Wer von Ihnen hat schon einmal in C programmiert?

Wer von Ihnen hat schon einmal in C programmiert?

Bitte beachten Sie, dass ein einzelner unachtsamer Programmierer dafür verantwortlich sein kann, dass ein Angreifer Root-Rechte auf einem System erlangt.

Pufferüberlauf - Beispiel

A ist ein 9 Byte String Puffer und B ein 2 Byte Integer Puffer

Var-Name	A										B	
Wert	<null string>										2000	
Hex-Wert	00	00	00	00	00	00	00	00	00	00	07	DE

Pufferüberlauf - Beispiel

A ist ein 9 Byte String Puffer und B ein 2 Byte Integer Puffer

Var-Name	A										B	
Wert	<null string>										2000	
Hex-Wert	00	00	00	00	00	00	00	00	00	00	07	DE

Jetzt schreiben wir das Wort „Zweitausend“ in A:

Var-Name	A										B	
Wert	'Z'	'w'	'e'	'i'	't'	'a'	'u'	's'	'e'	'n'	25600	
Hex-Wert	5A	77	65	69	74	61	75	73	65	6E	64	00

Pufferüberlauf - Beispiel

A ist ein 9 Byte String Puffer und B ein 2 Byte Integer Puffer

Var-Name	A										B	
Wert	<null string>										2000	
Hex-Wert	00	00	00	00	00	00	00	00	00	00	07	DE

Jetzt schreiben wir das Wort „Zweitausend“ in A:

Var-Name	A										B	
Wert	'Z'	'w'	'e'	'i'	't'	'a'	'u'	's'	'e'	'n'	25600	
Hex-Wert	5A	77	65	69	74	61	75	73	65	6E	64	00

Dabei wird auch der Wert von B überschrieben

→ Ein Pufferüberlauf

Pufferüberlauf ausnutzen

- Das Programm schreibt eine Nutzereingabe oder andere vom Nutzer beeinflussbare „Eingaben“ (z.B. Dateipfad) ungeschützt in einen Puffer

Pufferüberlauf ausnutzen

- Das Programm schreibt eine Nutzereingabe oder andere vom Nutzer beeinflussbare „Eingaben“ (z.B. Dateipfad) ungeschützt in einen Puffer
- Die Eingabe wird so gestaltet, dass die Rücksprungadresse mit einer anderen Adresse überschrieben wird, die gültig ist und zu schädlichem Code führt

Pufferüberlauf ausnutzen

- Das Programm schreibt eine Nutzereingabe oder andere vom Nutzer beeinflussbare „Eingaben“ (z.B. Dateipfad) ungeschützt in einen Puffer
- Die Eingabe wird so gestaltet, dass die Rücksprungadresse mit einer anderen Adresse überschrieben wird, die gültig ist und zu schädlichem Code führt
- Der schädliche Code, der von dem Nutzer eingeschleust wurde, wird ausgeführt
  - ➔ Das wollen wir verhindern

Pufferüberläufe erkennen mit Statischer Analyse

1) Einschränkungen für Variablen generieren
2) Makel identifizieren und entfernen
3) Lösung der Einschränkungen
    1) Direkte Lösung
    2) Hierarchische Lösung
    3) Vergleich der Methoden
4) Pufferüberläufe erkennen
5) Zusammenfassung

1) Einschränkungen für Variablen generieren
2) Makel identifizieren und entfernen
3) Lösung der Einschränkungen
    1) Direkte Lösung
    2) Hierarchische Lösung
    3) Vergleich der Methoden
4) Pufferüberläufe erkennen
5) Zusammenfassung

Einschränkungen generieren (1/3)

- Wir beschreiben die Variablen aus dem Quellcode mit Einschränkungen
  - Diese nennen wir auch Einschränkungsvariablen

Einschränkungen generieren (1/3)

- Wir beschreiben die Variablen aus dem Quellcode mit Einschränkungen
  - Diese nennen wir auch Einschränkungsvariablen
- Dabei unterscheiden wir Zeichenpuffer und Integer

Einschränkungen generieren (1/3)

- Wir beschreiben die Variablen aus dem Quellcode mit Einschränkungen
  - Diese nennen wir auch Einschränkungsvariablen
- Dabei unterscheiden wir Zeichenpuffer und Integer
- Ein Zeichenpuffer `buf` wird durch vier Einschränkungen beschrieben:
  - `buf!used!min` - Minimal genutzte Bytes
  - `buf!used!max` - Maximal genutzte Bytes
  - `buf!alloc!min` - Minimal zugewiesene Bytes
  - `buf!alloc!max` - Maximal zugewiesene Bytes

Einschränkungen generieren (2/3)

- Ein Integer `i` wird durch zwei Einschränkungen beschrieben:
  - `i!min` - Minimaler Wert von `i`
  - `i!max` - Maximaler Wert von `i`

Einschränkungen generieren (2/3)

- Ein Integer `i` wird durch zwei Einschränkungen beschrieben:
  - `i!min` - Minimaler Wert von `i`
  - `i!max` - Maximaler Wert von `i`
- Die Einschränkungsvariablen sind weder fluss- noch kontextsensitiv
  - Die Reihenfolge wird nicht beachtet
  - Mehrere Aufrufe werden nicht unterschieden

Einschränkungen generieren (3/3)

- Programmaufrufe im Quellcode werden durch lineare Einschränkungen umgesetzt

Einschränkungen generieren (3/3)

- Programmaufrufe im Quellcode werden durch lineare Einschränkungen umgesetzt
- 4 Arten von Anweisungen werden betrachtet:
  - Deklarationen
  - Zuweisungen
  - Funktionsaufrufe
  - Rückgabeanweisungen

Einschränkungen: Beispiel (1/3)

```
char buf[1024]
```

buf!alloc!min	≤	1024
buf!alloc!max	≥	1024

Einschränkungen: Beispiel (1/3)

```
char buf[1024]

buf[0] = 'a'
```

buf!alloc!min	≤	1024
buf!alloc!max	≥	1024
buf!used!min	≤	0
buf!used!max	≥	1

Einschränkungen: Beispiel (1/3)

```
char buf[1024]

buf[0] = 'a'

char hello[] =
   { „Hello World“ }
```

buf!alloc!min	≤	1024
buf!alloc!max	≥	1024
buf!used!min	≤	0
buf!used!max	≥	1
hello!alloc!min	≤	12
hello!alloc!max	≥	12
hello!used!min	≤	12
hello!used!min	≥	12

Einschränkungen: Beispiel (1/3)

```
char buf[1024]

buf[0] = 'a'

char hello[] =
   { „Hello World“ }

strcpy(buf, hello)
```

buf!alloc!min	≤	1024
buf!alloc!max	≥	1024
buf!used!min	≤	0
buf!used!max	≥	1
hello!alloc!min	≤	12
hello!alloc!max	≥	12
hello!used!min	≤	12
hello!used!min	≥	12
buf!used!max	≥	hello!used!max
buf!used!min	≤	hello!used!min

Einschränkungen: Beispiel (2/3)

```
int i = 0
```

i!min	≤	0
i!max	≥	0

Einschränkungen: Beispiel (2/3)

```
int i = 0

i=15
```

i!min	≤	0
i!max	≥	15

Einschränkungen: Beispiel (2/3)

```
int i = 0

i=15

i=-3
```

i!min	≤	-3
i!max	≥	15

Einschränkungen: Beispiel (3/3)

Betrachten wir eine for-Schleife:

```
for (int i=0; i<10; i++)
```

Einschränkungen: Beispiel (3/3)

Betrachten wir eine for-Schleife:

```
for (int i=0; i<10; i++)
```

i!min	≤	0
i!max	≥	0

Einschränkungen: Beispiel (3/3)

Betrachten wir eine for-Schleife:

```
for (int i=0; i<10; i++)
```

i!min	≤	0
i!max	≥	0

Einschränkungen: Beispiel (3/3)

Betrachten wir eine for-Schleife:

```
for (int i=0; i<10; i++)
```

i!min	≤	0
i!max	≥	0
i!max	≥	i!max + 1

Einschränkungen: Beispiel (3/3)

Betrachten wir eine for-Schleife:

```
for (int i=0; i<10; i++)
```

i!min	≤	0
i!max	≥	0
~~i!max~~	~~≥~~	~~i!max + 1~~
i'!max	≥	i!max + 1
i!max	≥	i'max

Diese Bedingung kann nicht von dem späteren Solver für die Einschränkungen interpretiert werden. Wir teilen die Anweisung in ein Paar von Anweisungen auf:

```
i' = i +1

i = i'
```

1) Einschränkungen für Variablen generieren
2) Makel identifizieren und entfernen
3) Lösung der Einschränkungen
    1) Direkte Lösung
    2) Hierarchische Lösung
    3) Vergleich der Methoden
4) Pufferüberläufe erkennen
5) Zusammenfassung

Makel identifizieren und entfernen

- Manche generierten Einschränkungen können unlösbar sein

Makel identifizieren und entfernen

- Manche generierten Einschränkungen können unlösbar sein
- Zum Lösen müssen alle Einschränkungen endliche Werte besitzen
  - Max Einschränkungen brauchen eine endliche untere Schranke
  - Min Einschränkungen brauchen eine endliche obere Schranke

Makel identifizieren und entfernen

- Manche generierten Einschränkungen können unlösbar sein
- Zum Lösen müssen alle Einschränkungen endliche Werte besitzen
  - Max Einschränkungen brauchen eine endliche untere Schranke
  - Min Einschränkungen brauchen eine endliche obere Schranke
- Wir betrachten zwei Aspekte:
  - Einschränkungen, die unendliche Werte annehmen
  - Einschränkungen, die nicht initialisiert sind

Unendliche Einschränkungen

- Alle Variablen, die vom Nutzer gesetzt werden, können beliebige Werte annehmen

Unendliche Einschränkungen

- Alle Variablen, die vom Nutzer gesetzt werden, können beliebige Werte annehmen
- Entsprechend werden die linearen Einschränkungen für diese Variablen auf unendlich gesetzt

Unendliche Einschränkungen

- Alle Variablen, die vom Nutzer gesetzt werden, können beliebige Werte annehmen
- Entsprechend werden die linearen Einschränkungen für diese Variablen auf unendlich gesetzt
- Da diese keine endliche obere/untere Schranke bilden, müssen sie entfernt werden

Nicht initialisierte Einschränkungen

- Variablen (z.B. `buf`), die nicht für alle Einschränkungsvariablen (z.B. `buf!used!min`) endliche Schranken haben, werden als nicht initialisiert bezeichnet

Nicht initialisierte Einschränkungen

- Variablen (z.B. `buf`), die nicht für alle Einschränkungsvariablen (z.B. `buf!used!min`) endliche Schranken haben, werden als nicht initialisiert bezeichnet
- Entsteht bei den Fällen:
  - Die Variable wurde im Quellcode nicht initialisiert
  - Es wurden Anweisungen, die den Wert bestimmen, nicht erfasst

Nicht initialisierte Einschränkungen

- Variablen (z.B. `buf`), die nicht für alle Einschränkungsvariablen (z.B. `buf!used!min`) endliche Schranken haben, werden als nicht initialisiert bezeichnet
- Entsteht bei den Fällen:
  - Die Variable wurde im Quellcode nicht initialisiert
  - Es wurden Anweisungen, die den Wert bestimmen, nicht erfasst
- Da nicht alle Einschränkungsvariablen endliche Schranken besitzen, müssen diese entfernt werden

Lösen der Einschränkungen

- Nutzen von linearer Programmierung

Lösen der Einschränkungen

- Nutzen von linearer Programmierung
- Zwei verschiedene Verfahren
  - Direkte Lösung
  - Hierarchische Lösung

Lösen der Einschränkungen

- Nutzen von linearer Programmierung
- Zwei verschiedene Verfahren
  - Direkte Lösung
  - Hierarchische Lösung
- Ziel:
  - Erreichen der bestmöglichen Ergebnisse für die Anzahl der genutzten Bytes der Puffer und Werte der Integer
  - Ganzzahlige Lösungen

Lineare Programmierung

- Lineares Programm ist Optimierungsproblem

Lineare Programmierung

- Lineares Programm ist Optimierungsproblem
- Die Optimierung, die wir für alle Einschränkungen `esv` suchen sind
  - Minimize: `esv!max`
  - Maximize: `esv!min`
  - Wobei `esv` auch für `buf!alloc` und `buf!used` steht

Lineare Programmierung

- Lineares Programm ist Optimierungsproblem
- Die Optimierung, die wir für alle Einschränkungen `esv` suchen sind
  - Minimize: `esv!max`
  - Maximize: `esv!min`
  - Wobei `esv` auch für `buf!alloc` und `buf!used` steht
- Je nach Verfahren nutzen wir als Grundlage dafür einen bestimmten Satz der linearen Einschränkungen

Lineare Programmierung

- Lineares Programm ist Optimierungsproblem
- Die Optimierung, die wir für alle Einschränkungen `esv` suchen sind
  - Minimize: `esv!max`
  - Maximize: `esv!min`
  - Wobei `esv` auch für `buf!alloc` und `buf!used` steht
- Je nach Verfahren nutzen wir als Grundlage dafür einen bestimmten Satz der linearen Einschränkungen
- Man kann alle einzelnen Programme zusammenfassen:
  - Minimize: $\sum_{esv} (esv!max - esv!min)$

1) Einschränkungen für Variablen generieren
2) Makel identifizieren und entfernen
3) Lösung der Einschränkungen
    1) Direkte Lösung
    2) Hierarchische Lösung
    3) Vergleich der Methoden
4) Pufferüberläufe erkennen
5) Zusammenfassung

Direkte Lösung (1/2)

- Nutzen des kompletten Satzes S der generierten Einschränkungen
  - Minimize: $\sum_{esv \in S} (esv!max - esv!min)$

    über alle Einschränkungen

Direkte Lösung (1/2)

- Nutzen des kompletten Satzes S der generierten Einschränkungen
  - Minimize: $\sum_{esv \in S} (esv!max - esv!min)$

    über alle Einschränkungen
- Es gibt nicht immer eine Lösung

Direkte Lösung (1/2)

- Nutzen des kompletten Satzes S der generierten Einschränkungen
  - Minimize: $\sum_{esv \in S} (esv!max - esv!min)$

    über alle Einschränkungen
- Es gibt nicht immer eine Lösung
- Sichere Lösung: Alle Einschränkungsvariablen auf entsprechend -∞ und ∞ setzen
  - Zu konservativ

Unlösbares Programm: Beispiel

- Die linearen Einschränkungen der for-Schleife:
  - $i'!max \geq i!max + 1$

    $i!max \geq i'!max$

Unlösbares Programm: Beispiel

- Die linearen Einschränkungen der for-Schleife:
  - $i'!max \geq i!max + 1$

    $i!max \geq i'!max$
- Versuchen wir, $i!max$ zu minimieren, so finden wir keine Lösung

Unlösbares Programm: Beispiel

- Die linearen Einschränkungen der for-Schleife:
  - $i'!max \geq i!max + 1$

    $i!max \geq i'!max$
- Versuchen wir, $i!max$ zu minimieren, so finden wir keine Lösung
- for-Schleifen sind weit verbreitet
  - Wir benötigen eine andere Lösung, als alles auf -∞ und ∞ zu setzen

Irreducibly Inconsistent Sets (1/2)

- Minimaler, nicht berechenbarer Satz

Irreducibly Inconsistent Sets (1/2)

- Minimaler, nicht berechenbarer Satz
- Entfernung einer linearen Einschränkung macht den Satz berechenbar

Irreducibly Inconsistent Sets (1/2)

- Minimaler, nicht berechenbarer Satz
- Entfernung einer linearen Einschränkung macht den Satz berechenbar
- Die beiden linearen Einschränkung der for-Schleife bilden *Irreducibly Inconsistent Set* (IIS)
  - Jede einzelne lineare Einschränkung ist berechenbar

Irreducibly Inconsistent Sets (2/2)

- In unserem ursprünglichen Satz S suchen wir alle IIS und entfernen sie
  - Wir speichern die entfernten IIS in S'

Irreducibly Inconsistent Sets (2/2)

- In unserem ursprünglichen Satz S suchen wir alle IIS und entfernen sie
  - Wir speichern die entfernten IIS in S'
- In S setzen wir alle Einschränkungen, die auch in S' enthalten sind, auf -∞ und ∞
  - Das Ergebnis speichern wir in S"

Irreducibly Inconsistent Sets (2/2)

- In unserem ursprünglichen Satz S suchen wir alle IIS und entfernen sie
  - Wir speichern die entfernten IIS in S'
- In S setzen wir alle Einschränkungen, die auch in S' enthalten sind, auf -∞ und ∞
  - Das Ergebnis speichern wir in S"
- In S" können wieder Makel enthalten sein
  - Makel entfernen

Direkte Lösung (2/2)

- Wir haben nun den Satz S"
  - Enthält keine nicht berechenbare Teile mehr
  - Enthält keine Makel mehr

Direkte Lösung (2/2)

- Wir haben nun den Satz S''
  - Enthält keine nicht berechenbare Teile mehr
  - Enthält keine Makel mehr
- Berechnen des Programms
  - Minimize: $\sum_{esv \in S''} (esv!max - esv!min)$

Direkte Lösung (2/2)

- Wir haben nun den Satz S''
  - Enthält keine nicht berechenbare Teile mehr
  - Enthält keine Makel mehr
- Berechnen des Programms
  - Minimize: $\sum_{esv \in S''} (esv!max - esv!min)$
- Wir haben eine Lösung!

Hierarchische Lösung (1/2)

- Unterteilen des gesamten Satzes von Einschränkungen in kleinere Teilsätze

Hierarchische Lösung (1/2)

- Unterteilen des gesamten Satzes von Einschränkungen in kleinere Teilsätze
- Zur Unterteilung konstruieren wir einen *directed acyclic graph* (DAG)
  - Ein Knoten ist ein Teilsatz
  - Berechnung in der topologischen Sortierung

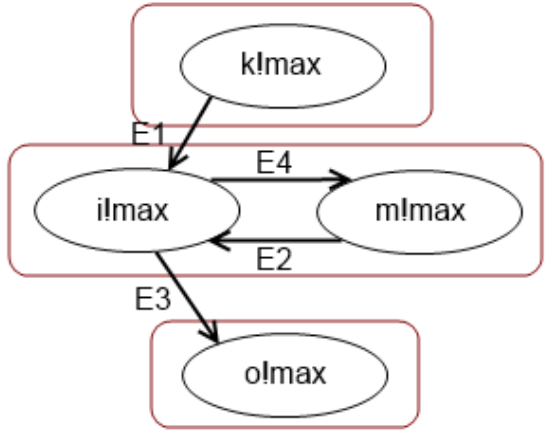
Hierarchische Lösung (1/2)

- Unterteilen des gesamten Satzes von Einschränkungen in kleinere Teilsätze
- Zur Unterteilung konstruieren wir einen *directed acyclic graph* (DAG)
  - Ein Knoten ist ein Teilsatz
  - Berechnung in der topologischen Sortierung
- Alle Einschränkungen in einem Knoten sind voneinander abhängig

DAG erstellen und lösen (1/2)

1. Erstellen eines Knotens für jede Einschränkungsvariable auf der linken Seite (Ovale Knoten)

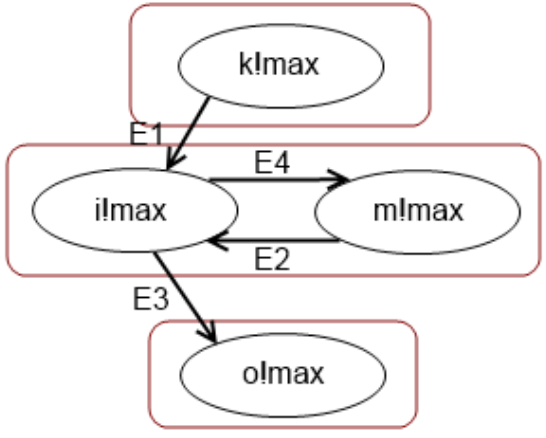
E1: i!max ≥ k!max
E2: i!max ≥ m!max + 15
E3: o!max ≥ i!max
E4: m!max ≥ i!max
E5: k!max ≥ 25



DAG erstellen und lösen (1/2)

1. Erstellen eines Knotens für jede Einschränkungsvariable auf der linken Seite (Ovale Knoten)
2. Verbinden von Knoten k mit Knoten l, wenn k auf der rechten und l auf der linken Seite steht (Pfeile)

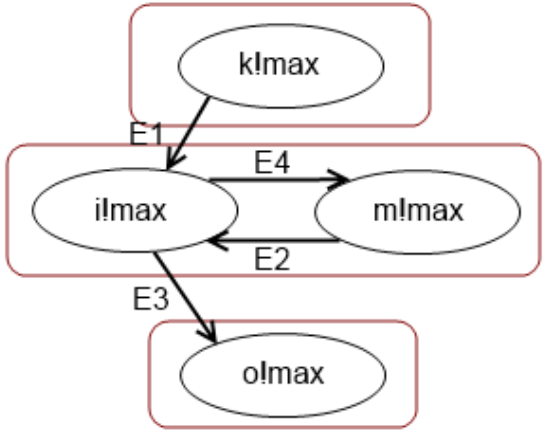
E1: i!max ≥ k!max
E2: i!max ≥ m!max + 15
E3: o!max ≥ i!max
E4: m!max ≥ i!max
E5: k!max ≥ 25



DAG erstellen und lösen (1/2)

1. Erstellen eines Knotens für jede Einschränkungsvariable auf der linken Seite (Ovale Knoten)
2. Verbinden von Knoten k mit Knoten l, wenn k auf der rechten und l auf der linken Seite steht (Pfeile)

E1: i!max ≥ k!max
E2: i!max ≥ m!max + 15
E3: o!max ≥ i!max
E4: m!max ≥ i!max
E5: k!max ≥ 25

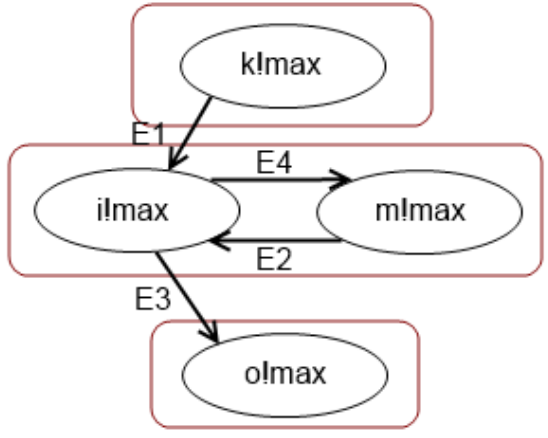


3. Finden von *Strongly Connected Components* (Rote Umrandungen)
   - Knoten in einem SCC sind voneinander abhängig

DAG erstellen und lösen (2/2)

- Der Graph der SCCs bildet den DAG

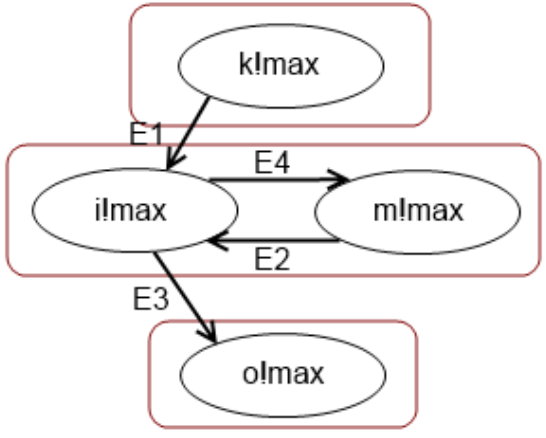
E1: i!max ≥ k!max
E2: i!max ≥ m!max + 15
E3: o!max ≥ i!max
E4: m!max ≥ i!max
E5: k!max ≥ 25



DAG erstellen und lösen (2/2)

- Der Graph der SCCs bildet den DAG
- Topologische Ordnung des DAG definiert uns eine Hierarchie

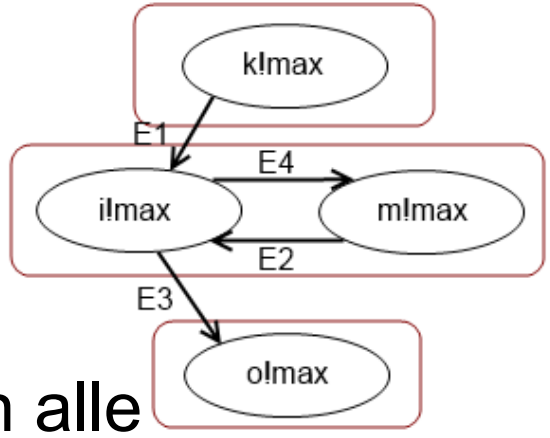
E1: i!max ≥ k!max
E2: i!max ≥ m!max + 15
E3: o!max ≥ i!max
E4: m!max ≥ i!max
E5: k!max ≥ 25



DAG erstellen und lösen (2/2)

- Der Graph der SCCs bildet den DAG
- Topologische Ordnung des DAG definiert uns eine Hierarchie
- Jede SCC wird mit der direkten Lösung gelöst
  - Ist eine SCC nicht berechenbar, so können alle Einschränkungsvariablen auf -∞ und ∞ gesetzt werden
  - Es müssen keine IIS identifiziert werden

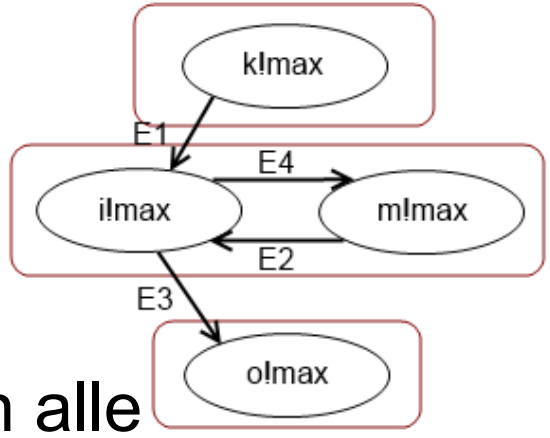
E1: i!max ≥ k!max
E2: i!max ≥ m!max + 15
E3: o!max ≥ i!max
E4: m!max ≥ i!max
E5: k!max ≥ 25



DAG erstellen und lösen (2/2)

- Der Graph der SCCs bildet den DAG
- Topologische Ordnung des DAG definiert uns eine Hierarchie
- Jede SCC wird mit der direkten Lösung gelöst
  - Ist eine SCC nicht berechenbar, so können alle Einschränkungsvariablen auf -∞ und ∞ gesetzt werden
  - Es müssen keine IIS identifiziert werden
- So können wir nach der gegebenen Hierarchie alle SCCs lösen
  - Wir haben eine Lösung

E1: i!max ≥ k!max
E2: i!max ≥ m!max + 15
E3: o!max ≥ i!max
E4: m!max ≥ i!max
E5: k!max ≥ 25



Hierarchische Lösung (2/2)

- Durch Substitution der Werte kann ein Solver überflüssig werden

Hierarchische Lösung (2/2)

- Durch Substitution der Werte kann ein Solver überflüssig werden
- Den Satz jeden Knotens im DAG einzeln zu lösen bietet die Möglichkeit, verschiedene Solver zu nutzen

Hierarchische Lösung (2/2)

- Durch Substitution der Werte kann ein Solver überflüssig werden
- Den Satz jeden Knotens im DAG einzeln zu lösen bietet die Möglichkeit, verschiedene Solver zu nutzen
- Bei breiten DAGs können die SCCs einer Ebene parallel gelöst werden
  - DAG der Tiefe T in T Schritten lösen

Vergleich der Methoden

- Direkte Lösung ist nur ein Approximationsverfahren
  - Es können unnötig viele lineare Einschränkungen entfernt werden
  - Entsprechend viele Einschränkungsvariablen werden auf -∞ und ∞ gesetzt

Vergleich der Methoden

- Direkte Lösung ist nur ein Approximationsverfahren
  - Es können unnötig viele lineare Einschränkungen entfernt werden
  - Entsprechend viele Einschränkungsvariablen werden auf -∞ und ∞ gesetzt
- Hierarchischer Ansatz ist präzise
  - Die kleinstmögliche Anzahl von Einschränkungsvariablen wird auf -∞ und ∞ gesetzt

Vergleich der Methoden

- Direkte Lösung ist nur ein Approximationsverfahren
  - Es können unnötig viele lineare Einschränkungen entfernt werden
  - Entsprechend viele Einschränkungsvariablen werden auf -∞ und ∞ gesetzt
- Hierarchischer Ansatz ist präzise
  - Die kleinstmögliche Anzahl von Einschränkungsvariablen wird auf -∞ und ∞ gesetzt
- Gegenüberstehen von der schnelleren, aber ungenaueren direkten Lösung und der langsamen, aber genaueren hierarchische Lösung

Pufferüberläufe erkennen

- Ergebnistabelle:

buf1!used!max	1024
buf1!used!min	0
buf1!alloc!max	1024
buf1!alloc!min	1024
buf2!used!max	2048
buf2!used!min	1024
buf2!alloc!max	1024
buf2!alloc!min	0
buf3!used!max	2048
buf3!used!min	1024
buf3!alloc!max	2048
buf3!alloc!min	1024

Pufferüberläufe erkennen

- Ergebnistabelle:

- buf1 ist sicher
  - used!max ≤ alloc!min

buf1!used!max	1024
buf1!used!min	0
buf1!alloc!max	1024
buf1!alloc!min	1024
buf2!used!max	2048
buf2!used!min	1024
buf2!alloc!max	1024
buf2!alloc!min	0
buf3!used!max	2048
buf3!used!min	1024
buf3!alloc!max	2048
buf3!alloc!min	1024

Pufferüberläufe erkennen

- Ergebnistabelle:

buf1!used!max	1024
buf1!used!min	0
buf1!alloc!max	1024
buf1!alloc!min	1024
buf2!used!max	2048
buf2!used!min	1024
buf2!alloc!max	1024
buf2!alloc!min	0
buf3!used!max	2048
buf3!used!min	1024
buf3!alloc!max	2048
buf3!alloc!min	1024

- buf1 ist sicher
  - used!max ≤ alloc!min
- buf2 ist ein Pufferüberlauf
  - used!max ≥ alloc!max

Pufferüberläufe erkennen

- Ergebnistabelle:

buf1!used!max	1024
buf1!used!min	0
buf1!alloc!max	1024
buf1!alloc!min	1024
buf2!used!max	2048
buf2!used!min	1024
buf2!alloc!max	1024
buf2!alloc!min	0
buf3!used!max	2048
buf3!used!min	1024
buf3!alloc!max	2048
buf3!alloc!min	1024

- buf1 ist sicher
  - used!max ≤ alloc!min
- buf2 ist ein Pufferüberlauf
  - used!max ≥ alloc!max
- buf3 ist ein möglicher Pufferüberlauf
  - used!max ≥ alloc!min und used!max ≤ alloc!max

Zusammenfassung

- Erzeugen von Einschränkungsvariablen

Zusammenfassung

- Erzeugen von Einschränkungsvariablen
- Generieren der linearen Einschränkungen

Zusammenfassung

- Erzeugen von Einschränkungsvariablen
- Generieren der linearen Einschränkungen
- Makel identifizieren und entfernen

Zusammenfassung

- Erzeugen von Einschränkungsvariablen
- Generieren der linearen Einschränkungen
- Makel identifizieren und entfernen
- Lösen
  - Direkte Lösung
    - IIS finden und entfernen, erneut Makel entfernen und lösen

Zusammenfassung

- Erzeugen von Einschränkungsvariablen
- Generieren der linearen Einschränkungen
- Makel identifizieren und entfernen
- Lösen
  - Direkte Lösung
    - IIS finden und entfernen, erneut Makel entfernen und lösen
  - Hierarchisch Lösung
    - Knoten für alle Einschränkungsvariablen erzeugen, Verbindungen erstellen, SCCs finden, DAG der SCCs lösen

Zusammenfassung

- Erzeugen von Einschränkungsvariablen
- Generieren der linearen Einschränkungen
- Makel identifizieren und entfernen
- Lösen
  - Direkte Lösung
    - IIS finden und entfernen, erneut Makel entfernen und lösen
  - Hierarchisch Lösung
    - Knoten für alle Einschränkungsvariablen erzeugen, Verbindungen erstellen, SCCs finden, DAG der SCCs lösen
- Pufferüberläufe erkennen