creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사학위논문

정적 분석 정확도 향상을 위한 코드 변환 규칙

Code Transformation Rules to Improve Static Analysis Accuracy

2022 년 8 월

서울대학교 대학원

컴퓨터 공학부

김 세 훈

공학석사학위논문

정적 분석 정확도 향상을 위한 코드 변환 규칙

Code Transformation Rules to Improve Static Analysis Accuracy

2022 년 8 월

서울대학교 대학원

컴퓨터 공학부

김 세 훈

정적 분석 정확도 향상을 위한 코드 변환 규칙

Code Transformation Rules to Improve Static Analysis Accuracy

지도교수 이 광 근

이 논문을 공학석사학위논문으로 제출함

2022 년 6 월

서울대학교 대학원

컴퓨터 공학부

김 세 훈

김세훈의 석사학위논문을 인준함

2022 년 7 월

위 원 장	문병로	(인)
부위원장	이광근	(인)
위 원	허충길	(인)

요약

분석 기술의 변화 없이 소스코드를 변환하여 정적 분석 정확도를 향상시키는 방법을 소개하고 실험을 통해 효용성을 확인한다. C프로그램의 범위를 벗어난 메모리 접근(buffer overrun) 오류를 탐지하는 정적 분석기를 대상으로 코드를 변환하여 분석 정확도를 향상시는 코드 변환 규칙을 찾아내었다. 각 코드 변환 규칙은 실행 의미를 보존하면서 별도의 값 비싼 분석 없이 코드의 모양(syntax)만을 확인하여 자동으로 변환될 수 있는 규칙이다. 반복문의 조건식과 반복문 안의 메모리 접근식을 변환하는 규칙으로 자동 변환기를 구현하여 실험한 결과, 정확도는 평균 0.51%, 분석 시간은 평균 0.7% 증가하였다. 변환 과정이 별도의 분석 없이 코드의 모양만을 확인하기 때문에 허위 경보를 저렴한 비용으로 제거하였지만, 제거에 성공한 허위 경보의 절대적인 양은 많지 않았다.

주요어: 프로그램 분석, 정적 분석, 코드 변환

학번: 2019-28745

목차

요약 **i**

목차 **ii**

그림 목차 **iv**

표 목차 **v**

제 1 장 서론 **1**

제 2 장 배경 지식 **4**

2.1 대상 분석기 4

2.1.1 대상 언어 4

2.1.2 요약 도메인 및 실행 의미 5

2.2 분석기 특성 6

2.2.1 정확도를 잃는 경우 1: 분기문 7

2.2.2 정확도를 잃는 경우 2: 함수 호출 8

2.2.3 정확도를 잃는 경우 3: 반복문 8

제 3 장 변환 규칙 **10**

3.1 변환 규칙 표기법 10

3.2 반복문의 조건식 변환 11

3.2.1 대소 비교가 아닌 조건식 12

3.2.2 실제 범위보다 넓은 조건식 14

3.3 반복문 안의 메모리 접근식 변환 16

제 4 장 실험 결과 **19**

제 5 장 논의 및 결론 **25**

5.1 코드 변환을 이용한 분석 정확도 향상의 한계 25

5.1.1 융통성 없는 변환 규칙 25

5.1.2 코드 변환이 정확도를 향상시키지 못하는 경우 26

5.2 가능한 다른 코드 변환 26

5.2.1 분기문 변환 27

5.2.2 함수 호출문 변환 27

5.3 관련 연구 28

5.4 결론 29

참고문헌 **30**

Abstract **32**

그림 목차

그림 1.1 동일한 실행 의미를 가지지만 분석 결과가 달라질 수 있는 코드 예시 2

그림 2.1 범위를 벗어난 배열 접근 오류 예시 6

그림 2.2 분기문 양쪽 가지의 정보를 합치며 정확도가 떨어지는 경우 예시 7

그림 2.3 여러 함수 호출문을 하나로 분석하여 정확도가 떨어지는 경우 예시 8

그림 2.4 반복문의 모양에 따라 분석 정확도가 달라지는 경우 예시 . . . 9

그림 3.1 같지 않음(≠)연산자를 사용하는 조건식 변환 예시 12

그림 3.2 실제 변수의 범위보다 넓은 조건식 변환 예시 14

그림 3.3 반복문 안의 메모리 접근식 변환 예시 16

그림 5.1 변환을 포기하는 경우 예시 25

그림 5.2 코드 변환이 정확도를 향상시키지 못하는 경우 예시 26

그림 5.3 함수 들여쓰기가 적용된 코드 예시 27

그림 5.4 코드 변환 패턴의 예시 28

표 목차

표 4.1 실험 결과 21
표 4.2 변환 규칙 적용률 24

제 1 장 서론

본 논문에서는 분석 기술의 변화 없이 분석 대상 소스코드(source code)의 변환만으로 정적 분석(static analysis) 정확도가 향상될 수 있음을 보이고, 실험을 통해 실제 효용성을 확인한다.

안전(sound)한 정적 분석은 분석 결과가 실제 상황을 모두 포함하지만, 실제로 일어나지 않는 상황도 포함하게 되어 정확도를 잃게 된다. 요약 해석(abstract interpretation)[1] 기반의 정적 분석에서는 실제 실행시 프로그램이 갖는 값을 요약하여 분석한다. 분석의 안전함을 위해서 요약 의미 구조(abstract semantics)는 분석 결과가 프로그램 실제 실행중 가질 수 있는 모든 상태를 포섭하도록 디자인된다. 따라서 요약 의미 구조를 따라 분석을 진행하면 분석 결과의 안전함은 보장되지만 분석의 진행에 따라 실제 이상의, 즉 실제로는 일어나지 않는 프로그램 상태를 포함하게 되어 정확도를 잃게 된다. 다시 말하면 실제로는 발생하지 않는 오류를 발생할수 있다고 말하는 허위 경보(false alarm)를 결과로 내게 되는 것이다.

허위 경보를 줄이고 실제로 일어나는 상황만을 더 잘 분석하기 위해서는 일반적으로 더 정교한 분석 기술이 요구된다. 예를 들어 함수 호출문을 더 정확하게 분석하고자 한다면 함수 호출 맥락을 고려하는 분석(context-sensitive) 기술이 필요하다. 함수 호출 맥락을 고려하지 않는(context-insensitive) 분석에서는 동일한 함수에 대한 호출문을 한데 뭉뚱그려서 분석하기 때문에 함수 호출문을 분석할 때 정확도가 떨어지게 된다. 따라서 함수 호출문을 이보다 잘 분석하기 위해서는 함수가 호출되는 코드 위치에 따라 개별적으로 분석을 하는 새로운 분석 기법의 도입이 필요한 것이다.

분석 기술의 정교화 없이 분석 대상 소스코드를 동일한 실행 의미(semantics)의 다른 코드로 변환하여, 같은 분석기로도 더 정확한 분석을 할 수 있다. 동일한

```
1 if (cond) {
2   x = -1;
3 } else {
4   x = 1;
5 }
6
7 1/x; // x->[-1,1]
```

```
1 if (cond) {
2   x = -1;
3   1/x; // x->[-1,-1]
4 } else {
5   x = 1;
6   1/x; // x->[1,1]
7 }
```

그림 1.1: 동일한 실행 의미를 가지지만 분석 결과가 달라질 수 있는 코드 예시

실행 의미를 갖는다 하더라도 겉모습(syntax)이 다르면 정적 분석 과정을 거쳐 다른 분석 결과를 가질 수 있다. 그림 1.1의 두 프로그램은 동일한 실행 의미를 갖는 다른 모양의 프로그램이다. 프로그램 값을 구간(interval domain)으로 요약하여 0으로 나누기(division by zero) 오류를 탐지하는, 분기문을 뭉뚱그려 분석하는 (path-insensitive) 정적 분석기로 두 프로그램을 분석하면 다른 결과를 얻게 된다. 분석기는 왼쪽 프로그램 5째 줄에서 분기가 끝날 때 변수 x의 값을 $[-1, 1]$의 요약값, 즉 -1에서 1 사이의 값을 가질 수 있다고 뭉뚱그린다. 실제로는 가질 수 없는 0이 포함되었기 때문에 7째줄의 `1/x` 식에서 0으로 나누기가 가능하다는 허위 경보를 발생시킨다. 반면 동일한 실제 실행 의미를 갖는 오른쪽 프로그램에서는 분기문이 끝나기 전 각 가지 안에서 나눗셈이 계산된다. `1/x` 식이 분석되는 각 지점에서 변수 x의 값이 0이 아님이 확실하기 때문에 허위경보가 발생하지 않는다.

이번 연구에서는 분석 정확도를 향상시킬 수 있는 다양한 코드 변환 규칙을 조사하고, 실험을 통해 정확도와 비용 측면에서 얼마나 효과가 있는지 확인한다. 정적 분석의 정확도 향상을 위한 코드 변환은 새로운 개념이 아니다. 반복문 풀어 쓰기(loop unrolling), 함수 들여쓰기(function inlining) 등의 코드 변환 기술은 컴파일러(compiler) 최적화 뿐 아니라 정적 분석에서도 사용되고 있다. 본 연구에서는 이러한 기존 기술들 이외의, 별도의 값비싼 분석 없이 프로그램의 모양만을 보고 분석 정확도를 높이는 방향으로 코드를 자동 변환시키는 규칙들을 알아본다.

이러한 코드 변환 규칙들은 다음과 같은 성질을 만족해야 한다.

1. 프로그램 식의 실행 의미를 그대로 보존해야 한다.
2. 소스 코드의 속 내용 분석(data-flow analysis) 없이 겉 모습(syntax)만 보고 빠르게 변환할 수 있어야 한다.
3. 변환이 적합한지 판단하는 과정과 변환 과정이 전자동으로 진행될 수 있어야 한다.
4. 분석 정확도를 향상 시킬 여지가 있어야 한다.

이러한 규칙들을 자동 변환기로 구현하여 실험한 결과 100개의 벤치마크(benchmark) 프로그램의 13,575개의 경보 중, 6개의 프로그램에서 총 17개의 허위 경보를 제거하였다. 전체 경보 대비 0.13%의 허위 경보가 제거되었고, 각 프로그램 별 제거된 허위 경보 비율의 평균은 0.51%, 증가된 분석 비용(시간)의 평균은 0.7%이다. 제거된 경보는 모두 허위 경보였으며, 코드 변환으로 인해 새로 발생한 허위 경보는 없었다.

2장에서는 코드 변환 전후의 성능을 비교할 대상 분석기와, 분석기가 정확도를 잃게 되는 경우를 설명한다. 3장에서 구체적인 변환 규칙, 즉 코드의 겉모양 만으로 변환이 적합한지 판단하고 실제로 변환을 적용하는 규칙을 자세히 설명한다. 4장에서는 변환 규칙을 구현한 변환기를 사용하여, 적용 전후의 분석기의 성능을 비교하여 효용성을 판단한다. 5장에서는 본 연구의 한계, 다루지 않은 변환 방법 및 관련 연구를 소개하고 결론을 짓는다.

제 2 장 배경 지식

이번 장에서는 본 연구를 통해 정확도를 향상시키고자 하는 대상 분석기 스패로우[2]에 대해 소개한다. 2.1절에서 분석기의 대상 언어 및 분석기 디자인을 설명하고, 2.2절에서 분석의 정확도를 잃게 되는 경우들을 예시와 함께 설명한다.

2.1 대상 분석기

스패로우는 요약 해석 틀(abstract interpretation framework)[1]을 바탕으로 설계된, C 프로그램의 다양한 오류들을 실행 전에 탐지하는 정적 분석기이다. 본 연구에서는 스패로우의 다양한 기능 중 구간 도메인(interval domain)을 이용한 범위를 벗어난 메모리 참조(buffer overrun) 탐지만을 다룰 것이다. 분석기 및 본 연구의 코드 변환기는 전체 C 언어를 대상으로 하지만, 간결한 설명을 위해 간략화된 대상 언어를 사용한다. 대상 언어 및 요약 실행 의미 표기는 동일한 분석기를 대상으로 한 논문[3]을 참고하였다.

2.1.1 대상 언어

$$
\begin{array}{lll}
S \to L := E & E \to n & L \to x \\
\quad \mid S ; S & \quad \mid L \mid \&L & \quad \mid \star E \\
\quad \mid \texttt{if}\, E\ S\ S & \quad \mid \texttt{alloc}_l(E) & \quad \mid E[E] \\
\quad \mid \texttt{while}\, E\ S & \quad \mid E\ bop\ E \mid !E & bop \to + \mid - \mid \star \mid / \mid == \mid <
\end{array}
$$

분석기와 코드 변환 규칙의 대상 언어는 위와 같다. 프로그램 구문은 할당문($L := E$), 연결문($S ; S$), 분기문($\texttt{if}\, E\ S\ S$), 반복문($\texttt{while}\, E\ S$)이다. 프로그램의 값은 정수 또는 주소이고, 주소는 변수 이름이거나 할당식($\texttt{alloc}_l(E)$)을 통해 생성된 메모리 주소이다. 메모리 주소는 참조식($\star E$, $E[E]$)을 통해 접근할 수 있다. 분기문과 반복문은 조건식 E의 값이 0이 아닐 경우 참으로, 0일 경우 거짓으로

취급한다. 비교 연산(E==E, E<E) 은 참일 경우 1로, 거짓일 경우 0으로 계산된다. 역 연산식(!E)은 E의 값이 0이 아닐 경우 0으로, 0일 경우 1으로 계산된다. 프로그램이 상식적인 명령형 언어의 실행 의미를 따르기 때문에, 실제 도메인(concrete domain) 및 실제 실행 의미(concrete semantics)에 대한 정의는 생략한다.

2.1.2 요약 도메인 및 실행 의미

$$
\begin{array}{rcl}
s \in \mathbb{S} & = & \mathbb{L} \to \mathbb{V} \\
\mathbb{L} & = & \mathit{Var} + \mathit{Allocsite} \\
\mathbb{V} & = & \mathbb{I} \times \wp(\mathbb{L}) \times \wp(\mathbb{A}) \\
\mathbb{I} & = & \{\bot\} \cup \{[l, u] \,|\, l, u \in \mathbb{Z} \cup \{\pm\infty\}\} \\
\mathbb{A} & = & \mathbb{L} \times \mathbb{I} \times \mathbb{I}
\end{array}
$$

요약 메모리 $\mathbb{S}$ 는 요약 메모리 주소($\mathbb{L}$)와 요약 값($\mathbb{V}$)의 매핑(mapping)이다. 요약 메모리 주소는 프로그램 변수 주소(Var) 또는 메모리 할당 지점($\mathit{Allocsite}$)이다. 요약 값은 정수를 요약한 구간 값($\mathbb{I}$), 요약 메모리 주소 집합, 요약 배열 값 집합으로 구성된다. 요약 배열 값($\mathbb{A}$)은 메모리 할당 지점, 오프셋(offset), 배열 크기로 정의된다.

$$
\begin{array}{rcl}
F & : & C \times \mathbb{S} \to \mathbb{S} \\
\mathcal{L} & : & L \times \mathbb{S} \to \wp(\mathbb{L}) \\
\mathcal{V} & : & E \times \mathbb{S} \to \mathbb{V}
\end{array}
$$

$$
\begin{array}{rcl}
F(L := E, s) & = & s[\mathcal{L}(L, s) \mapsto \mathcal{V}(E, s)] \\
F(S_1 ; S_2, s) & = & F(S_2, F(S_1, s)) \\
F(\texttt{if}\ E\ S_1\ S_2) & = & F(S_1) \sqcup F(S_2) \\
F(\texttt{while}\ E\ S) & = & \mathbf{fix}(\lambda X.s \sqcup F(C, X))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{L}(x,s) & = & \{x\} \\
\mathcal{L}(\star E,s) & = & \mathcal{V}(E,s).2 \\
\mathcal{L}(E_1\,\texttt{[}\,E_2\,\texttt{]},s) & = & \{a \mid \langle a,\_,\_\rangle \in \mathcal{V}(E_1,s).3\} \\
\mathcal{V}(n,s) & = & \langle [n,n],\emptyset,\emptyset\rangle \\
\mathcal{V}(L,s) & = & \bigsqcup\{s(l) \mid l \in \mathcal{L}(L,s)\} \\
\mathcal{V}(\&L,s) & = & \langle \bot,\mathcal{L}(L,s),\emptyset\rangle \\
\mathcal{V}(\texttt{alloc}_l\,\texttt{(}\,E\,\texttt{)},s) & = & \langle \bot,\emptyset,\{\langle l,[0,0],\mathcal{V}(E,s).1\rangle\}\rangle \\
\mathcal{V}(E_1\,bop\,E_2,s) & = & \mathcal{V}(E_1,s)\ \hat{bop}\ \mathcal{V}(E_2,s) \\
\mathcal{V}(\,!\,E,s) & = & \hat{!}\,\mathcal{V}(E,s)
\end{array}
$$

요약 실행 의미는 실행 의미 함수 $F : C \times \mathbb{S} \to \mathbb{S}$에 의해 정의된다. 주어진 요약 상태에 대하여, 두 함수 $\mathcal{L} : L \times \mathbb{S} \to \wp(\mathbb{L})$ 과 $\mathcal{V} : E \times \mathbb{S} \to \mathbb{V}$ 는 각각 프로그램 식 L과 E를 요약 실행하여 요약 메모리 주소($\mathbb{L}$)의 집합, 요약 값($\mathbb{V}$)으로 계산한다. 각각의 요약 실행 함수는 표준적인 정의를 따르며, 기본적인 이항 연산자($\hat{+}, \hat{-}, \cdots$) 및 역 연산자($\hat{!}$)의 정의는 생략한다.

2.2 분석기 특성

```
1 char *p = malloc(5);  // size->[5,5]
2
3 // size->[5,5], offset->[4,4] : safe
4 p[4] = 0;
5
6 // size->[5,5], offset->[5,5] : unsafe
7 p[5] = 0;
```

그림 2.1: 범위를 벗어난 배열 접근 오류 예시

분석기는 각 프로그램 식이 가질 수 있는 값을 구간(interval)으로 요약하여, 메모리 접근식에서 배열의 크기와 오프셋(offset)이 안전하지 못한 경우를 탐지한다.그림 2.1의 프로그램으로 예를 들면, 4번째 줄의 메모리 접근은 안전하고 7번째 줄의 메모리 접근은 안전하지 않다는 분석 결과를 내게 된다.

안전한 분석을 위해, 분석기는 실제 이상의 프로그램 실행 상태를 요약하게 되는데 이 과정에서 분석 정확도를 잃게 된다. 2.2.1, 2.2.2 에서는 각각 분기문과 함수 호출문에서 정확도를 잃게 되는 경우를, 2.2.3 에서는 반복문에서 정확도를 잃게 되는 경우를 설명한다. 본 연구에서는 2.2.3 의 반복문 분석 정확도 개선을 위한 코드 변환 규칙을 제시한다.

2.2.1 정확도를 잃는 경우 1: 분기문

```
1 if (x > 0) {
2   y = 1;  // x->[1,+oo], y->[1,1]
3 } else {
4   y = -1;  // x->[-oo,0], y->[-1,-1]
5 }
6
7 // x->[-oo,+oo], y->[-1,1]
```

그림 2.2: 분기문 양쪽 가지의 정보를 합치며 정확도가 떨어지는 경우 예시

분기문의 양쪽 가지를 분석한 후, 각각의 요약 메모리 상태를 하나로 합치는 과정에서 정확도를 잃게 된다. 그림 2.2 프로그램에 대해 분석기는, 참 가지는 x의 값이 1 이상이고 y의 값이 1인 상태만을, 거짓 가지는 x의 값이 0 이하이고 y의 값이 -1인 상태만을 가질 수 있다고 정확하게 분석한다. 하지만 분기문이 끝나는 지점에서 각 가지의 요약 메모리 상태를 합치게 되고, 이 과정에서 x의 값은 어떤 정수도 가능하고 y의 값은 -1, 0, 1 중 하나인 것으로 요약되어 x와 y사이 관계에 대한 정보는 사라진다. 이로 인해 분석기는 7번째 줄에서 (x == -1) ∧ (y == 1), (x == 0) ∧ (y == 0) 등의 실제로는 일어나지 않는 프로그램 상태 또한 가능하다고 잘못 분석한다.

2.2.2 정확도를 잃는 경우 2: 함수 호출

```
1 int *malloc_int (int len) {
2   return malloc(sizeof(int)*len);
3 }
4
5 void foo () {
6   int *p = malloc_int(5);  // size->[5,10]
7   int *q = malloc_int(10);  // size->[5,10]
8
9   p[4] = 0;  //size->[5,10], offset->[4,4] : safe
10  q[9] = 0;  //size->[5,10], offset->[9,9] : unsafe
11 }
```

그림 2.3: 여러 함수 호출문을 하나로 분석하여 정확도가 떨어지는 경우 예시

하나의 함수 정의에 대한 여러 함수 호출 결과값을 하나로 뭉뚱그려 계산하는 과정에서 정확도를 잃게 된다. 현실적인 분석 비용을 위해, 각각의 함수 호출문 마다 별개로 반환값을 분석하는 대신 동일한 함수에 대한 호출 결과는 모두 뭉뚱그려 하나의 값으로 분석하고, 그 과정에서 실제로 일어나지 않는 경우를 포함하게 된다. 그림 2.3 프로그램에서 분석기는 함수 `malloc_int`의 입력값 5와 10에 대한 개별의 배열을 반환하는 대신, '5 이상 10 이하' 로 요약된 입력값에 대해, $[5, 10]$ 크기의 배열을 반환하다. 따라서 10번째 줄의 배열 접근식은 안전함에도 불구하고 경보를 발생하게 된다.

2.2.3 정확도를 잃는 경우 3: 반복문

반복문의 고정점(fixpoint) 계산 과정에서 정확도를 잃게 된다. 안전한 분석을 위해 분석기는 반복문 안의 프로그램 식을 계속하여 요약 실행한다. 요약 실행 후, 계산된 요약 메모리 상태가 마지막 요약 실행 전과 동일한 고정점에 도달하게 되면 해당 반복문의 분석이 완료된다. 하지만 요약값의 도메인(domain)이 유한하지

```
1 // size [5,5]
2 char *p = malloc(5);
3
4 for(i = 0; i < 5; i++) {
5   // size->[5,5],
6   // offset->[0,+oo] : unsafe
7   *p = 0;
8   p++;
9 }
```

(a) 허위 경보가 발생한 경우

```
1 // size [5,5]
2 char *p = malloc(5);
3
4 for(i = 0; i < 5; i++) {
5   // size->[5,5],
6   // offset->[0,4] : safe
7   p[i] = 0;
8 }
```

(b) 정확하게 분석된 경우

그림 2.4: 반복문의 모양에 따라 분석 정확도가 달라지는 경우 예시

않을 경우 고정점에 도달하지 못하고 반복문 요약 실행을 계속 하여 분석이 끝나지 않게 된다. 이때 계속 갱신되는 변수의 요약값을 단순히 넓게 어림잡는 넓히기 기법(widening) 을 사용하여 유한한 시간안에 분석을 끝낼 수 있게 한다. 이를 통해 분석의 안전함과 현실적인 시간 내에 종료됨이 보장되지만 분석의 정확도를 상당히 잃게 되고, 이로 인해 반복문은 정적 분석기의 정확도가 저하되는 주 요인이다. 그림 2.4의 (a) 프로그램에서, 분석기는 7번째 줄 메모리 접근식이 안전하지 않다고 경보를 내게 된다. 실제 실행시에는 범위를 벗어나지 않는 안전한 메모리 접근이지만, 반복문 안의 프로그램 식이 요약 실행될 때마다 `p`의 값이 갱신되기 때문에, 넓히기 기법에 의해 실제 이상으로 넓게 어림잡혀 허위 경보가 발생한다.

반복문의 조건식을 이용해 반복문 분석을 더 정확하게 할 수 있다. 반복문의 조건식은 반복문 안의 프로그램 식이 시작되는 지점에서 반드시 만족해야 하는 불변식으로서 분석기에게 중요한 힌트를 제공한다. 그림 2.4의 (b) 프로그램 반복문 안에서 `i`의 값은 조건식 (`i < 5`)에 의해 $[0,+\infty]$이 아닌 [0,4]로 범위가 좁혀지고, 7번째 줄의 메모리 접근식의 안전함이 정확하게 분석될 수 있다.

제 3 장 변환 규칙

3.1 변환 규칙 표기법

$$\frac{\begin{array}{rl}\text{safety}: & \textit{condition} \\ \text{transform}: & \textit{explain}\end{array}}{\mathsf{trans}_{\mathsf{A}}(S) \quad \Rightarrow \quad S'}$$

변환 규칙은 위와 같은 형태로 표현되며, 그 뜻은 다음과 같다. 변환 규칙 A 는 코드 변환이 안전하기 위한 조건 *condition*을 만족할 때에 프로그램 구문 S를 같은 실행 의미를 갖는 다른 모양의 프로그램 구문 S'으로 변환한다. *explain*은 간결한 변환 규칙 표기를 위한 설명문이다. 조건을 만족하지 않거나 변환 규칙에 명시되지 않은 모양을 가진 프로그램에 대해서는 부분 프로그램 구문(sub-statement)들이 변환 규칙 A에 의해 변환된 코드를 반환한다. 상수 조건식을 갖는 분기문을 단순화하는 다음의 코드 변환 규칙을 간단한 예시로서 살펴보겠다.

$$\frac{\begin{array}{rl}\text{safety}: & E_c = n, n \neq 0 \\ \text{transform}: & \mathsf{trans}_{\mathsf{pr}}(S_1) \Rightarrow S_1'\end{array}}{\mathsf{trans}_{\mathsf{pr}}(\ \texttt{if}\ E_c\ S_1\ S_2\ ) \quad \Rightarrow \quad S_1'}$$

위 변환 규칙pr(pruning, 실행되지 않는 코드 가지치기)은 '분기문 ($\texttt{if}\ E_c\ S_1\ S_2$)에 대해서 조건식 E_c가 0이 아닌 상수일 때 S_1'으로 변환한다'는 의미이다.

$$\frac{\begin{array}{rl}\text{safety}: & \neg(E_c = n, n \neq 0) \\ \text{transform}: & \mathsf{trans}_{\mathsf{pr}}(S_1) \Rightarrow S_1' \\ & \mathsf{trans}_{\mathsf{pr}}(S_2) \Rightarrow S_2'\end{array}}{\mathsf{trans}_{\mathsf{pr}}(\ \texttt{if}\ E_c\ S_1\ S_2\ ) \quad \Rightarrow \quad \texttt{if}\ E_c\ S_1'\ S_2'}$$

만일 조건을 만족하지 않는다면, 부분 프로그램 구문(sub-statement)들이 동일한 규칙에 의해 변환된 ($\texttt{if}\ E_c\ S_1'\ S_2'$)로 변환한다. 간결한 설명을 위해 앞으로는

조건이 만족되지 않는 경우의 규칙은 생략할 것이다. 마찬가지로, 명시되지 않은 프로그램 구문(이번 경우에는, 분기문을 제외한 모든 프로그램 구문)에 대해서도 부분 프로그램 구문이 동일한 규칙에 의해 변환된 코드를 반환할 것이다(즉, $\mathsf{trans_{pr}}(\texttt{while}\ \ E_c\ S_b) \Rightarrow \texttt{while}\ \ E_c\ \mathsf{trans_{pr}}(S_b)$).

변환 규칙 표기를 간단히 하기 위해 다음과 같은 함수들을 사용한다. 모든 함수는 별도의 속 내용 분석(data-flow analysis)없이 겉 모습(syntax)만으로 계산 가능한 함수이다.

- newVar(): 기존에 없던 새로운 이름의 변수 반환
- updatedVars(S): 프로그램 구문 S안에서 x:=E의 형태로 값을 할당받는 모든 변수 x의 집합
- noAddrTaken(E): 프로그램 식 E 안의 모든 변수 x에 대해, 프로그램 전체에서 x의 주소를 취하는(&x)프로그램 식이 없음
- subst(S, p, p', E'): 프로그램 구문 S안의 메모리 접근식 p[E]을 p'[E+E']로 치환한 새로운 프로그램 구문 반환

다음 3.2, 3.3절에서, 변환 규칙 표기법을 사용하여 실제 코드 변환 규칙을 설명할 것이다. 각 코드 변환 규칙은 실행 의미를 보존하면서 분석 정확도가 향상될 수 있는, 그리고 값 비싼 분석 없이 코드의 겉 모습만을 보고 자동으로 변환을 수행할 수 있는 규칙이다.

3.2 반복문의 조건식 변환

조건식은 변수의 범위를 한정짓게 하여 반복문 내에서 분석의 정확도를 높여주는 중요한 힌트이다. 하지만 조건식이 대소 비교 연산자가 아니거나 실제보다 넓은 범위를 갖는다면 변수의 범위를 적절히 좁힐 수 없고, 이는 허위경보로 이어질 수

있다. 이때 코드의 모양만으로 변수의 범위를 명확히 알 수 있는 경우 조건식을 변환하여 분석 정확도를 향상시킬 수 있다.

3.2.1 대소 비교가 아닌 조건식

```
char *p = malloc(10);

int i = 0;
while (i != 10) {
  p[i] = 0;
  i++;
}
```

(a) 변환 전

```
char *p = malloc(10);

int i = 0;
while (i < 10) {
  p[i] = 0;
  i++;
}
```

(b) 변환 후

그림 3.1: 같지 않음($\neq$)연산자를 사용하는 조건식 변환 예시

조건식에 대소 비교 연산자가 아닌 다른 연산자를 사용하는 경우이다. 이 경우 분석기가 조건식에서 변수의 범위를 한정짓지 못하지만, 변수의 범위를 명백히 판단할 수 있는 경우에 코드를 변환하여 정확한 정보를 분석기에 전달할 수 있다.

$$\frac{\begin{array}{rl} \text{safety}: & \mathsf{noAddrTaken}(i) \\ & i \notin \mathsf{updatedVars}(S_1) \\ & (n_1 < n_2) \wedge (n_3 > 0) \wedge n_3 | (n_2 - n_1) \\ \text{transform}: & \mathsf{trans}_{\mathsf{lc}}(S_1) \Rightarrow S_1' \end{array}}{\mathsf{trans}_{\mathsf{lc}}\left(\begin{array}{l} i := n_1 \,; \\ \texttt{while}\ (i\ != n_2)\ (S_1\ ;\ i := i + n_3) \end{array} \right) \Rightarrow \begin{array}{l} i := n_1 \,; \\ \texttt{while}\ (i < n_2)\ (S_1'\ ;\ i := i + n_3) \end{array}}$$

매 반복마다 값이 일정하게 증가하는 변수 i에 대해서, i의 값이 간접적으로 변할 수 없고($\mathsf{noAddrTaken}(i)$) 증가 연산(increment operation)이외에 값이 변하는 일이 없으며($i \notin \mathsf{updatedVars}(S_1)$) n_2가 i의 시작값(n_1), 증가값(n_3)에 의해 실제로 도달하게 되는 값일 경우에(즉, $(n_1 < n_2) \wedge (n_3 > 0) \wedge n_3 | (n_2 - n_1)$) 조건식을 안전하게 대소 비교식으로 변환할 수 있다. 이 변환 방법은 증가값이 음수이거나

조건식 연산자가 나머지(modulo)연산, 혹은 비트곱셈(bitwise AND)연산인 경우에도 쉽게 적용될 수 있다.

Theorem 1. *프로그램* $S = (i := n_1 \; ; \texttt{while} \; (i \; != n_2) \; (S_1 \; ; \; i := i + n_3))$ *에 대하여, (1)* i*는 주소값을 통한 간접적인 값 할당이 없고 (2)* S_1*에서* i*의 값을 바꾸지 않고 (3)* $(n_1 < n_2) \wedge (n_3 > 0) \wedge n_3 | (n_2 - n_1)$ *일 때* $S' = \mathsf{trans}_{\mathsf{lc}}(S) = (i := n_1 \; ; \texttt{while} \; (i < n_2) \; (\mathsf{trans}_{\mathsf{lc}}(S_1) \; ; \; i := i + n_3))$*의 실행 의미는* S*와 동일하다.*

Proof. (3)에 의해 $n_2 = n_1 + d \times n_3$ 를 만족하는 0 이상의 정수 d가 존재한다. (1), (2)에 의해 i의 값이 $(i := i + n_3)$에 의해서만 바뀌기 때문에 반복문 안에서 i가 가질 수 있는 값은 $n_1 + k \times n_3$ (k는 0 이상의 정수) 의 형태로 표현될 수 있고, $k = d$일때 반복문이 종료되기 때문에 반복문 안에서 i가 가질 수 있는 값은 $\{n_1 + k \times n_3 \,|\, 0 \leq k < d\}$이다. 따라서 조건식을 $(i < n_2)$로 바꾸어도 프로그램 실제 실행시 반복문 안에서 i가 가질 수 있는 값의 범위는 동일하다. 따라서

case 1) $\mathsf{trans}_{\mathsf{lc}}(S_1) = S_1$인 경우: S'의 실행 의미는 S와 동일하다.

case 2) $\mathsf{trans}_{\mathsf{lc}}(S_1) \neq S_1$인 경우: 귀납 정의에 의해 $\mathsf{trans}_{\mathsf{lc}}(S_1)$의 실행 의미는 S_1과 동일하다. 또한 변환된 조건식이 메모리 상태에 영향을 주지 않으므로 S'의 실행 의미 또한 S와 동일하다.

□

3.2.2 실제 범위보다 넓은 조건식

```
1  char *p = malloc(16);
2
3  int i = 0;
4  while (i < 16) {
5    p[i] = 0;
6    p[i + 1] = 0;
7    p[i + 2] = 0;
8    p[i + 3] = 0;
9
10   i += 4;
11 }
```

(a) 변환 전

```
1  char *p = malloc(16);
2
3  int i = 0;
4  while (i <= 12) {
5    p[i] = 0;
6    p[i + 1] = 0;
7    p[i + 2] = 0;
8    p[i + 3] = 0;
9
10   i += 4;
11 }
```

(b) 변환 후

그림 3.2: 실제 변수의 범위보다 넓은 조건식 변환 예시

그림 3.2의 (a) 프로그램에서, 반복문 안에서 실제 `i`의 범위는 [0, 12]이다. 하지만 정적 분석기는 조건식을 통해 변수 `i`의 범위를 [0, 15]로 좁히고, 따라서 5번째 줄의 메모리 접근을 제외한 나머지 3개의 메모리 접근에서 허위경보가 발생한다. 이 경우 변수 `i`의 초기값, 증가값 및 조건식의 범위를 이용해 실제 변수가 가질 수 있는 범위의 조건식으로 바꿔줄 수 있다.

$$
\begin{array}{rl}
\text{safety}: & \mathsf{noAddrTaken}(i) \\
& i \notin \mathsf{updatedVars}(S_1) \\
& (n_1 < n_2) \wedge (n_3 > 0) \\
\text{transform}: & \mathsf{trans}_{\mathsf{lc}}(S_1) \Rightarrow S_1' \\
& n_2' = \mathsf{max}(\{n | n < n_2 \wedge n = n_1 + n_3 \times k, k \in \{0, 1, 2, \cdots\}\})
\end{array}
$$

$$
\mathsf{trans}_{\mathsf{lc}}\left(\begin{array}{l} i := n_1\ ; \\ \texttt{while}\ (i < n_2)\ (S_1\ ;\ i := i + n_3) \end{array}\right) \Rightarrow \begin{array}{l} i := n_1\ ; \\ \texttt{while}\ (i <= n_2')\ (S_1'\ ;\ i := i + n_3) \end{array}
$$

매 반복마다 값이 일정하게 증가하는 변수 i에 대해서, i의 값이 간접적으로 변할 수 없고($\mathsf{noAddrTaken}(i)$) 증가 연산(increment operation)이외에 값이 변하는 일

이 없을 때($i \notin \mathsf{updatedVars}(S_1)$), i의 초깃값, 증가값 및 조건식을 만족하는 가장 큰 수($\mathsf{max}(\{n|n < n_2 \wedge n = n_1 + n_3 \times k, k \in \{0, 1, 2, \cdots\}\})$), 즉 반복문 실행중 i가 실제로 가질 수 있는 가장 큰 수 n_2'를 사용하여 조건식의 범위를 좁힐 수 있다.

Theorem 2. *프로그램* $S = (i := n_1 \; ; \texttt{while} \; (i < n_2) \; (S_1 \; ; i := i + n_3))$ *에 대하여, (1)* i*는 주소값을 통한 간접적인 값 할당이 없고 (2)* S_1*에서* i*의 값을 바꾸지 않고 (3)* $(n_1 < n_2) \wedge (n_3 > 0)$ *일 때* $S' = \mathsf{trans}_{\mathsf{lc}}(S) = (i := n_1 \; ; \texttt{while} \; (i <= n_2')$ $(\mathsf{trans}_{\mathsf{lc}}(S_1) \; ; i := i + n_3))$*의 실행 의미는* S*와 동일하다*($n_2' = \mathsf{max}(\{n|n < n_2 \wedge n = n_1 + n_3 \times k, k \in \{0, 1, 2, \cdots\}\})$).

Proof. (1), (2)에 의해 i의 값이 $(i := i + n_3)$에 의해서만 바뀌기 때문에 반복문 안에서 i가 가질 수 있는 값은 $n_1 + k \times n_3$ (k는 0 이상의 정수) 의 형태로 표현될 수 있고, 그 중 반복문 안에서 실제로 i가 가질 수 있는 가장 큰 수 n_2'은 조건식을 만족하는 $\mathsf{max}(\{n|n < n_2 \wedge n = n_1 + n_3 \times k, k \in \{0, 1, 2, \cdots\}\})$이다. 조건식을 $(i <= n_2')$로 바꾸어도 S'의 반복문 안에서 i가 가질 수 있는 값의 범위는 동일하다. 따라서

case 1) $\mathsf{trans}_{\mathsf{lc}}(S_1) = S_1$인 경우: S'의 실행 의미는 S와 동일하다.

case 2) $\mathsf{trans}_{\mathsf{lc}}(S_1) \neq S_1$인 경우: 귀납 정의에 의해 $\mathsf{trans}_{\mathsf{lc}}(S_1)$의 실행 의미는 S_1과 동일하다. 또한 변환된 조건식이 메모리 상태에 영향을 주지 않으므로 S'의 실행 의미 또한 S와 동일하다.

□

3.3 반복문 안의 메모리 접근식 변환

```
1  char *p = malloc(16);
2
3  int i;
4  for (i = 0; i < 4; i++) {
5    p[0] = 0;
6    p[1] = 0;
7    p[2] = 0;
8    p[3] = 0;
9
10   p += 4;
11 }
```

(a) 변환 전

```
1  char *p = malloc(16);
2
3  int i;
4  char *p' = p;
5  for (i = 0; i < 4; i++) {
6    p'[4 * i + 0] = 0;
7    p'[4 * i + 1] = 0;
8    p'[4 * i + 2] = 0;
9    p'[4 * i + 3] = 0;
10
11   p += 4;
12 }
```

(b) 변환 후

그림 3.3: 반복문 안의 메모리 접근식 변환 예시

그림 3.3 (a) 프로그램에서, [0, 3] 범위를 갖는 변수 i는 5-8번째 줄의 메모리 접근식에 대해 분석기에 어떠한 힌트도 줄 수 없다. i를 오프셋으로 사용하지 않고 포인터 변수 p를 직접 증가시켜가며 메모리 접근을 하기 때문이다. 이 경우 그림 3.3 (b) 와 같이 메모리 접근식의 오프셋을 반복문의 조건식에서 사용되는 변수 i와 연관지어서 분석의 정확도를 향상시킬 수 있다.

$$\frac{\begin{array}{rl} \text{safety}: & \mathsf{noAddrTaken}(i) \wedge i \notin \mathsf{updatedVars}(S_1) \\ & \mathsf{noAddrTaken}(p) \wedge p \notin \mathsf{updatedVars}(S_1) \\ \text{transform}: & p' = \mathsf{newVar}(),\ \mathsf{trans}_{\mathsf{lb}}(S_1) \Rightarrow S_1' \\ & S_1'' = \mathsf{subst}(S_1', p, p', (n_3 * (i - n_1)/n_4)) \end{array}}{\mathsf{trans}_{\mathsf{lb}}\left(\begin{array}{l} i := n_1\ ; \\ \texttt{while}\ (i < n_2)\ (S_1;\ p := p + n_3;\ i := i + n_4) \end{array}\right) \Rightarrow \begin{array}{l} i := n_1;\ p' := p; \\ \texttt{while}\ (i < n_2)\ (S_1'';\ p := p + n_3;\ i := i + n_4) \end{array}}$$

정수 변수 i와 포인터 변수 p에 대해서, i와 p의 값이 간접적으로 변하지 않고 $(\mathsf{noAddrTaken}(i), \mathsf{noAddrTaken}(p))$ 증가 연산(increment operation)이외에 값이

변하는 일이 없을 때($i, p \notin$ updatedVars(S_1)) 프로그램 구문S안의 메모리 접근식 $p[E]$는 $p'[E+(n_3 \star (i-n_1)/n_4)]$로 변환될 수 있다.($p'$은 기존에 없던 새로운 변수, 반복문 직전의 p의 주소값을 가짐)

Theorem 3. *프로그램 $S = (i := n_1$; while $(i < n_2)$ $(S_1; p := p+n_3; i := i+n_4))$ 에 대하여, (1) i와 p는 주소값을 통한 간접적인 값 할당이 없고 (2) S_1에서 i와 p의 값이 바뀌지 않을때 $S' = \mathsf{trans}_{\mathsf{lb}}(S) = (i := n_1$; $p' := p$; while $(i < n_2)$ $(S_1''$; $p := p+n_3$; $i := i+n_4))$의 실행 의미는 S와 동일하다. ($S_1'' = \mathsf{subst}(\mathsf{trans}_{\mathsf{lb}}(S_1), p, p', (n_3 \star (i-n_1)/n_4))$)*

Proof. (1), (2)에 의해에 p의 값은 $(p := p+n_3)$에서, 그리고 i의 값은 $(i := i+n_4)$에서만 변한다. 반복문 직전의 p의 주소값을 l이라고 할 때 반복문 안에서의 p와 i의 값은 다음과 같다.

iter	p	i
0	l	n_1
1	$l+n_3$	n_1+n_4
2	$l+n_3 \star 2$	$n_1+n_4 \star 2$
...	...	...
k	$l+n_3 \star k$	$n_1+n_4 \star k$

따라서 반복문 안에서 k의 값은 $(i-n_1)/n_4$와 동일하고, p의 주소값은 $(l+n_3 \star (i-n_1)/n_4)$로 표현할 수 있다. 변수 p'의 주소값은 반복문 직전의 p의 주소값인 l이기 때문에 반복문 안에서의 메모리 접근식 $p[E]$이 참조하는 주소는 $p'[E+(n_3 \star (i-n_1)/n_4)]$의 주소와 동일하다. 따라서

case 1) $\mathsf{trans}_{\mathsf{lb}}(S_1) = S_1$인 경우: S'의 실행 의미는 S와 동일하다.

case 2) $\mathsf{trans}_{\mathsf{lb}}(S_1) \neq S_1$인 경우: 귀납 정의에 의해 $\mathsf{trans}_{\mathsf{lb}}(S_1)$의 실행 의미는 S_1과 동일하다. 또한 변환된 식에 의한 메모리 상태의 변화는 $\mathsf{trans}_{\mathsf{lb}}(S_1)$

에서 사용되지 않는 새로운 변수 p'이 추가된 것 뿐이기 때문에 S'의 실행 의미는 S와 동일하다.

$\square$

제 4 장 실험 결과

2장에서 소개한 기준(baseline) 정적 분석기 스패로우의 파서(parser)에 3장의 코드 변환 규칙을 구현한 변환기를 장착하여 총 100개의 오픈소스(open-source) 프로그램을 대상으로 실험을 진행하였다. 코드 변환 적용 전후의 정적 분석 결과를 비교하고, 코드 변환에 걸린 시간과 분석 시간의 변화를 측정하였다(표 4.1). 각 프로그램 별로 적용된 코드 변환의 종류와, 허위 경보가 제거되었을 시 어떤 변환에 의해 제거 되었는지 집계하였다(표 4.2). 코드 변환 후 제거된 경보를 확인한 결과 모두 허위 경보였으며, 코드 변환 후 새로운 경보가 생기는 경우는 없었다.

실험 결과 소스코드를 변환 없이 정적 분석한 결과와, 같은 소스코드에 앞서 기술한 변환 규칙을 적용한 후 정적 분석한 결과를 비교하였다(표 4.1). 코드를 변환한 후 정적 분석한 경우에 전체 벤치마크 100개의 프로그램 중 6개의 프로그램에서 총 17개의 허위 경보가 제거되었다. 각 벤치마크에서 제거된 허위 경보 비율의 평균은 0.51%이고(전체 경보의 개수를 기준으로 하면 13,575 → 13,558, 0.13%), 분석 시간은 코드 변환 과정에서 0.6%, 그리고 분석 과정에서 0.1%로 총 0.7% 증가하였다.

반복문 안의 메모리 접근식 변환이 16회로 분석 정확도 향상에 가장 많이 기여했고, 변환 횟수 대비 정확도 향상 비율은 15.8%(101회 변환, 16회 기여)였다. 반복문의 조건식 변환은 1회의 분석 정확도 향상을 기록했고, 변환 횟수 대비 정확도 향상 비율은 12.5% (8회 변환, 1회 기여)였다(표 4.2).

Program	LOC	baseline		w/ transformation				alr-red	Overhead	
		#alr	time (ms)	#alr	trans (ms)	main (ms)	total (ms)	(%)	trans (%)	main (%)
4digits	843	7	89.9	7	0.5	88.6	89.1	0.00	0.6	-1.4
a52dec	5,415	6	588.0	6	4.8	548.2	553.1	0.00	0.8	-6.8
aalib-1.4p5	13,132	126	8404.8	126	7.0	8434.2	8441.2	0.00	0.1	0.4
acpi-1.4	1,462	20	217.6	20	1.2	216.4	217.6	0.00	0.6	-0.5
acpi-1.7	1,479	20	220.7	20	1.1	218.7	219.9	0.00	0.5	-0.9

acpid-2.0.21	4,415	23	711.7	23	5.1	792.4	797.5	0.00	0.7	11.3
adjtimex-1.29	5,325	92	691.5	92	5.3	682.9	688.2	0.00	0.8	-1.2
aes2501-wy-0.1	2,181	28	330.9	**27**	2.2	316.8	319.0	**3.57**	0.7	-4.3
aespipe-2.3e.c	5,643	183	2937.8	183	6.4	2935.3	2941.8	0.00	0.2	-0.1
aewan-1.0.01	8,705	383	2391.6	383	9.6	2402.9	2412.5	0.00	0.4	0.5
afterstep-2.2.12	3,781	162	301.8	162	1.9	300.2	302.1	0.00	0.6	-0.5
agedu-8642	10,029	703	6037.6	703	14.5	5967.2	5981.7	0.00	0.2	-1.2
aggregate-1.6	692	9	107.0	9	0.5	107.9	108.4	0.00	0.5	0.8
aha-0.4.6.1	1,075	42	264.4	42	1.0	264.4	265.3	0.00	0.4	0.0
alac-decoder-0.1.3	2,388	66	404.0	66	2.7	402.1	404.8	0.00	0.7	-0.5
alac-decoder-0.2.0	2,557	94	460.3	94	2.9	453.9	456.9	0.00	0.6	-1.4
alevt-1.6.2	453	13	47.3	13	0.4	48.3	48.8	0.00	0.9	2.1
altermime-0.3.10	10,786	232	5944.0	232	11.9	6115.6	6127.5	0.00	0.2	2.9
ample-0.5.7	4,848	123	1545.4	123	5.5	1553.8	1559.3	0.00	0.4	0.5
ampliconnoise-1.29	1,708	135	205.3	135	2.6	207.3	209.9	0.00	1.3	1.0
amtterm-1.3	1,405	14	184.2	14	1.3	189.6	190.9	0.00	0.7	2.9
and-1.2.2	1,594	63	257.0	63	2.0	315.8	317.7	0.00	0.8	22.9
aoetools-30	1,232	6	131.2	6	0.8	122.7	123.5	0.00	0.6	-6.5
aoeui-1.6	12,708	176	2422.3	176	13.4	2420.8	2434.2	0.00	0.6	-0.1
apg-2.2.3	5,193	29	938.4	29	5.7	954.1	959.9	0.00	0.6	1.7
apng2gif-1.5	3,685	265	738.8	265	4.8	784.6	789.4	0.00	0.6	6.2
apngdis-2.5	2,555	178	514.1	178	4.2	545.3	549.4	0.00	0.8	6.1
apngopt-1.2	4,665	330	1183.0	330	7.8	1263.0	1270.7	0.00	0.7	6.8
aprsdigi-2.4.4	1,192	37	75.8	37	1.0	76.4	77.4	0.00	1.3	0.8
archimedes	8,310	100	4430.2	100	16.6	4397.4	4414.0	0.00	0.4	-0.7
asp-1.8	1,733	6	157.6	6	1.7	158.2	159.9	0.00	1.1	0.4
aspic-1.05	9,355	1,361	3659.3	1,361	9.7	3725.0	3734.6	0.00	0.3	1.8
atsar-1.7	4,544	162	690.1	162	4.9	705.0	709.9	0.00	0.7	2.1
autolog-0.40	1,664	105	301.7	105	1.9	310.0	311.9	0.00	0.6	2.8
avr-evtd-1.7.7	1,620	13	196.4	13	1.4	195.3	196.7	0.00	0.7	-0.6
axel-2.4	3,951	349	1285.9	349	4.2	1293.5	1297.6	0.00	0.3	0.6
balance-3.54	2,582	136	768.2	136	2.3	754.7	757.0	0.00	0.3	-1.8
barcode-0.96	4,343	291	1148.8	291	5.4	1129.6	1135.0	0.00	0.5	-1.7
barcode-0.98	5,015	272	1384.1	272	6.4	1398.8	1405.2	0.00	0.5	1.1
bchunk-1.2.0	738	7	151.3	7	0.7	151.8	152.5	0.00	0.4	0.3
beep-1.2.2	401	2	33.4	2	0.3	33.1	33.4	0.00	1.0	-1.1
beep-1.3	515	2	45.2	2	0.3	45.0	45.3	0.00	0.7	-0.5
bfbtester-2.0.1	3,185	48	332.7	48	2.5	343.5	346.0	0.00	0.7	3.2
binstats-1.08	484	13	52.5	13	0.4	51.6	51.9	0.00	0.7	-1.8
birthday-1.6.2	2,318	201	754.4	201	2.7	754.8	757.5	0.00	0.4	0.1
bkhive-1.1.1	649	6	69.9	6	0.6	71.0	71.6	0.00	0.8	1.6
bmf-0.9.4	3,079	68	407.8	68	2.4	417.0	419.3	0.00	0.6	2.3
boa-0.94.14rc21	10,496	332	4980.1	332	11.1	4980.9	4992.0	0.00	0.2	0.0
bombardier-0.8.3	1,723	36	291.3	36	1.6	300.0	301.6	0.00	0.5	3.0
bootchart-0.90.2	640	5	98.6	5	0.4	97.3	97.7	0.00	0.4	-1.3
bopm-3.1.3	9,897	320	2005.5	320	8.6	2046.9	2055.5	0.00	0.4	2.1
bottlerrocker-0.05b3	2,166	46	457.8	46	2.7	456.0	458.6	0.00	0.6	-0.4
brother-cups-wrapper-bh7-1.0.0-10	922	58	258.7	**57**	0.7	240.4	241.1	**1.72**	0.3	-7.0
brother-cups-wrapper-common-1.0.0-10	922	58	249.5	**57**	0.7	240.5	241.2	**1.72**	0.3	-3.6

brutefir-1.0f	182	4	15.8	4	0.1	14.5	14.6	0.00	0.8	-8.4
buthead-1.0	123	6	7.9	6	0.1	7.8	7.9	0.00	1.0	-1.4
c2050-0.3b	599	19	54.8	19	0.4	55.4	55.8	0.00	0.8	0.9
cabextract-1.4	11,582	38	968.6	38	12.0	954.3	966.3	0.00	1.2	-1.5
canlock-2b	1,247	40	187.2	**29**	1.8	191.1	192.9	**27.50**	0.9	2.1
cciss-vol-status-1.11	3,153	72	766.1	72	2.5	762.2	764.8	0.00	0.3	-0.5
ccontrol-1.0	2,853	76	1116.7	76	2.1	1108.9	1111.1	0.00	0.2	-0.7
cd-discid-1.1	203	1	15.6	1	0.2	14.3	14.5	0.00	1.0	-7.9
cd-discid-1.4	256	4	23.4	4	0.2	21.8	22.0	0.00	0.8	-6.8
cddlib-094g	28,225	266	11028.3	266	38.3	10330.9	10369.2	0.00	0.3	-6.3
cdi2iso-0.1	188	11	21.5	11	0.2	22.1	22.3	0.00	0.9	2.8
cdparanoia-3.10.2	14,127	200	11545.7	200	14.3	11592.2	11606.5	0.00	0.1	0.4
cgiemail-1.6	2,274	86	393.8	86	2.4	392.1	394.5	0.00	0.6	-0.4
chemtool-1.6.14	5,442	157	5531.3	157	6.1	5441.3	5447.3	0.00	0.1	-1.6
chkrootkit-0.49	1,528	1	120.9	1	1.1	119.6	120.6	0.00	0.9	-1.1
chordii-4.3	4,089	40	3222.8	40	4.6	3259.7	3264.2	0.00	0.1	1.1
ciso-1.0.0	556	8	60.6	8	0.5	60.8	61.3	0.00	0.8	0.3
clamsmtp-1.10	4,693	35	911.9	35	4.8	903.2	908.0	0.00	0.5	-1.0
cliquer-1.21	5,156	212	1001.7	212	6.3	991.7	998.0	0.00	0.6	-1.0
cmdpack-1.03	1,837	85	248.9	85	1.6	249.3	250.9	0.00	0.7	0.2
cmph-0.9	13,869	1,466	6306.4	1,466	16.2	6326.9	6343.1	0.00	0.3	0.3
codegroup-19981025	898	13	81.0	**11**	0.6	82.5	83.2	**15.38**	0.8	1.9
combine-0.3.3	11,686	699	7504.9	699	24.2	7523.4	7547.6	0.00	0.3	0.2
comgt-0.32	7,101	346	3050.5	346	3.6	2990.3	2993.9	0.00	0.1	-2.0
compartment-1.1.0	887	50	197.4	50	0.4	194.2	194.7	0.00	0.2	-1.6
confget-1.02	658	2	46.0	2	0.5	45.9	46.3	0.00	1.0	-0.2
consol-calculator	591	21	65.4	21	0.7	63.6	64.3	0.00	1.0	-2.8
cpipe-3.0.1	1,376	19	104.5	19	1.4	105.2	106.6	0.00	1.3	0.6
cpulimit-2.0	1,015	12	166.3	12	0.7	165.4	166.2	0.00	0.4	-0.5
cracklib2-2.9.1	3,175	10	414.2	10	2.3	413.5	415.9	0.00	0.6	-0.1
cramfsswap-1.4.1	429	48	76.3	48	0.4	76.3	76.7	0.00	0.5	0.0
crashme-2.4	2,458	27	763.0	27	2.5	765.1	767.6	0.00	0.3	0.3
cron-3.0pl1	4,484	96	1375.3	**95**	4.9	1386.5	1391.4	**1.04**	0.4	0.8
cstream-3.0.0	2,309	13	407.5	13	2.3	401.1	403.4	0.00	0.6	-1.6
csv2latex-0.18	960	8	106.5	8	0.8	106.6	107.4	0.00	0.7	0.1
cutils-1.6	4,020	97	492.4	97	2.9	508.9	511.9	0.00	0.6	3.4
cutter-1.03	1,057	9	148.8	9	0.7	148.8	149.5	0.00	0.5	0.0
dancer-xml-0.8.2.1	874	50	105.5	50	0.6	105.3	106.0	0.00	0.6	-0.1
dbview-1.0.4	588	16	58.4	16	0.4	57.9	58.3	0.00	0.7	-0.8
ddns3-client-1.8	1,656	59	198.6	59	1.9	199.5	201.4	0.00	0.9	0.4
dds2tar-2.5.2	3,495	132	861.1	132	2.7	831.6	834.3	0.00	0.3	-3.4
debram-1.0.3	4,837	190	2924.9	190	3.5	2933.9	2937.4	0.00	0.1	0.3
depqbf-2.0	18,770	1,114	3880.0	1,114	22.2	3927.0	3949.2	0.00	0.6	1.2
dhcp-helper-1.1	996	13	227.3	13	0.7	232.3	233.0	0.00	0.3	2.2
dhcping-1.2	784	7	84.8	7	0.5	86.3	86.9	0.00	0.6	1.8
dhex-0.68	15,970	135	3404.1	135	13.1	3377.9	3391.0	0.00	0.4	-0.8
Total	395,654	13,575	132,852	13,558	428	132,576	133004.3	0.51	0.6	0.1

표 4.1: 실험 결과. **LOC**: 코드의 줄 수. **baseline**: 코드 변환 없는 기본 분석기. **w/ transformation**: 코드 변환을 포함한 분석기. **# alr**: 경보 갯수. **trans, main**: 코드 변환, 본 분석 시간. **alr-red**: 경보 감소율. **Overhead**: 코드 변환, 본 분석에서 추가로 소요된 시간.

Program	LOC	Baseline	w/ Trans	Alarm	Transformed by		Remark
		#alarm	#alarm	reduc-tion(%)	loop cond	loop body	
4digits	843	7	7	0.00			
a52dec	5,415	6	6	0.00			
aalib-1.4p5	13,132	126	126	0.00			
acpi-1.4	1,462	20	20	0.00			
acpi-1.7	1,479	20	20	0.00			
acpid-2.0.21	4,415	23	23	0.00			
adjtimex-1.29	5,325	92	92	0.00			
aes2501-wy-0.1	2,181	28	**27**	**3.57**	3	2	1 alarm removed by Loop Cond trans
aespipe-2.3e.c	5,643	183	183	0.00		8	
aewan-1.0.01	8,705	383	383	0.00			
afterstep-2.2.12	3,781	162	162	0.00			
agedu-8642	10,029	703	703	0.00			
aggregate-1.6	692	9	9	0.00			
aha-0.4.6.1	1,075	42	42	0.00			
alac-decoder-0.1.3	2,388	66	66	0.00			
alac-decoder-0.2.0	2,557	94	94	0.00			
alevt-1.6.2	453	13	13	0.00		1	
altermime-0.3.10	10,786	232	232	0.00		3	
ample-0.5.7	4,848	123	123	0.00			
ampliconnoise-1.29	1,708	135	135	0.00			
amtterm-1.3	1,405	14	14	0.00			
and-1.2.2	1,594	63	63	0.00			
aoetools-30	1,232	6	6	0.00	1	1	
aoeui-1.6	12,708	176	176	0.00	2	4	
apg-2.2.3	5,193	29	29	0.00	1	1	
apng2gif-1.5	3,685	265	265	0.00		19	
apngdis-2.5	2,555	178	178	0.00		16	
apngopt-1.2	4,665	330	330	0.00		18	
aprsdigi-2.4.4	1,192	37	37	0.00			
archimedes	8,310	100	100	0.00			
asp-1.8	1,733	6	6	0.00			
aspic-1.05	9,355	1361	1,361	0.00		4	
atsar-1.7	4,544	162	162	0.00			
autolog-0.40	1,664	105	105	0.00			
avr-evtd-1.7.7	1,620	13	13	0.00			
axel-2.4	3,951	349	349	0.00			
balance-3.54	2,582	136	136	0.00			
barcode-0.96	4,343	291	291	0.00		1	
barcode-0.98	5,015	272	272	0.00		1	
bchunk-1.2.0	738	7	7	0.00			
beep-1.2.2	401	2	2	0.00			
beep-1.3	515	2	2	0.00			
bfbtester-2.0.1	3,185	48	48	0.00		1	
binstats-1.08	484	13	13	0.00			
birthday-1.6.2	2,318	201	201	0.00			
bkhive-1.1.1	649	6	6	0.00			
bmf-0.9.4	3,079	68	68	0.00			
boa-0.94.14rc21	10,496	332	332	0.00			

bombardier-0.8.3	1,723	36	36	0.00			
bootchart-0.90.2	640	5	5	0.00			
bopm-3.1.3	9,897	320	320	0.00		2	
bottlerrocker-0.05b3	2,166	46	46	0.00			
brother-cups-wrapper-bh7-1.0.0-10	922	58	**57**	**1.72**		1	1 alarm removed by Loop Body trans
brother-cups-wrapper-common-1.0.0-10	922	58	**57**	**1.72**		1	1 alarm removed by Loop Body trans
brutefir-1.0f	182	4	4	0.00			
buthead-1.0	123	6	6	0.00			
c2050-0.3b	599	19	19	0.00			
cabextract-1.4	11,582	38	38	0.00		3	
canlock-2b	1,247	40	**29**	**27.50**		5	11 alarms removed by Loop Body trans
cciss-vol-status-1.11	3,153	72	72	0.00			
ccontrol-1.0	2,853	76	76	0.00			
cd-discid-1.1	203	1	1	0.00			
cd-discid-1.4	256	4	4	0.00			
cddlib-094g	28,225	266	266	0.00			
cdi2iso-0.1	188	11	11	0.00			
cdparanoia-3.10.2	14,127	200	200	0.00			
cgiemail-1.6	2,274	86	86	0.00			
chemtool-1.6.14	5,442	157	157	0.00		2	
chkrootkit-0.49	1,528	1	1	0.00			
chordii-4.3	4,089	40	40	0.00			
ciso-1.0.0	556	8	8	0.00			
clamsmtp-1.10	4,693	35	35	0.00			
cliquer-1.21	5,156	212	212	0.00			
cmdpack-1.03	1,837	85	85	0.00			
cmph-0.9	13,869	1466	1,466	0.00		2	
codegroup-19981025	898	13	**11**	**15.38**	1	2	2 alarms removed by Loop Body trans
combine-0.3.3	11,686	699	699	0.00			
comgt-0.32	7,101	346	346	0.00			
compartment-1.1.0	887	50	50	0.00			
confget-1.02	658	2	2	0.00			
consol-calculator	591	21	21	0.00			
cpipe-3.0.1	1,376	19	19	0.00			
cpulimit-2.0	1,015	12	12	0.00			
cracklib2-2.9.1	3,175	10	10	0.00			
cramfsswap-1.4.1	429	48	48	0.00			
crashme-2.4	2,458	27	27	0.00			
cron-3.0pl1	4,484	96	**95**	**1.04**		1	1 alarm removed by Loop Body trans
cstream-3.0.0	2,309	13	13	0.00			
csv2latex-0.18	960	8	8	0.00			
cutils-1.6	4,020	97	97	0.00		1	
cutter-1.03	1,057	9	9	0.00			
dancer-xml-0.8.2.1	874	50	50	0.00			
dbview-1.0.4	588	16	16	0.00			
ddns3-client-1.8	1,656	59	59	0.00			

dds2tar-2.5.2	3,495	132	132	0.00			
debram-1.0.3	4,837	190	190	0.00		1	
depqbf-2.0	18,770	1114	1,114	0.00			
dhcp-helper-1.1	996	13	13	0.00			
dhcping-1.2	784	7	7	0.00			
dhex-0.68	15,970	135	135	0.00			
Total	395,654	13,575	13,558	0.51	8	101	

표 4.2: 변환 규칙 적용률. **LOC**: 코드의 줄 수. **Baseline**: 코드 변환 없는 기본 분석기. **w/ Trans**: 코드 변환을 포함한 분석기. **# alarm**: 경보 갯수. **Alarm reduction**: 경보 감소율. **loop cond, loop body**: 반복문 조건식, 반복문 안의 메모리 접근식 변환 횟수.

제 5 장 논의 및 결론

5.1 코드 변환을 이용한 분석 정확도 향상의 한계

본 연구를 통해 달성하고자 하였던 코드 변환을 이용한 분석 정확도 향상이 충분히 이루어지지 못한 원인을 분석한다. 4장의 실험 결과에서 평균 0.51%의 정확도 향상과 0.7%의 분석 시간 증가를 확인할 수 있었는데, 이는 비용 대비 성능 향상은 준수하지만 제거에 성공한 허위 경보의 절대적인 양은 많지 않음을 나타낸다. 그 원인으로 보수적인 코드 변환 규칙으로 인해 변환되지 않는 경우가 있음을, 그리고 코드가 변환되었음에도 정확도가 개선되지 않는 경우가 있음을 확인하였다.

5.1.1 융통성 없는 변환 규칙

```
1 len = length(p);
2 for (i = 0; i != len; i++) {
3   p[i] = 0;
4 }
```

그림 5.1: 변환을 포기하는 경우 예시

안전한 변환을 위한 융통성 없는 변환 규칙으로 인해 많은 경우에 변환을 포기한다. 변환된 코드가 같은 실행 의미를 갖는 경우에만 변환을 수행해야 하는데, 코드의 겉모습 만으로 이를 파악하기 위해서 변환 규칙은 보수적으로 안전함을 확인한다. 그림 5.1의 반복문에서 조건식 (`i != len`)를 (`i < len`)로 변환할 수 없다. 변수 `i`의 종료값이 상수가 아니기 때문이다. 만일 초기값이 종료값보다 큰 수일 경우 프로그램의 실행 의미가 달라져 변환을 할 수 없는데, 상수가 아닐 경우 이를 값 분석 없이 확인할 수 없다. 코드의 모양 만으로 변환의 안전함을 보장할 수 없을때 변환을 포기하는 보수적인 규칙으로 인해, 설령 간단한 값 분석 만으로

확인 가능한 간단한 경우에도 변환을 포기하게 된다.

5.1.2 코드 변환이 정확도를 향상시키지 못하는 경우

```
1 // n -> [-oo,+oo]
2 for (i = 0; i < n; i++) {
3   p[0] = 0;
4   p += 1;
5 }
```

(a) 변환 전

```
1 // n -> [-oo,+oo]
2 p' = p;
3 for (i = 0; i < n; i++) {
4   p'[i] = 0;
5   p += 1;
6 }
```

(b) 변환 후

그림 5.2: 코드 변환이 정확도를 향상시키지 못하는 경우 예시

코드 변환 그 자체 만으로는 정확도 향상을 보장할 수 없다. 코드의 모양만으로 정확도 향상 여부를 어림잡아 변환을 수행하기 때문에 변환된 코드가 여전히 정확하게 분석되지 않을 수 있다. 그림 5.2의 경우, 변수 n의 요약 값이 정확히 분석되지 않았기 때문에 변환된 코드에서 메모리 접근식의 오프셋 i 또한 부정확하여 결과적으로 개선되지 못한 분석 결과를 내게 된다. 즉, 코드 변환을 통해 정확도 향상의 여지를 만들어 줄 수 있지만, 실제 정확도 향상의 여부는 분석기의 성능에 좌우될 수 있다.

5.2 가능한 다른 코드 변환

본 연구에 포함되지 않은 다른 가능한 코드 변환 방법을 살펴본다. 2.2절에 언급된, 분기문과 함수 호출문에서 정확도를 잃는 경우 어떤 코드 변환이 가능한지 확인하고, 또한 본 연구에 포함되지 않은 이유를 알아본다.

5.2.1 분기문 변환

분기문 변환의 경우 실효성이 부족한 것으로 판단되어 연구 내용에서 제외되었다. 서론에서 소개한 간단한 코드 변환의 예시(그림 1.1)와 같이 분기문의 분석 정확도를 향상시키는 코드 변환이 가능하다. 하지만 실험을 통해 성능을 확인한 결과, 본 연구에서 다루는 범위를 벗어난 메모리 접근 분석에는 효용이 없었다. 4장과 동일한 벤치마크에 실험을 하였을 때, 총 112개의 분기문 변환이 있었지만 이를 통해 제거된 허위 경보는 없었다.

5.2.2 함수 호출문 변환

```
1 int *malloc_int (int len) {
2   return malloc(sizeof(int)*len);
3 }
4
5 void foo () {
6   // int *p = malloc_int(5);  // size->[5,10]
7   // int *q = malloc_int(10);  // size->[5,10]
8   int *p = malloc(sizeof(int)*5);  // size->[5,5]
9   int *q = malloc(sizeof(int)*10);  // size->[10,10]
10
11  p[4] = 0;  //size->[5,5], offset->[4,4] : safe
12  q[9] = 0;  //size->[10,10], offset->[9,9] : safe
13 }
```

그림 5.3: 함수 들여쓰기가 적용된 코드 예시

함수 호출문의 경우 코드 변환이 아닌 다른 효율적인 분석 연구[4]가 있기 때문에 본 연구에 포함되지 않았다. 그림 5.3은 2.2.2절의 그림 2.3 프로그램에 함수 들여쓰기(function inlining)를 적용한 예시이다. 함수 들여쓰기를 통해, 함수 호출 맥락을 고려하는(context-sensitive) 분석 기법[5, 6]과 마찬가지로 각각의 함수

호출을 별개로 인식하여 분석이 정확해질 수 있다. 하지만 이러한 방법을 통해 모든 함수 호출을 개별적으로 분석하게 되면 얻게 되는 정확도 향상에 비해 분석 비용 증가량이 더욱 크기 때문에 효율적인 분석이 어렵다. 저렴한 사전 분석을 통해 선별적으로 이를 적용하는 연구[4]에서는 개별적인 분석이 필요한 함수 호출 지점만을 선택하여 효율적으로 분석 정확도를 향상시킬수 있음을 보였다. 본 연구에서는 코드의 겉모습만을 분석해서 코드 변환 여부를 결정하기 때문에 기존 연구보다 더 효율적으로 대상을 선별하기 어려울 것이라 판단하여, 함수 호출문 코드 변환을 연구의 범위로 포함하지 않았다.

5.3 관련 연구

```
1 if ((:[v] = :[fn](:[args])) === false)
```

(a) 변환 대상 패턴

```
1 :[v] = :[fn](:[args]);
2 if (:[v] === false )
```

(b) 변환 적용 패턴

그림 5.4: 코드 변환 패턴의 예시

분석기 사용자가 입력한 변환 패턴으로 코드를 변환하여 허위 경보를 줄이는 기법[7]이 연구되었다. 즉, 정적 분석기 사용자가 발견한 허위 경보를 줄일 수 있는 패턴으로 분석기 실행 전 분석 대상 코드를 변환하는 방법이다. 그림 5.4는 변수의 할당과 반환값 확인이 모두 분기문의 조건식 안에서 이루어 지는 코드를 분리하는 패턴이다. 변환기는 (a)의 패턴과 일치하는 모든 프로그램 식을 (b)와 같이 변환한다. 이처럼 분석기 사용자는 자신이 사용하는 분석기의 성능과 분석 대상 프로그램의 특성에 따라 원하는 패턴을 입력하여 불필요한 경보를 제거할 수 있다. 분석기 개발자가 아닌 사용자가 손쉽게 이용할 수 있도록 설계 되었기 때문에 본 연구처럼 각 패턴의 안전성을 따지지 않지만, 다양한 상황에 대해 유연하게

적용할 수 있는 장점이 있고 실험적으로 유용성을 확인하였다.

5.4 결론

본 연구에서는 코드 변환을 통해 저렴한 비용으로 정적 분석 정확도를 향상하는 방법과 그 실효성을 조사하였다. 분석 기술을 정교화 하여 정확한 분석 결과를 얻는 기존 방법과 달리, 분석 대상 소스코드를 분석에 적합하게 변환하여 분석기의 성능 향상 없이 허위 경보를 제거하였다. 변환 과정은 별도의 값비싼 분석 없이 코드의 모양만으로 변환의 안전함과 적합도를 판단하기 때문에 비용이 저렴하지만, 정확도 향상 효과 또한 크지 않았다.

변환기는 0.7%의 추가비용으로 0.51%의 정확도 향상을 얻기 때문에 비용 대비 성능 향상은 준수하고, 변환된 코드가 새로운 허위 경보를 발생시키지 않음과 제거된 경보는 모두 허위 경보임을 확인하였다. 또한 범위를 벗어난 메모리 접근식 분석에 직접적인 영향을 주는 코드 변환 규칙을 사용하였기 때문에, 다른 정확도 (context-sensitivity, path-sensitivity 등) 향상만으로는 줄일 수 없는 허위 경보를 제거하였다. 따라서 이미 정교화된 분석기에서는 큰 비용 없이 안전한 정확도 향상 효과를 볼 수 있을 것이다. 하지만 정확도 향상의 양을 보면 그 효과가 미미하기 때문에 분석기의 성능이 시급한 경우에는 우선적으로 고려할 대상이 아니다.

참고문헌

[1] Patrick Cousot and Radhia Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Los Angeles, California, USA, January 1977. pp 238-252.

[2] Sparrow analyzer. http://ropas.snu.ac.kr/sparrow/

[3] Kihong Heo, Hakjoo Oh and Kwangkeun Yi, "Machine-learning-guided selectively unsound static analysis", in *Proceedings of the 39th International Conference on Software Engineering*, Buenos Aires, Argentina, June 2017. pp 519-529.

[4] Hakjoo Oh and Wonchan Lee and Kihong Heo and Hongseok Yang and Kwangkeun Yi, "Selective context-sensitivity guided by impact pre-analysis", in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Edinburgh, United Kingdom, June 2014. pp 475-484.

[5] Micha Sharir and Amir Pnueli, "Two approaches to interprocedural data flow analysis" in *Program Flow Analysis: Theory and Applications*, pages 189-234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[6] Olin Grigsby Shivers, "Control-flow analysis of higher-order languages -or- taming lambda" PhD thesis, CMU, 1991. Univ. of Illinois at U-C, 1993.

[7] Rijnard van Tonder and Claire Le Goues, "Tailoring programs for static analysis via program transformation," in *ICSE '20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, June 2020. pp 824-834.

Abstract

Code Transformation Rules to Improve Static Analysis Accuracy

Sehoon Kim
School of Computer Science & Engineering
College of Engineering
The Graduate School
Seoul National University

This paper introduces code transformation rules which improves static analysis accuracy without change of analysis technique, and checks effectiveness by experiments. To improve accuracy of static analyzer which detects buffer overrun of C programs, this paper proposes code transformation rules. Each of them describes a rule which can automatically transform by checking syntax without any costly analysis, while conserving semantics. With transformation rules which transform loop conditions and memory access expressions in loop body, the analysis accuracy was improved by 0.51% while analysis time was increased by 0.7%. Even a few false alarms were able to be removed with inexpensive costs, the absolute amount of removed false alrams was not significant.

Keywords: Program Analysis, Static Analysis, Code Transformation
Student Number: 2019-28745