같은 집합 제한식이 생성되고,

$$m_1 \supseteq x_0$$
$$x_1 \supseteq x_0$$
$$n_1 \supseteq n_0$$

블록 2에서는 *Rule(Ass$_{exp}$)*에 따라 다음과 같은 집합 제한식이 생성된다.

$$m_2 \supseteq n_1$$
$$n_2 \supseteq n_1$$
$$x_2 \supseteq x_1$$

위 집합 제한식으로부터 변수 m의 정보 누출 여부를 생각해보자.

$m_2 \supseteq n_1$과 $n_1 \supseteq n_0$를 가지고 $m_2 \supseteq n_0$를 유추해 낼 수 있고, $n_0 \supseteq \{L\}$이므로 결국 $m_2 \supseteq \{L\}$이다. 따라서 프로그램 실행 후 변수 m의 보안등급은 L이 된다. 결과적으로 문장 S를 실행한 후에 변수 m과 관련된 정보흐름은 안전성이 보장된다고 할 수 있다.

4.4. 집합 제한식의 해

집합 제한식의 해를 구한다는 것은 프로그램의 흐름그래프에서 생성된 모든 집합 제한식을 이용하여 각 블록에서의 집합 변수의 보안등급(L 또는 H)을 모두 구하는 것을 의미한다. 따라서 해를 구한 이후에는 프로그램의 어떠한 블록 지점에서도 임의의 변수에 대한 보안등급을 알 수 있으므로, 초기 보안등급이 L인 임의의 변수 x의 정보흐름 안전성은 프로그램 출구에서 집합변수 $x_l(l \in final(S))$의 보안등급을 전부 구하여 최소상한(least upper bound)을 고려하여 판단할 수 있다. 이때 출구에서의 보안등급이 L이라면 변수 x의 정보흐름은 안전하다고 하고, 반면에 출구에서의 보안등급이 H라면 변수 x에서 정보 누출이 발생하였다고 할 수 있다.

집합 제한식의 해를 구하는 알고리즘은 다음과 같다.

```
fun solveSetConstraints(SC : Set Constraints,
                        SV_old : Set Variables)
{   SV_new = updateSV(SC, SV_old)
    if (SV_new = SV_old) then
        return(SV_old)
    else
    {   SV_old = SV_new
        solveSetConstraints(SC, SV_old)
    }
}
```

함수의 입력 SC는 프로그램에서 생성된 집합 제한식의 집합을 나타내고, SV는 집합 변수들의 해를 나타낸다. updateSV 함수는 입력된

SV_{old}를 통해서 집합 제한식 SC의 구할 수 있는 모든 집합 변수들의 해 SV_{new}를 구하여 새로운 집합 제한식의 해를 내주는 함수이다. SV_{new}와 SV_{old}가 같다는 의미는 해의 고정점을 구했다는 의미이며, 그때의 SV_{old}가 구하려는 집합 제한식의 해가 된다.

해를 구하는 알고리즘을 적용하기 전에는 각 변수 x의 초기 보안등급을 나타내는 x_0 형태의 집합 변수들의 보안등급만 알 수 있고, 그 외 기본블록에서 생성되는 집합 변수의 보안등급은 알 수 없으므로 $\perp$으로 취급한다. 앞으로 본 논문에서는 이러한 초기 집합 제한식의 해를 SV_0라고 하고, 알고리즘에서 함수를 k번 호출한 후의 새로운 집합 제한식의 해를 SV_k라고 한다.

다음 예제 프로그램에서 알고리즘을 적용하여 집합 제한식의 해를 구하는 과정을 살펴보자.

$$S \equiv [m := x]^1; [m := n]^2$$

4.3절에서 살펴본 바와 같이 프로그램 S의 생성된 집합 제한식은 다음과 같다.

$$\begin{aligned} SC = \{&m_1 \supseteq x_0,\ x_1 \supseteq x_0,\ n_1 \supseteq n_0,\\ &m_2 \supseteq n_1,\ n_2 \supseteq n_1,\ x_2 \supseteq x_1\} \end{aligned}$$

알고리즘을 적용하기 전에는 다음과 같이 변수의 초기 보안등급만 알 수 있고,

$$\begin{aligned} SV_0 = \{&(m_0,\ \mathrm{L}),\ (n_0,\ \mathrm{L}),\ (x_0,\ \mathrm{H}),\\ &(m_1,\ \perp),\ (n_1,\ \perp),\ (x_1,\ \perp),\\ &(m_2,\ \perp),\ (n_2,\ \perp),\ (x_2,\ \perp)\} \end{aligned}$$

SC와 SV_0를 입력값으로 알고리즘에 적용하면 SC의 정보 $\{m_1 \supseteq x_0,\ x_1 \supseteq x_0,\ n_1 \supseteq n_0\}$와 SV_0로부터 다음을 구할 수 있다.

$$\begin{aligned} SV_1 = \{&(m_0,\ \mathrm{L}),\ (n_0,\ \mathrm{L}),\ (x_0,\ \mathrm{H}),\\ &(m_1,\ \mathrm{H}),\ (n_1,\ \mathrm{L}),\ (x_1,\ \mathrm{H}),\\ &(m_2,\ \perp),\ (n_2,\ \perp),\ (x_2,\ \perp)\} \end{aligned}$$

SV_1의 해가 SV_0의 해와 같지 않기 때문에 한 번 더 함수를 적용하면,

$$\begin{aligned} SV_2 = \{&(m_0,\ \mathrm{L}),\ (n_0,\ \mathrm{L}),\ (x_0,\ \mathrm{H}),\\ &(m_1,\ \mathrm{H}),\ (n_1,\ \mathrm{L}),\ (x_1,\ \mathrm{H}),\\ &(m_2,\ \mathrm{L}),\ (n_2,\ \mathrm{L}),\ (x_2,\ \mathrm{H})\} \end{aligned}$$

같은 이유로, SV_2의 해가 SV_1의 해와 같지 않기 때문에 한번 더 함수를 적용하면,

$$\begin{aligned} SV_3 = \{&(m_0,\ \mathrm{L}),\ (n_0,\ \mathrm{L}),\ (x_0,\ \mathrm{H}),\\ &(m_1,\ \mathrm{H}),\ (n_1,\ \mathrm{L}),\ (x_1,\ \mathrm{H}),\\ &(m_2,\ \mathrm{L}),\ (n_2,\ \mathrm{L}),\ (x_2,\ \mathrm{H})\} \end{aligned}$$

이때 SV_3와 SV_2의 결과가 같기 때문에 SV_2가 구하려는 해의 고정점이라 할 수 있으며, 집합 제한식의 해가 된다. 프로그램 실행 후 m의 정보 누출 여부를 검사하면, SV_2를 통해서 프로그램의 출구 지점에서 집합 변수 m_2의 보안등급이 L임을 알 수 있다. 따라서 변수 m의 정보흐름은 안전성이 보장된다고 할 수 있다.

4.5. 분석의 예

다음 While 프로그램을 분석해 본다.

```
while [(x <= 3)]^1
    do { if [p = g]^2 then [f := 1]^3
                      else [f := 0]^4
         [x := x + 1]^5;
         [g := g + 10]^6;
       }
       [f:=2]^7;
       [x:=0]^8
```

초기 제한식은 $p_0 \supseteq \{\mathrm{H}\}$이고 $f_0,\ g_0,\ x_0 \supseteq \{\mathrm{L}\}$이라고 가정하면 다음과 같은 집합 제한식이 생성된다.

$$f_1 \supseteq f_0 \cup f_6$$
$$g_1 \supseteq g_0 \cup g_6$$

$p_1 \supseteq p_0 \cup p_6$
$x_1 \supseteq x_0 \cup x_6$
$f_2 \supseteq f_1$
$g_2 \supseteq g_1$
$p_2 \supseteq p_1$
$x_2 \supseteq x_1$
$f_3 \supseteq x_1 \cup p_2 \cup g_2$
$g_3 \supseteq g_2$
$p_3 \supseteq p_2$
$x_3 \supseteq x_2$
$f_4 \supseteq x_1 \cup p_2 \cup g_2$
$g_4 \supseteq g_2$
$p_4 \supseteq p_2$
$x_4 \supseteq x_2$
$f_5 \supseteq f_3 \cup f_4$
$g_5 \supseteq g_3 \cup g_4$
$p_5 \supseteq p_3 \cup p_4$
$x_5 \supseteq x_3 \cup x_4 \cup x_1$
$f_6 \supseteq f_5$
$g_6 \supseteq g_5 \cup x_1$
$p_6 \supseteq p_5$
$x_6 \supseteq x_5$
$f_7 \supseteq \varnothing$
$g_7 \supseteq g_1$
$p_7 \supseteq p_1$
$x_7 \supseteq x_1$
$f_8 \supseteq f_7$
$g_8 \supseteq g_7$
$p_8 \supseteq p_7$
$x_8 \supseteq \varnothing$

집합 제한식의 해를 구하는 절차를 표로 나타내면 다음과 같다.

Lab	Var	SV								
		SV_0	SV_1	SV_2	SV_3	SV_4	SV_5	SV_6	SV_7	SV_8
0	f	L	L	L	L	L	L	L	L	L
	g	L	L	L	L	L	L	L	L	L
	p	H	H	H	H	H	H	H	H	H
	x	L	L	L	L	L	L	L	L	L
1	f	⊥	L	L	L	L	L	H	H	H
	g	⊥	L	L	L	L	L	L	L	L
	p	⊥	H	H	H	H	H	H	H	H
	x	⊥	L	L	L	L	L	L	L	L
2	f	⊥	⊥	L	L	L	L	L	H	H
	g	⊥	⊥	L	L	L	L	L	L	L
	p	⊥	⊥	H	H	H	H	H	H	H
	x	⊥	⊥	L	L	L	L	L	L	L
3	f	⊥	⊥	L	H	H	H	H	H	H
	g	⊥	⊥	⊥	L	L	L	L	L	L
	p	⊥	⊥	⊥	H	H	H	H	H	H
	x	⊥	⊥	⊥	L	L	L	L	L	L
4	f	⊥	⊥	L	H	H	H	H	H	H
	g	⊥	⊥	⊥	L	L	L	L	L	L
	p	⊥	⊥	⊥	H	H	H	H	H	H
	x	⊥	⊥	⊥	L	L	L	L	L	L
5	f	⊥	⊥	⊥	L	H	H	H	H	H
	g	⊥	⊥	⊥	⊥	L	L	L	L	L
	p	⊥	⊥	⊥	⊥	H	H	H	H	H
	x	⊥	⊥	L	L	L	L	L	L	L
6	f	⊥	⊥	⊥	⊥	L	H	H	H	H
	g	⊥	⊥	L	L	L	L	L	L	L
	p	⊥	⊥	⊥	⊥	⊥	H	H	H	H
	x	⊥	⊥	⊥	L	L	L	L	L	L
7	f	⊥	L	L	L	L	L	L	L	L
	g	⊥	⊥	L	L	L	L	L	L	L
	p	⊥	⊥	⊥	H	H	H	H	H	H
	x	⊥	⊥	L	L	L	L	L	L	L
8	f	⊥	⊥	L	L	L	L	L	L	L
	g	⊥	⊥	⊥	L	L	L	L	L	L
	p	⊥	⊥	⊥	H	H	H	H	H	H
	x	⊥	L	L	L	L	L	L	L	L

위 집합 제한식의 해를 분석해보면, 프로그램의 출구에서 $f_8 \supseteq \{L\}$ 이므로 변수 f의 정보흐름은 안전하다고 할 수 있다.

5. 결론 및 향후 과제

본 논문에서는 집합 제한식 분석을 통하여 명령형 프로그램의 핵심 부분에 대한 정보흐름의 안전성 문제를 푸는 해법을 제시하였다. 이 논문에서 제안한 분석법은 Denning 부부의 방법과 타입시스템을 기반한 방법과 비교해서 더 정밀하며, 이 기법을 Standard ML로 구현하였

다. 본 논문에서 제안한 분석방법의 안전성(Soundness) 증명은 향후 과제로 남겨둔다.

본 논문에서 제안한 방식에 따르면, 프로그램에서 블록의 개수가 m, 변수의 개수가 n이라고 할때, m×n개의 집합변수와 집합제약식이 생성된다. 분석의 효율성 향상을 위해서 생성되는 집합변수와 집합제약식의 개수를 줄일 필요가 있는데, 이것 또한 향후 과제로 남겨둔다.

참고문헌

[1] D.E. Denning, A lattice model of secure information flow, Communication of the ACM, 19(5):236-243, 1976.

[2] D.E. Denning and P.J. Denning, Certification of programs for secure information flow, Communication of the ACM, 20(7):504-512, 1977.

[3] G.R. Andrews and R.P. Reitman, An axiomatic approach to information flow in programs, ACM Transactions on Programming Languages and Systems, 2(1):56-76, 1980.

[4] J.-P. Banatre, C. Bryce, and D. Metayer, Compile-time detection of information flow in sequential programs, In D. Gollmann, editor, Computer Security - ESORICS'94, the 3rd European Symposium on Research in Computer Security, Lecture Notes in Computer Science, volume 875, pages 55-73, Springer-Verlag, 1994.

[5] D. Volpano and G. Smith, A type-based approach to program security, In TAPSOFT'97, the 7th international Conference on Theory and Practice of Software Development, Lecture Notes in Computer Science, pages 607-621, Springer-Verlag, 1997.

[6] D. Volpano and G. Smith and C. Irvine, A sound type system for secure information flow, Journal of Computer Security, 4:1-21, 2001.

[7] N. Heintze, Set constraints in program analysis, Proceedings of International Symposium on Logic Programming, 1993.

[8] F. Nielson, H.R. Neilson and C. Hankin, Principles of Program Analysis, Springer, 1999.

임 흥 태

2002년 2월 한양대학교 전자계산학과 졸업(학사)

2002년 3월 ~ 현재 한양대학교 컴퓨터공학과 (석사과정)

관심분야 : 프로그래밍언어, 프로그램 분석 및 검증

신 승 철

1987년 인하대학교 전자계산학과(학사)

1989년 인하대학교 전자계산학과(석사)

1996년 인하대학교 전자계산공학과(박사)

1999년 9월 ~ 2000년 9월 Postdoc Researcher, Kansas State University

1996년 ~ 현재 동양대학교 컴퓨터공학부 조교수

관심분야 : 프로그래밍언어, 프로그램 분석 및 검증, 정형기법, 이론 전산학

도 경 구

1980년 한양대학교 산업공학(학사)

1987년 미국 Iowa State University 전산학(석사)

1992년 미국 Kansas State University 전산학(박사)

1993년 4월 ~ 1995년 9월 일본 University of Aizu 교수

1995년 9월 ~ 현재 한양대학교 부교수

관심분야 : 프로그래밍언어, 프로그램 분석 및 검증, 의미론