17-3-R08

정보 흐름에 대한 SSA기반 분석

(SSA-based Analysis for Information-Flow)

최 성 권*, 신 승 철*, 도 경 구**

*동양대학교 컴퓨터공학과, **한양대학교 컴퓨터공학과

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

요 약

정보 흐름 분석은 주어진 프로그램에서 보안등급이 서로 다른 변수들 사이의 정보 흐름이 안전한가를 정적으로 검사한다. 부분타입을 포함하는 타입시스템을 이용하여 프로그램의 정보 흐름을 분석하면 정보 누출을 손쉽게 발견할 수 있지만 정보 누출이 전후 방향으로 복구되는 상황을 인지하지 못하기 때문에 너무 많은 안전한 프로그램을 거절하는 단점이 있다. 이것은 타입 추론의 결과가 프로그램의 흐름에 민감하지 않기 때문인데, 본 논문에서는 주어진 프로그램을 SSA 형태(static single assignment form)로 변환한 후에 타입시스템을 적용함으로써 이 문제를 해결하는 방법을 설명한다. 본 논문의 방법은 정보 누출의 복구되는 경우에도 정확하게 분석하기 때문에 요약 해석법과 같은 정도의 정밀도를 갖는다.

1. 서론

오늘날 컴퓨팅 및 인터넷 환경에서는 신뢰할 수 없는 사이트로부터 신뢰할 수 없는 프로그램을 내려받아 설치하고 실행하는 경우가 날로 확산되고 있다. 인터넷을 통해 내려받은 프로그램이 사용자의 의도와 관계없이 사용자의 주소록 정보와 같은 비밀정보를 인터넷 등을 통하여 외부로 유출할 수도 있다. 그러나 현재 이런 정보 유출이 발생할 수 있는지를 미리 검사하는 도구는 사용되고 있지 않다.

주어진 프로그램이 사용자의 비밀정보를 다루더라도 이것의 일부 또는 전체를 프로그램 종료 후에 공개되는 변수, 파일, 인터넷 등으로 유출하지 않는다는 것을 확인하는 문제를 정보 흐름 보안성 문제라고 한다. 이러한 문제가 DePaul 등에 의해 제시된 후, 이를 정적으로 결정하고자 하는 시도가 70년대 Denning 부부의 선구적인 논문[1,2]에서 제안되었고 오랫동안 연구되어 왔다.

이 문제를 구문적으로 푸는 방법으로는 데이터 흐름 분석(data flow analysis)을 통한 방법[3,4,5], 타입시스템(type systems)을 기반으로 하는 방법 [6,7] 등이 있다. 이러한 방법들은 정보 흐름이 안전하지 않은(insecure) 프로그램에 대해서 "안전하다(secure)"고 틀린 분석결과를 절대로 내주지 않는다는 측면에서 대부분 분석의 안전함(soundness)이 증명되었다. 그러나 Denning 부부의 방법[2], Mizuno와 Schmidt의 접근방법[4], Volpano와 Smith의 타입시스템을 기반으로 하는 방법[6]은 모두 너무 보수적인 경향이 있어서, 정보흐름이 안전한 프로그램을 안전하지 않다고 판정하는 경우가 많아 분석의 정밀성이 떨어진다.

이에 비해서 프로그램 의미론적으로 접근하는 방식은 요약해석법을 이용하거나 Joshi, 0

Leino나 Sabelfeld와 Sands가 제시한 방법[8,9]이 알려져 있다. 특히 요약해석법을 이용하면 정보 누출이 전후 방향으로 복구되는 상황을 인식할 수 있어서 구문적인 접근방식에 비해 높은 정밀도를 보인다.

이 논문에서는 명령형 프로그램의 핵심 부분을 대상으로 타입시스템을 이용한 분석법의 개선에 대하여 논하고자 한다. 타입시스템을 이용하는 방법을 개선하고 확장하여 요약해석법에 상응하는 정밀도를 보이기 위해서, 프로그램 흐름에 민감한(flow-sensitive) 타입시스템을 구성할 수 있다. 그러나 흐름에 민감한 타입시스템의 설계는 단순한 타입시스템에 비해 상대적으로 복잡하고 어렵다. 본 논문에서 채택한 방법은 SSA 형태(Static Single Assignment Form)로의 프로그램 변환[10,11,12,13]을 통하여 프로그램 흐름의 효과를 명확히 구별하고 세분화한 다음에 단순한 타입 추론시스템을 적용하여 흐름-민감 타입시스템이나 요약해석법과 같은 결과를 얻어내는 것이다.

이후의 논문 구성은 다음과 같다. 2장에서는 검사하고자 하는 정보흐름의 안전성이 무엇인지 간단히 설명하고 3장에서 간단한 명령형 프로그램을 SSA 형태로 변환하는 과정을 설명한다. 4장에서는 타입 추론을 위한 타입 시스템을 보여주고 그 효과에 대하여 5장에서 원시 프로그램에 대한 타입 추론의 결과와 비교하여 설명하고 6장으로 결론을 맺고 추후 연구 방향을 제시한다.

2. 정보 흐름의 안전성

이 논문에서 다루는 명령형 프로그램에 대한 정보 흐름의 보안성 분석 문제를 다음과 같이 약식으로 설명할 수 있다. 변수가 비밀변수(높은 보안수준을 가진 변수)와 공개변수(낮은 보안수준을 가진 변수)로 구별되어 있는 프로그램에서, 공개변수의 최종 값을 보고 비밀변수에 초기에 저장된 값에 관련된 어떤 정보를 알아낼 수 있는지 여부를 검사하는 문제를 정보 흐름의 보안성 분석 문제라고 한다. 즉, 공개변수의 실행전 값과 실행후 값을 보고 비밀변수의 실행전 값에 관한 정보를 절대로 알아낼 수 없는 경우, "정보 흐름은 안전이 보장된다."라고 한다.

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

```
m := x
```

왜냐하면 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
```

왜냐하면 m 값을 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.

```
if x then m := 0 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 정보누출을 모두 이 논문에서는 즉시 누출(immediate leak)이라고 한다. 정보는 전이적으로 누출될 수도 있다. 예를 들어, 다음 프로그램에서

```
m := x;
n := m;
```

변수 x에 저장된 값은 변수 m 뿐 아니라 변

수 n에 저장된 값을 보면 알 수 있다. 즉, 변수 x에서 변수 n으로의 정보 누출이 변수 m을 통하여 발생한다. 이러한 누출은 전이누출(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;
m := x;
```

왜냐하면, 프로그램 실행 전에 변수 x에 저장되어 있던 값은 변수 m에 저장되어 있던 공개 값으로 변경되었기 때문이다. 이런 경우 "변수 x에서 변수 m으로의 정보누출이 후방향으로 복구되었다(backwardly recovered)"라고 한다. 따라서 정보누출이 전혀 발생하지 않거나, 발생했더라도 모두 전방향 또는 후방향으로 복구된다면, 프로그램의 정보 흐름은 안전하다고 말할 수 있다.

Denning 부부의 최초 방법[2], Mizuno와 Schmidt의 데이터 흐름 분석 알고리즘[4], Volpano와 Smith의 타입시스템을 기반으로 하는 방법[6]은 모두 위에서 언급한 정보 누출의 복구를 감지할 수 없고, 따라서 바로 위의 두 예제 프로그램에 대해서 정보흐름이 안전하지 않다고 판정한다는 점에서 분석의 정밀도가 낮다. 본 논문의 방법은 마지막 두 예제 프로그램에 정보 누출이 없고 정보흐름이 안전하다고 판정한다.

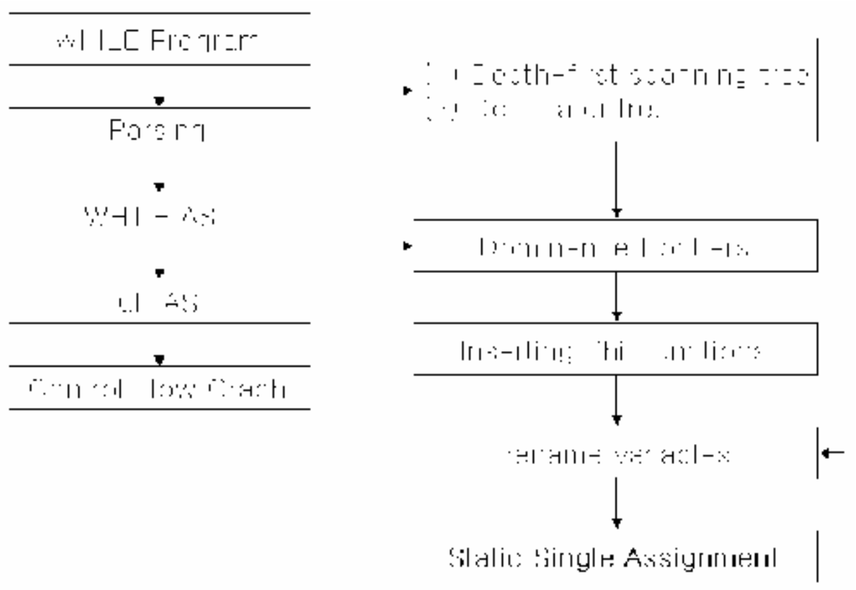
3. SSA 형태로의 변환

SSA 형태는 현재 데이터 흐름 분석이나 코드 최적화를 위해 컴파일러의 중간 표현으로 사용된다. SSA 형태는 (표 1)과 같이 각 변수들이 프로그램 안에서 한번씩만 정의된다.

(표 1) SSA form

source program	SSA program	current value	
		v	x
		v0	x0
v := 0;	v1 := 0;	v1	x0
x := v+1;	x1 := v1+1;	v1	x1
v := x;	v2 := x1;	v2	x1

각 변수에 대한 배정문이 정적인 시멘틱스 상에서만 단 한번이고, 동적인 시멘틱스 상에서는 루프에 의해 여러번 발생할 수 있다. 그래서 Static이라고 불린다. SSA 형태로의 변환 과정의 핵심은 소스 프로그램으로부터 같은 이름의 변수들이 서로 관련이 없을 때 변수이름을 재정의 하는 것이다. (그림 1)은 변환 과정의 단계들을 보여준다.



(그림 1) SSA 변환과정

본 논문의 대상언어인 WHILE 언어는 명령형 언어로서 배정문, 조건문, while문, skip문의 구조를 이루고 있다. 문법구조는 (표 2)와 같다. SSA 형태로 변환을 하기 위해서는 WHILE 프로그램을 그래프의 형태로 표현해야 되므로 그래프로 표현하기 좋은 언어인 FCL(Flow Chart

Language)[14]를 사용하여 그래프로 표현한다. SSA 형태 변환과정이 여러 단계로 이루어져 있어 각 단계별로 처리되는 과정을 간략히 알아본다.

(표 2) WHILE 문법

Syntax Domain:

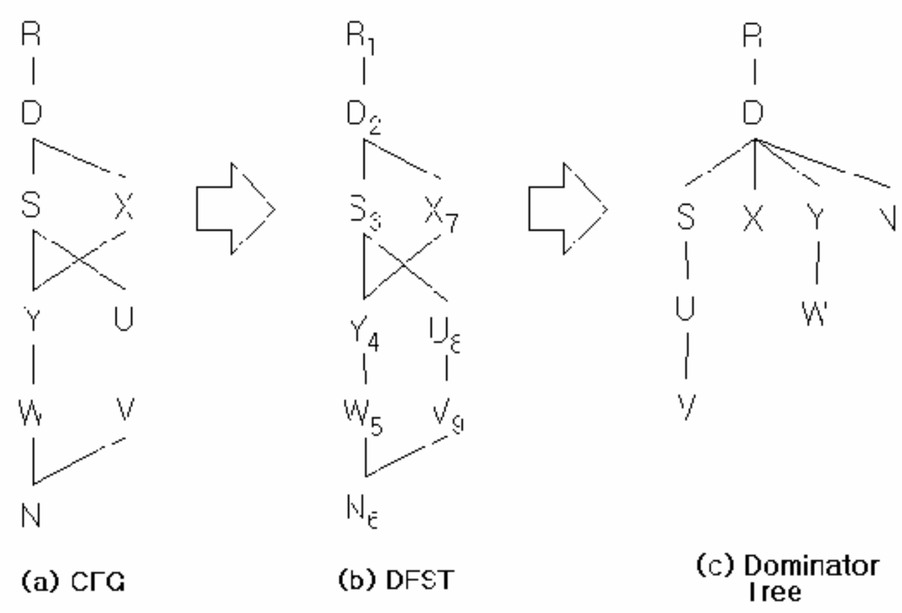
a ∈	**AExp**	arithmetic expressions
b ∈	**BExp**	boolean expressions
S ∈	**Stmt**	statements
x ∈	**Var**	variables

Abstract Syntax:

S ::= [x:=a] | [skip] | S1 ; S2 |
if [b] then S1 else S2 |
while [b] do S

3.1 지배자 트리

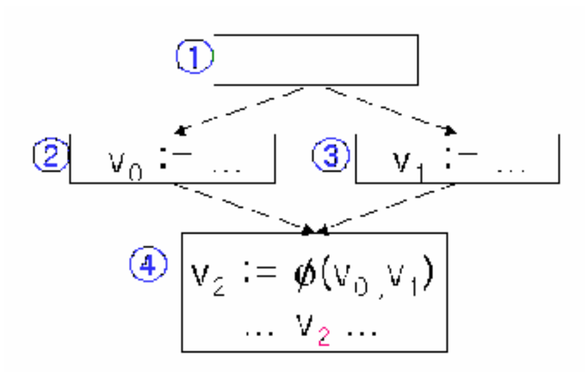
지배자 트리란 흐름 그래프에서 초기노드로부터 노드 n 까지 도달할 때 모든 경로는 d를 거쳐야 한다면 노드 d는 n을 지배한다. 예를 들어 [그림 2]에서 보듯이 노드 W의 경로는 R-D-Y-W의 최단 경로를 가지며 이때 노드 Y는 노드 W를 지배한다. 그래프에서 Depth-First Spanning Tree를 구한 다음 지배자 트리를 만든다.



(그림 2) 지배자 트리 변환 과정

3.2 Dominance Frontier

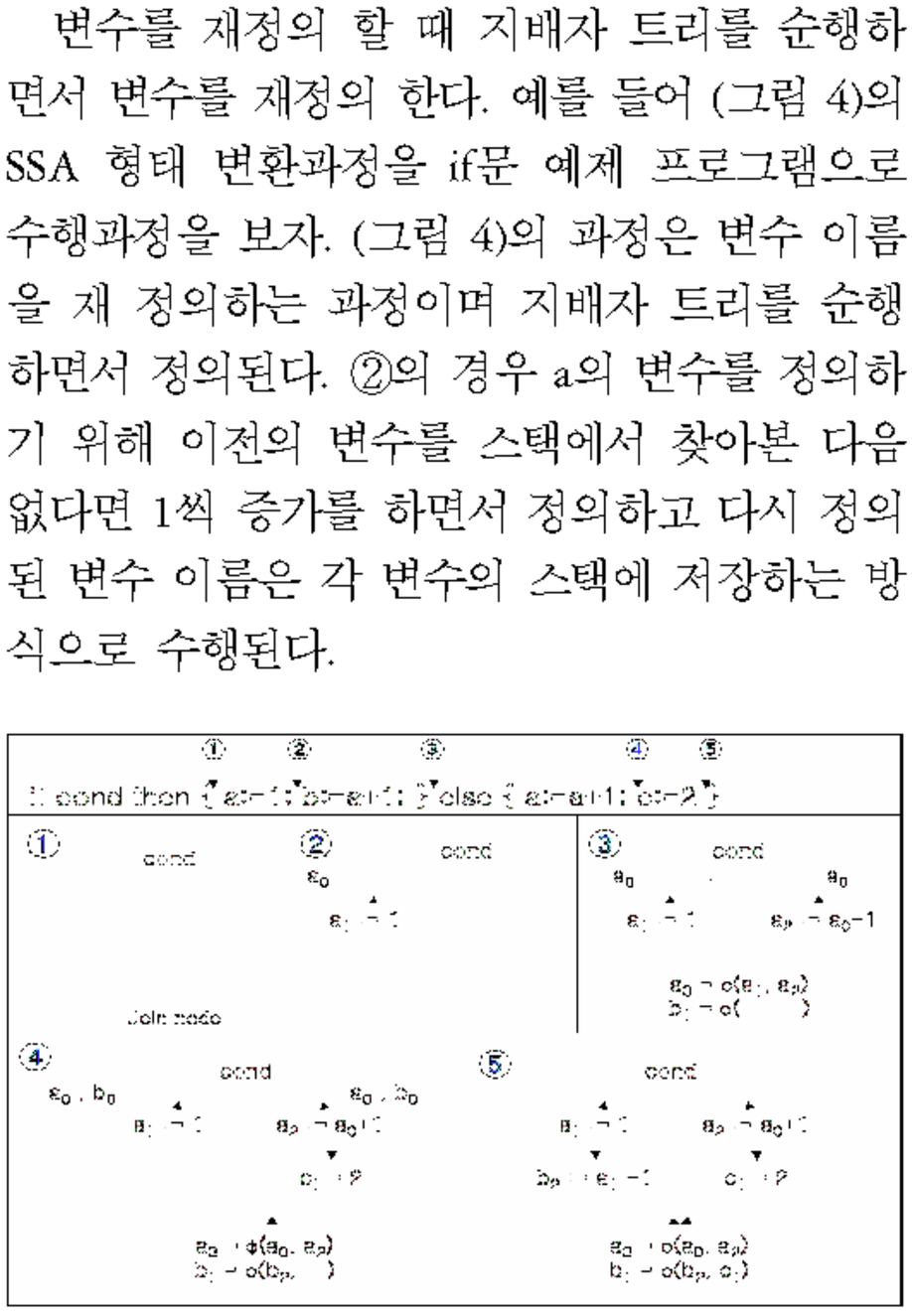
(그림 3)과 같이 ④노드 다음에 변수 V를 사용한다고 하자. 이때 ②,③노드에서 각각 변수 V가 재정의 되었다면 어느 변수를 사용할 것이지 결정할 수 없다. 이러한 점 때문에 ②,③노드를 병합하여 변수 V의 이름을 Φ 함수를 사용하여 재정의 한다. 이때 노드①은 노드④를 지배하지 않으므로 노드④는 노드①의 Dominance Frontier라 한다. 이때 Φ 함수는 if문과 while문에서 삽입된다.



(그림 3) Φ 함수 삽입

3.3 Rename variables

변수를 재정의 할 때 지배자 트리를 순행하면서 변수를 재정의 한다. 예를 들어 (그림 4)의 SSA 형태 변환과정을 if문 예제 프로그램으로 수행과정을 보자. (그림 4)의 과정은 변수 이름을 재 정의하는 과정이며 지배자 트리를 순행하면서 정의된다. ②의 경우 a의 변수를 정의하기 위해 이전의 변수를 스택에서 찾아본 다음 없다면 1씩 증가를 하면서 정의하고 다시 정의된 변수 이름은 각 변수의 스택에 저장하는 방식으로 수행된다.



(그림 4) if문 예제의 SSA 수행과정

4. SSA 형태를 위한 정보흐름분석

정보흐름 분석을 위해 타입 시스템을 이용하면 정형화된 타입 시스템 골격과 안전성(soundness) 증명 과정을 그대로 이용할 수 있다.

본 논문에서는 두 개의 보안 수준 H(high; secret)와 L(low; public)만 존재한다고 가정한다. 보안 수준의 집합이 기본 데이터 타입을 이룬다고 하면 WHILE 언어의 데이터 타입 τ는 다음과 같다.

$$\tau ::= \mathrm{H} \mid \mathrm{L} \mid \widehat{\mathrm{x}_{\mathrm{i}}} \mid \tau\ \mathrm{var} \mid \tau\ \mathrm{cmd}$$

여기서 $\widehat{x}_i$는 타입 명칭을 나타내는데 SSA 형태의 프로그램 변수 x_i의 타입을 지칭하는데 사용된다. 다형성(polymorphism)을 나타내기 위해 필요한 타입 변수와는 다르다는 것에 주의해야 한다. 이것이 [6,7]의 타입 시스템과 다른 점이다.

타이핑 관계는 다음과 같이 나타내는데

$$\Gamma, I \vdash P : \tau$$

프로그램 P는 타입 환경 Γ하에서 τ 타입을 갖는다는 것을 의미한다. 타입 환경 Γ는 주어진 SSA 변수의 초기 타입을 주는데, 초기에 모든 프로그램 변수에 보안 수준이 정해지므로 각 변수 x의 SSA 변수 x_0에 대한 타입을 주며 또한 I는 변수 $\widehat{x}_i$에 대한 집합을 말한다.

타입 규칙을 구문 지향적으로 나타내면 (그림 5)와 같다. 그 밖의 수식 등에 관한 규칙은 표준적인 규칙을 따르므로 여기서는 생략한다. 본 논문의 타입 규칙이 [6,7]과 다른 점은 각 타입 규칙이 부분 타입 관계를 강제변환하지 않고 부분 타입 관계로부터 제약 조건을 생성한다는 것이다. 이렇게 생성된 제약 조건의 집합은 최소해를 구하는 알고리즘에 의해 풀 수 있다.

$$\text{(Assign)}\ (\phi\text{-fun}):\ \frac{\Gamma, I \vdash x : \tau\ var,\ \Gamma, I \vdash a' : \tau \quad y_i \in FV(a) \cup I}{\Gamma, I \vdash x := a : \tau\ cmd \quad \{\hat{x} \geq \hat{y}_i\}}$$

$$\text{(Skip)}:\ \frac{}{\Gamma, I \vdash skip : int\ cmd \quad \{\}}$$

$$\text{(Comp)}:\ \frac{\Gamma, I \vdash S_1 : \tau\ var \quad C_1,\ \Gamma, I \vdash S_2 : \tau\ var \quad C_2}{\Gamma, I \vdash S_1; S_2 : \tau\ cmd \quad C_1 \cup C_2}$$

$$\text{(If)}:\ \frac{\Gamma, I \vdash e : bool,\quad \begin{array}{l}\Gamma, I \cup FV(e) \vdash S_1 : \tau\ cmd \quad C_1 \\ \Gamma, I \cup FV(e) \vdash S_{2'} : \tau\ cmd \quad C_2\end{array}}{\Gamma, I \vdash \text{if } e \text{ then } S_1 \text{ else } S_2 : \tau\ cmd \quad C_1 \cup C_2}$$

$$\text{(While)}:\ \frac{\Gamma, I \vdash e : bool \quad \Gamma, I \cup FV(e) \vdash S : \tau\ cmd \quad C}{\Gamma, I \vdash while\ e\ do\ S : \tau\ cmd \quad C}$$

(그림 5) 타입 규칙

알고리즘은 [15]의 abstract worklist 알고리즘을 따르고 있으며 본 논문의 제약 조건 형태에 맞도록 수정한 것이다. flow[x]는 x의 영향을 받는 SSA 변수 y에 대한 제약조건 $y \geq x$ 들의 집합이고, extract는 Worklist에서 하나의 제약조건을 LIFO 방식으로 처리하도록 내보내준다. 또한 알고리즘이 항상 멈추는지 그리고 알고리즘이 항상 원하는 최소해를 주는지에 대한 증명은 [15]의 증명을 거의 그대로 적용할 수 있다.

5. 분석의 정밀도

이 장에서는 정보 흐름 안전성 검사를 위한 타입유추의 알고리즘을 제시하고 타입 검사가 어떻게 이루어지는지 예제를 통하여 살펴본다.

(표 3) 제약조건의 해 구하는 알고리즘

입력:

제약조건의 집합 $S = x_1 \geq t_1, \cdots, x_n \geq t_n$

출력: 제약 조건의 최소해 Analysis

방법:

<단계 1> 초기화

```
Worklist := empty;
for all x ≥ t in S do
    Worklist := {x ≥ t} ∪ Worklist;
    if ( t = x̂_0 ) then Analysis[x]:= x̂_0;
                   else Analysis[x]:= ⊥;
    flow[x] := ∅;
```

for all $x \geq t$ in S do
    flow[t] := flow[t] ∪ {$x \geq t$}

<단계 2> Worklist와 Analysis의 반복 계산

while Worklist ≠ empty do
    (($x \geq t$), Worklist) := extract(Worklist);
    new := Analysis[t];
    if Analysis[x] ≱ new then
        Analysis[x] := Analysis[x] ⊔ new;
        for all $x' \geq t'$ in flow[x] do
            Worklist := {$x' \geq t'$} ∪ Worklist;

(예제) x:H m,n:L

```
m := n + 1;
while x > 0 do
  ( m := m + x;  x := x - 1 );
m := n - 1;
```

(SSA 변환)

```
m1 := n0 + 1;
m2 ← phi(m1, m3);
x1 ← phi(x0, x2);
while (x1>0) do
 (
    m3 := m2 + x1;
    x2 := x1 - 1;
 );
m4 := n0 - 1;
```

위 변환된 SSA 프로그램을 [그림 5]의 타입 규칙에 따라 풀이하면 아래와 같은 제약 조건의 집합을 구할 수 있다.

{($\widehat{m}_1 \geq \widehat{n}_0$),($\widehat{m}_2 \geq \widehat{m}_1$),($\widehat{m}_2 \geq \widehat{m}_3$),($\widehat{m}_3 \geq \widehat{m}_2$),($\widehat{m}_3 \geq \widehat{x}_1$),($\widehat{m}_4 \geq \widehat{n}_0$),
($\widehat{x}_1 \geq \widehat{x}_0$),($\widehat{x}_1 \geq \widehat{x}_2$),($\widehat{x}_2 \geq \widehat{x}_1$)}

이러한 제약조건의 집합은 리스트의 구조와 같다. 그 이유는 SSA 형태가 각 변수 마다 체인(chain)을 형성하고 있어 리스트의 구조와 같으므로 LIFO방식인 (표 3)의 알고리즘을 수행하면서 해를 구한다. 해를 구하는 과정은 아래와 같다.

(풀이)

[Analysis]

	SC	1	2	3	4	5	6	7	8	9	10	11	12
$\widehat{x}_0$	H												
$\widehat{n}_0$	L												
$\widehat{m}_0$	L												
$\widehat{x}_1$	⊥							H					H
$\widehat{x}_2$	⊥									H			
$\widehat{m}_1$	⊥	L											
$\widehat{m}_2$	⊥		L										
$\widehat{m}_3$	⊥				L						L	H	
$\widehat{m}_4$	⊥						L						

flow[t]

$\widehat{x}_0$	($\widehat{x}_1 \geq \widehat{x}_0$)
$\widehat{n}_0$	($\widehat{m}_1 \geq \widehat{n}_0$), ($\widehat{m}_4 \geq \widehat{n}_0$)
$\widehat{m}_0$	
$\widehat{x}_1$	($\widehat{m}_3 \geq \widehat{x}_1$), ($\widehat{x}_2 \geq \widehat{x}_1$)
$\widehat{x}_2$	($\widehat{x}_1 \geq \widehat{x}_2$)
$\widehat{m}_1$	($\widehat{m}_2 \geq \widehat{m}_1$)
$\widehat{m}_2$	($\widehat{m}_3 \geq \widehat{m}_2$)
$\widehat{m}_3$	($\widehat{m}_2 \geq \widehat{m}_3$)
$\widehat{m}_4$	

위의 풀이와 같이 3번째 ($\widehat{m}_2 \geq \widehat{m}_3$)에서 $\widehat{m}_3$의 정보를 알 수 없으므로 다시 worklist에 삽입을 하고 다음을 수행한다. 이와 같이 반복적으로 수행을 하여 모든 변수의 정보를 구할 수 있다. 5번째 단계인 ($\widehat{m}_3 \geq \widehat{x}_1$)은 $\widehat{x}_1$의 정보를 7번째 단계에서 알 수 있으며 $\widehat{m}_3$의 보안수준이 H(high)가 되었지만 $\widehat{m}_4$

의 보안수준이 L(low)임을 알 수 있으므로 m 변수의 정보흐름이 전방향으로 회복되었음을 인지할 수 있다. 결과적으로 누출이 일어나지 않았으므로 정보흐름이 안전함을 분석할 수 있다.

6. 결론 및 향후 연구 방향

본 논문에서는 주어진 프로그램의 변수들이 서로 다른 보안 수준을 가진다고 할 때 보안 수준이 다른 변수들 간의 정보 흐름이 안전한 지를 정적으로 분석하는 문제를 다루었다. 구문적인 방법 중에서 구현의 용이성으로 각광받는 타입 시스템을 이용하는 경우에 정보 누출이 복구되는 상황을 인식하지 못하는 단점을 해결하기 위해 주어진 프로그램을 SSA 형태로 변환한 후에 타입 시스템을 적용하는 방법을 설명하였다. 이때 적용되는 타입 시스템은 보안 수준 변수를 도입하고 부분 타입 관계에 의한 제약 조건 집합을 생성하여 제약 조건의 최소해를 반복 알고리즘을 이용하여 구하도록 하였다.

이러한 방법은 컴파일러의 중간 표현으로 많이 사용되는 SSA 형태를 이용한다는 점에서 추가적인 단계나 오버헤드가 필요하지 않고, 간단한 제약조건 기반 분석을 적용가능하다는 이점이 있다. 다음으로 이 방법을 프로시져를 포함하는 프로그램과 객체지향 프로그램에 적용하는 연구가 이어질 수 있다.

참고문헌

[1] D. Denning, Secure Information Flow in Computer Systems, Purdue University ph.D. Thesis, 1975.

[2] D. Denning, "A Lattice Model of Secure Information Flow", Communications of the ACM 19, 5, (1976), 236-242.

[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] M. Mizuno and D.A. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. Formal Aspects of Computing, 4:722-754, 1992.

[5] Kyung-Goo Doh, and Seung Cheol Shin, Data Flow Analysis of Secure Information- Flow, In Proceedings of the 3rd Asian Workshop on Programming Languages and Systems, 2002.

[6] Dennis Volpano, Geoffrey Smith and Cynthia Irvine, "A Sound Type System for Secure Flow Analysis", Journal of Computer Security, Jul 29, 1996.

[7] Dennis Volpano and Geoffrey Smith, "A Type-Based Approach to Program Security", In Proc. TAPSOFT '97, volume 1214 of Lecture Notes in Computer Science, pages 607-621, Springer-Verlag, April 1997.

[8] R. Joshi and K. R. M. Lenio, A semantic approach to secure information flow. Science of Computer Programming, 37:113-138, 2000.

[9] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. Higher-Order andSymbolic Computations, 14: 59-91, 2001.

[10] Andrew W. appel, "Modern Compiler Implementation in Java", CAMBRIDGE UNIVERSITY PRESS, 1998, pp437-477.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", March 7, 1991.

[12] Jeremy Singer, "Static Single Information for Reverse Engineering", October 18, 2002.

[13] M. M. Brandis, "Single-Pass Generation of Static Single Assignment Form for Structured Languages", ACM Transactions on Programming Languages and Systems, Vol 16. No 6, November 1994, pp 1684-1698.

[14] John Hatcliff, "An Introduction to Online

and Offline Partial Evaluation Using a Simple Flowchart Language", Department of Computing and Information Science Kansas State University, 1998.

[15] F. Nielson H.R. Nielson, and C. Hankin. Principles of Program Analysis, Springer, 1999. pp363-385.

최 성 권

1996년~2002년 동양대학교 컴퓨터공학과(학사)

2002년~2003년 동양대학교 컴퓨터공학과(석사과정)

관심분야 : 프로그래밍 언어, 컴파일러, 타입시스템

신 승 철

1987년 인하대학교 전자계산학과(학사)

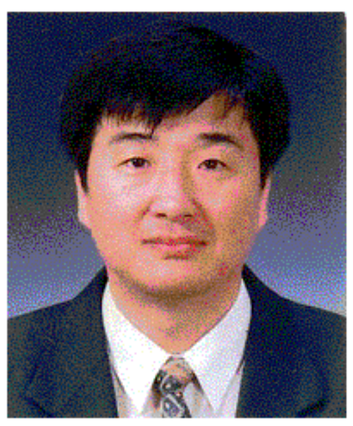
1989년 인하대학교 전자계산학과(석사)

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