# *Verificarea acceselor la memorie în programe C folosind constrângeri simbolice și slicing*

*lucrare de diplomă*

Edvin Török

`coordonator: conf. dr. ing.` Marius Minea

Facultatea de Automatică și Calculatoare
Departamentul Calculatoare

23 iunie 2009

# Erori de memorie în programe C

În programe C printre cele mai des întâlnite erori sunt erorile de memorie.

- dereferențieri de pointer `NULL` :

  **struct** foo ∗a = malloc (...) ; a−>x = 42;

- memory leak
- depășire în aritmetica de pointeri când nu e respectată condiția

  $$pointer + unsigned > pointer$$

- depășiri de limite în accesarea unui pointer :

  **char** a[n]; a[n]=0;

# Erori de memorie în programe C

În programe C printre cele mai des întâlnite erori sunt erorile de memorie.

- dereferențieri de pointer `NULL` :

```
struct foo *a = malloc (...) ; a->x = 42;
```

- memory leak
- depășire în aritmetica de pointeri când nu e respectată condiția

$$pointer + unsigned > pointer$$

- depășiri de limite în accesarea unui pointer :

```
char a[n]; a[n]=0;
```

# Instrumente existente

Există multe instrumente pentru detectarea depășirilor de limite, dar:

- Unele sunt comerciale (Coverity, Polyspace, . . . )
- Altele realizează o verificare foarte bună, însă nu corespund cerinței de timp (KLEE, . . . )
- Altele nu detectează erori dacă dereferențierile se află în alte funcții (splint, . . . )

# Scop

Scopul este de-a realiza un instrument care:

- să caute erori de depășiri de limite, chiar dacă alocările și dereferențierile se întâmplă în funcții diferite
- să fie ușor integrabil în procesul de compilare
- să ruleze într-un timp comparabil cu cel de compilare
- să fie preferabil open-source

# De ce depășiri de limite?

- Sunt greșeli grave în programe C
- Cauzează probleme de securitate (vulnerabilități)
- Deseori e greu de detectat dacă a avut loc sau nu această eroare
- Spre deosebire de o dereferențiere de pointer NULL, programul nu e terminat de SO

# Alegere infrastructură

Folosim reprezentarea intermediară a unui compilator existent: Low Level Virtual Machine

- LLVM este o infrastructură pentru compilatoare
- un compilator folosit în producție: LLVM-GCC
- folosit de Adobe (Alchemy, Hydra), Apple Inc. (OpenCL, OpenGL engine, Xcode), Cray Inc. (x86 compiler), etc.
- folosit de multe alte instrumente similare: KLEE, Calysto, ...
- analiză flux de date și control
- formă Static Single Assignment
- transformări și optimizările folosite într-un compilator modern
- lucrează la un nivel mai scăzut decât C (bitcode)
- conversii de tip explicite (trunchiere, extensie prin semn/zero, etc.)
- design modular
- open source, drepturi de scris în SVN

# Principiul abordării

```
int foo(unsigned size, char *p)
{
    if (!size) return -1;
    unsigned s=0;
    for (int i=0;i<size;i++)
        s += foobar(i);
    process(p, size, s);
    p[size - 1]++;
    unsigned pos = foobar(0);
    if (pos < size)
      p[pos] = 0;
    process(p, size, 0);
    return 0;
}
```

Vrem să analizăm validitatea *dereferențierilor* de pointeri:

- Verificăm doar depășiri de limite
- Urmărim evoluția valorilor în cicluri, aritmetica de pointeri
- Păstrăm constrângeri pentru accesele la memorie
- Rezolv constrângerile, dacă nu sunt îndeplinite raportez o eroare

# Principiul abordării

```
int foo(unsigned size, char *p)
{
    if (!size) return -1;
    unsigned s=0;
    for (int i=0;i<size;i++)
        s += foobar(i);
    process(p, size, s);
    p[size - 1]++;
    unsigned pos = foobar(0);
    if (pos < size)
      p[pos] = 0;
    process(p, size, 0);
    return 0;
}
```

Vrem să analizăm validitatea *dereferențierilor* de pointeri:

- Verificăm doar depășiri de limite
- Urmărim evoluția valorilor în cicluri, aritmetica de pointeri
- Păstrăm constrângeri pentru accesele la memorie
- Rezolv constrângerile, dacă nu sunt îndeplinite raportez o eroare

# Principiul abordării

```
int foo(unsigned size, char *p)
{
    if (!size) return -1;
    unsigned s=0;
    for (int i=0;i<size;i++)
        s += foobar(i);
    process(p, size, s);
    p[size - 1]++;
    unsigned pos = foobar(0);
    if (pos < size)
      p[pos] = 0;
    process(p, size, 0);
    return 0;
}
```

Vrem să analizăm validitatea *dereferențierilor* de pointeri:

- Verificăm doar depășiri de limite
- Urmărim evoluția valorilor în cicluri, aritmetica de pointeri
- Păstrăm constrângeri pentru accesele la memorie
- Rezolv constrângerile, dacă nu sunt îndeplinite raportez o eroare

# Principiul abordării

```
int foo(unsigned size, char *p)
{
    if (!size) return -1;
    unsigned s=0;
    for (int i=0;i<size;i++)
        s += foobar(i);
    process(p, size, s);
    p[size - 1]++;
    unsigned pos = foobar(0);
    if (pos < size)
      p[pos] = 0;
    process(p, size, 0);
    return 0;
}
```

Vrem să analizăm validitatea *dereferențierilor* de pointeri:

- Verificăm doar depășiri de limite
- Urmărim evoluția valorilor în cicluri, aritmetica de pointeri
- Păstrăm constrângeri pentru accesele la memorie
- Rezolv constrângerile, dacă nu sunt îndeplinite raportez o eroare

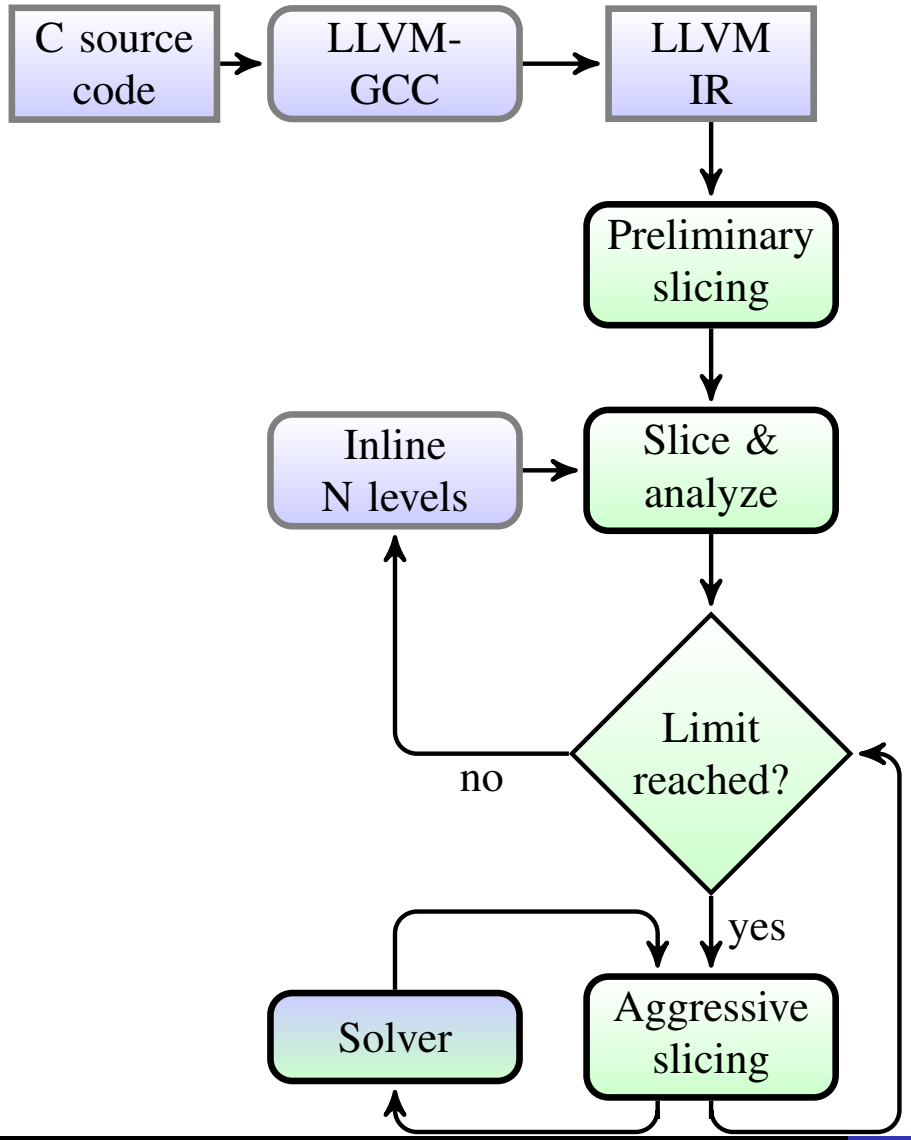
# Principiul abordării

```
int foo(unsigned size, char *p)
{
    if (!size) return -1;
    unsigned s=0;
    for (int i=0;i<size;i++)
        s += foobar(i);
    process(p, size, s);
    p[size - 1]++;
    unsigned pos = foobar(0);
    if (pos < size)
      p[pos] = 0;
    process(p, size, 0);
    return 0;
}
```

Vrem să analizăm validitatea *dereferențierilor* de pointeri:

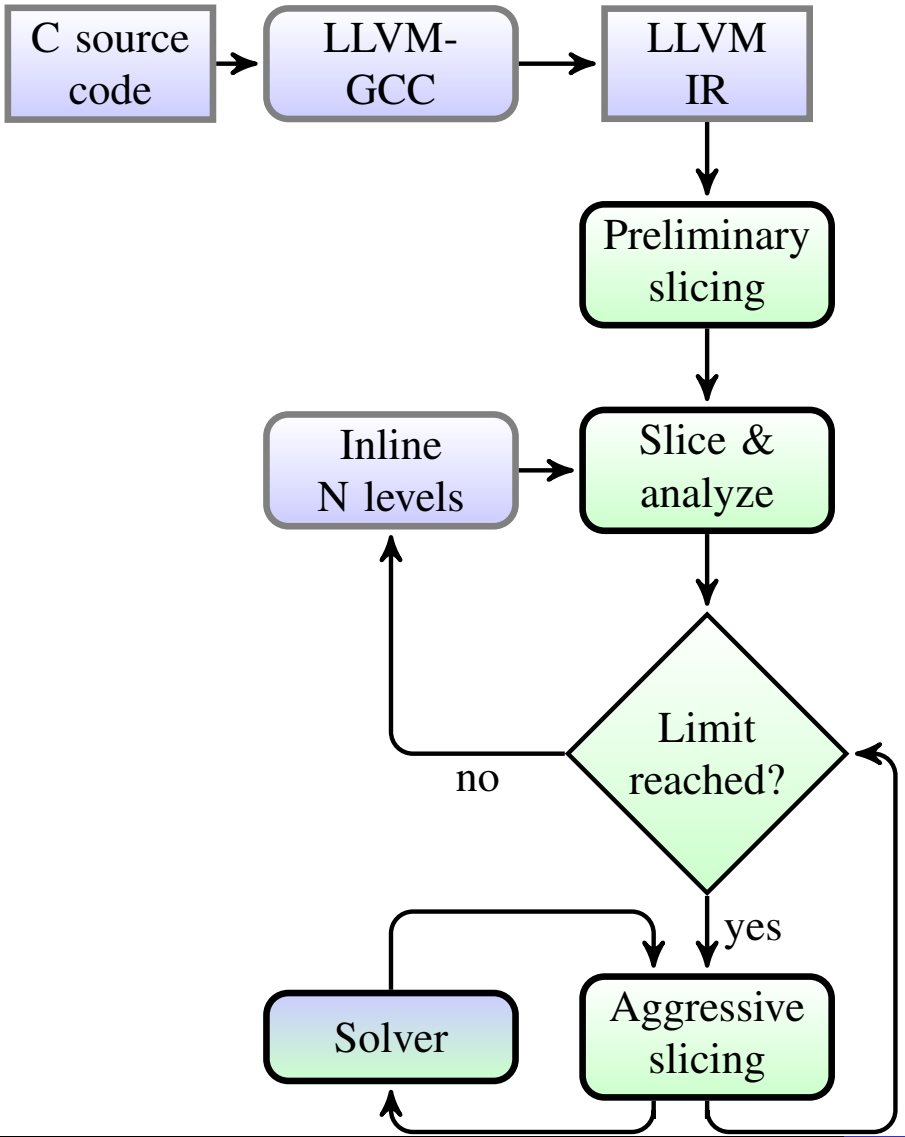
- Verificăm doar depășiri de limite
- Urmărim evoluția valorilor în cicluri, aritmetica de pointeri
- Păstrăm constrângeri pentru accesele la memorie
- Rezolv constrângerile, dacă nu sunt îndeplinite raportez o eroare

# Structura aplicației



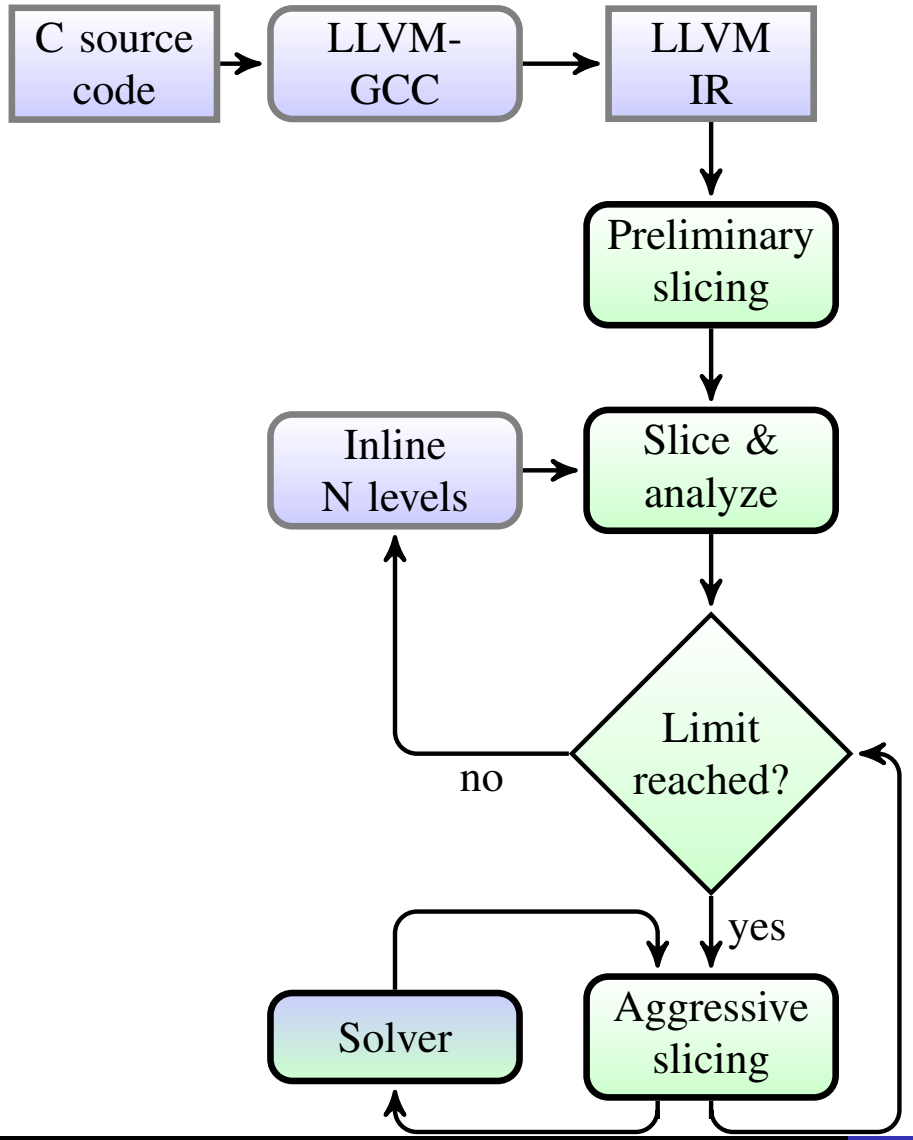

- Trebuie să cunoaștem dimensiunea zonelor alocate, deci pornim de la alocări.
- Marcăm toate variabilele care pot conține pointeri și dimensiunile acestora.
- Obținem o expresie *Scalar Evolution* pentru fiecare dereferențiere de pointer.
- Simplificăm, păstrăm doar valorile care intervin în expr. Scalar Evolution obținute anterior.

# Structura aplicației



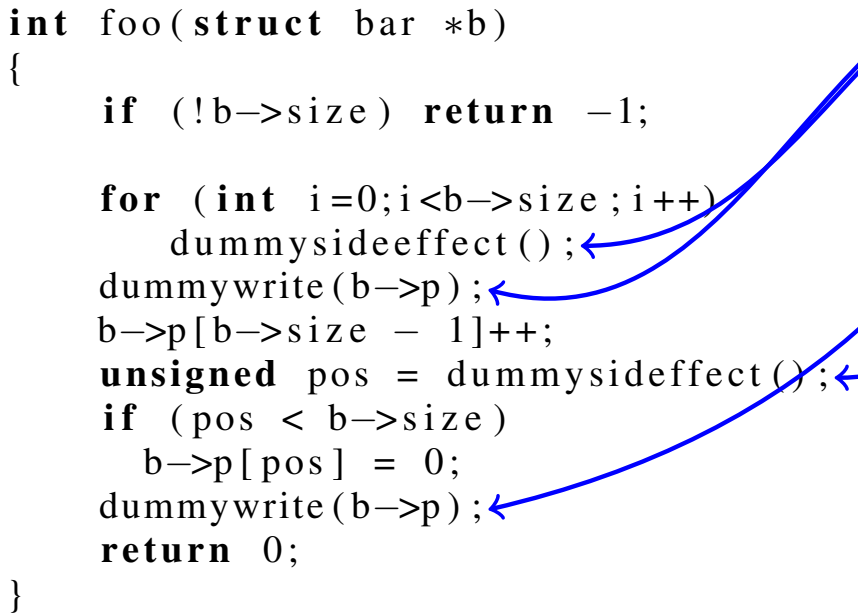

- Trebuie să cunoaștem dimensiunea zonelor alocate, deci pornim de la alocări.
- Marcăm toate variabilele care pot conține pointeri și dimensiunile acestora.
- Obținem o expresie *Scalar Evolution* pentru fiecare dereferențiere de pointer.
- Simplificăm, păstrăm doar valorile care intervin în expr. Scalar Evolution obținute anterior.

# Structura aplicației




- Trebuie să cunoaștem dimensiunea zonelor alocate, deci pornim de la alocări.
- Marcăm toate variabilele care pot conține pointeri și dimensiunile acestora.
- Obținem o expresie *Scalar Evolution* pentru fiecare dereferențiere de pointer.
- Simplificăm, păstrăm doar valorile care intervin în expr. Scalar Evolution obținute anterior.

# Slicing
## Pasul 1



```
int foo(struct bar *b)
{
    if (!b->size) return -1;

    for (int i=0;i<b->size;i++)
        dummysideeffect();
    dummywrite(b->p);
    b->p[b->size - 1]++;
    unsigned pos = dummysideeffect();
    if (pos < b->size)
      b->p[pos] = 0;
    dummywrite(b->p);
    return 0;
}
```

- Valorile și apelurile de proceduri *neinteresante* l-am înlocuit cu apeluri către funcții dummy
- Simplificăm rezultatul folosind:
  - Scalar Replacement of Aggregates
  - Global Value Numbering
  - Loop Invariant Code Motion
  - Dead Code Elimination
  - Loop Deletion

# Slicing
## Pasul 1

```
int foo(struct bar *b)
{
    if (!b->size) return -1;

    for (int i=0;i<b->size;i++)
        dummysideeffect();
    dummywrite(b->p);
    b->p[b->size - 1]++;
    unsigned pos = dummysideeffect();
    if (pos < b->size)
      b->p[pos] = 0;
    dummywrite(b->p);
    return 0;
}
```

- Valorile și apelurile de proceduri *neinteresante* l-am înlocuit cu apeluri către funcții dummy
- Simplificăm rezultatul folosind:
  - Scalar Replacement of Aggregates
  - Global Value Numbering
  - Loop Invariant Code Motion
  - Dead Code Elimination
  - Loop Deletion

# Constrângeri. Verificare

## Pasul 2

- `char *base = malloc(n); p = base + offset; memset(p, 0, length);`
- accesează `p[0] ... p[length-1]`
- accesează `base[offset] ... base[offset+length-1]`
- access valid: $\left[\texttt{offset}, \texttt{offset} + \texttt{length}\right) \subseteq \left[0, \texttt{n}\right)$
- constrângeri:

$$\begin{aligned} \texttt{offset} &\in \left[0, \texttt{n}\right) \\ \texttt{offset} + \texttt{length} &\in \left[0, \texttt{n}\right] \\ \texttt{length} &\in \left[0, \texttt{n}\right] \end{aligned}$$

# Constrângeri. Verificare

## Pasul 2

- `char *base = malloc(n); p = base + offset; memset(p, 0, length);`
- accesează `p[0] ... p[length-1]`
- accesează `base[offset] ... base[offset + length - 1]`
- access valid: $\left[\texttt{offset}, \texttt{offset} + \texttt{length}\right) \subseteq \left[0, \texttt{n}\right)$
- constrângeri:

$$\texttt{offset} \in \left[0, \texttt{n}\right)$$
$$\texttt{offset} + \texttt{length} \in \left[0, \texttt{n}\right]$$
$$\texttt{length} \in \left[0, \texttt{n}\right]$$

# Constrângeri. Verificare

## Pasul 2

- `char * base = malloc( n );`
  `p = base + offset ;memset( p , 0, length );`
- accesează `p[0] ... p[ length -1]`
- înlocuim `p = base + offset`
- accesează `base[ offset ] ... base[ offset + length -1]`
- access valid: $\left[\texttt{offset}, \texttt{offset} + \texttt{length}\right) \subseteq \left[0, \texttt{n}\right)$
- constrângeri:

$$\texttt{offset} \in \left[0, \texttt{n}\right)$$
$$\texttt{offset} + \texttt{length} \in \left[0, \texttt{n}\right]$$
$$\texttt{length} \in \left[0, \texttt{n}\right]$$

# Constrângeri. Verificare

## Pasul 2

- `char * base = malloc( n );`
  `p = base + offset ;memset( p , 0, length );`
- accesează `p[0] ... p[length-1]`
- accesează `base[offset] ... base[offset + length -1]`
- access valid: $\left[\texttt{offset}, \texttt{offset} + \texttt{length}\right) \subseteq \left[0, \texttt{n}\right)$
- constrângeri:

$$\texttt{offset} \in \left[0, \texttt{n}\right)$$
$$\texttt{offset} + \texttt{length} \in \left[0, \texttt{n}\right]$$
$$\texttt{length} \in \left[0, \texttt{n}\right]$$

# Constrângeri. Verificare

## Pasul 2

- `char * base = malloc( n );`
  `p = base + offset ; memset( p , 0, length );`
- accesează `p [0] ... p [ length -1]`
- accesează `base [ offset ] ... base [ offset + length -1]`
- access valid: $\left[\texttt{offset}, \texttt{offset} + \texttt{length}\right) \subseteq \left[0, \texttt{n}\right)$
- constrângeri:

$$\begin{aligned} \texttt{offset} &\in \left[0, \texttt{n}\right) \\ \texttt{offset} + \texttt{length} &\in \left[0, \texttt{n}\right] \\ \texttt{length} &\in \left[0, \texttt{n}\right] \end{aligned}$$

# Scalar Evolution
## Principiul abordării

### Exemplu

```
j=4;
for(i=1;i < 100;i += 3)
   j = i + 8 +j
```

### Valori j în funcție de iterație

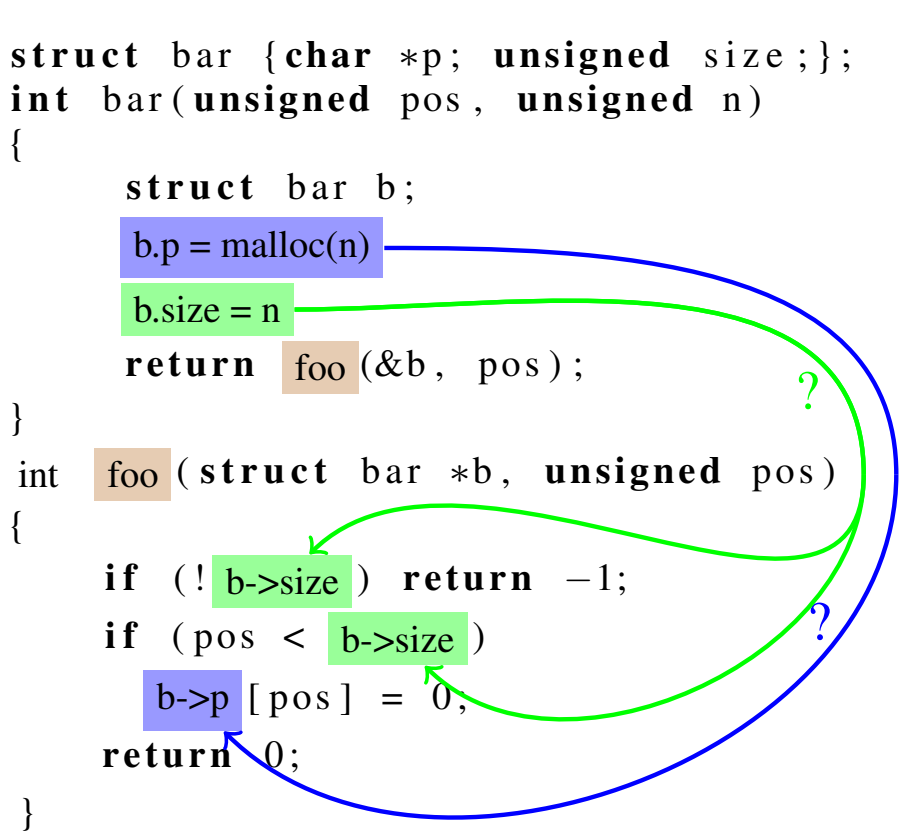
$$i = 1,4,7,10,13,\ldots,97$$
$$j = 4,13,25,40,58,\ldots,1885$$
$$j(n) = \sum_{i=0}^{p} \Delta_0^i \binom{n}{i}$$

Tabela: diferențiere pentru $j$

| $k$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $\Delta_k^0$ | 4 | 13 | 25 | 40 | 58 |
| $\Delta_k^1$ | 9 | 12 | 15 | 18 | |
| $\Delta_k^2$ | 3 | 3 | 3 | | |
| $\Delta_k^3$ | 0 | 0 | | | |

$$j(n) = 4\binom{n}{0} + 9\binom{n}{1} + 3\binom{n}{2}$$
$$= \frac{3}{2}n^2 + \frac{15}{2}n + 4$$

# Propagarea interprocedurală a constrângerilor




- Nu cunoaștem în ce măsură `b->size` e legat de dimensiunea lui `b->p`
- În mod obișnuit se calculează un sumar de funcție, și se propagă aceste informații interprocedural
- Am ales să folosesc inlining în loc de sumare de funcție:
  - \+ Putem folosi mai multe analize LLVM
  - \- Dar crește foarte mult dimensiunea codului
  - Putem șterge accesele deja verificate (dacă nu sunt folosite de celelalte accese)
  - Folosind *slicing* putem reduce codul la dimensiuni acceptabile

# Propagarea interprocedurală a constrângerilor

```
struct bar {char *p; unsigned size;};
int bar(unsigned pos, unsigned n)
{
        struct bar b;
        b.p = malloc(n)
        b.size = n
        return foo(&b, pos);
}
int foo(struct bar *b, unsigned pos)
{
      if (! b->size ) return -1;
      if (pos < b->size )
         b->p [pos] = 0;
      return 0;
}
```

- Nu cunoaștem în ce măsură `b->size` e legat de dimensiunea lui `b->p`
- În mod obișnuit se calculează un sumar de funcție, și se propagă aceste informații interprocedural
- Am ales să folosesc inlining în loc de sumare de funcție:
  - \+ Putem folosi mai multe analize LLVM
  - \- Dar crește foarte mult dimensiunea codului
  - Putem șterge accesele deja verificate (dacă nu sunt folosite de celelalte accese)
  - Folosind *slicing* putem reduce codul la dimensiuni acceptabile

# Propagarea interproceduralǎ a constrângerilor

```
struct bar {char *p; unsigned size;};
int bar(unsigned pos, unsigned n)
{
        struct bar b;
        b.p = malloc(n)
        b.size = n
        return foo(&b, pos);
}
int foo(struct bar *b, unsigned pos)
{
      if (!b->size) return -1;
      if (pos < b->size)
         b->p[pos] = 0;
      return 0;
}
```

- Nu cunoaștem în ce măsură `b->size` e legat de dimensiunea lui `b->p`
- În mod obișnuit se calculează un sumar de funcție, și se propagă aceste informații interprocedural
- Am ales să folosesc inlining în loc de sumare de funcție:
  - \+ Putem folosi mai multe analize LLVM
  - \- Dar crește foarte mult dimensiunea codului
  - Putem șterge accesele deja verificate (dacă nu sunt folosite de celelalte accese)
  - Folosind *slicing* putem reduce codul la dimensiuni acceptabile

# Propagarea interproceduralǎ a constrângerilor

```
struct bar {char *p; unsigned size;};
int bar(unsigned pos, unsigned n)
{
        struct bar b;
        b.p = malloc(n)
        b.size = n
        return foo(&b, pos);
}
int foo(struct bar *b, unsigned pos)
{
      if (! b->size ) return -1;
      if (pos < b->size )
         b->p [pos] = 0;
      return 0;
}
```

- Nu cunoaștem în ce măsură `b->size` e legat de dimensiunea lui `b->p`
- În mod obișnuit se calculează un sumar de funcție, și se propagă aceste informații interprocedural
- Am ales să folosesc inlining în loc de sumare de funcție:
  - \+ Putem folosi mai multe analize LLVM
  - \- Dar crește foarte mult dimensiunea codului
  - Putem șterge accesele deja verificate (dacă nu sunt folosite de celelalte accese)
  - Folosind *slicing* putem reduce codul la dimensiuni acceptabile

# Inlining
## Pasul 3

```
struct bar {char *p; unsigned size;};
int bar(unsigned pos, unsigned n)
{
        struct bar b;
        b.p = malloc(n)
        b.size = n
        return foo(&b, pos);
}
int foo(struct bar *b, unsigned pos)
{
     if (!b->size) return -1;
     if (pos < b->size)
        b->p[pos] = 0;
     return 0;
}
```

Folosire inlining:

- `foo()` va fi expandat inline în locul de apel (`bar()`)
- Vom vedea că `b->size` e de fapt dimensiunea lui `b->p`
- Vom vedea că există un predicat care garantează validitatea accesului

# Inlining
## Pasul 3

```
struct bar {char *p; unsigned size;};
int bar(unsigned pos, unsigned n)
{
      struct bar b;
      b.p = malloc(n)
      b.size = n
      return foo(&b, pos);
}
int foo(struct bar *b, unsigned pos)
{
     if (! b->size ) return -1;
     if (pos < b->size )
       b->p [pos] = 0;
     return 0;
}
```

Folosire inlining:

- `foo()` va fi expandat inline în locul de apel (`bar()`)
- Vom vedea că b->size e de fapt dimensiunea lui b->p
- Vom vedea că există un predicat care garantează validitatea accesului

# Solver
## Pasul 4

- Simplificăm codul
- Ne uităm la predicate
- Am implementat un solver simplu bazat pe expresii Scalar Evolution
- Repetăm până când atingem limita de inlining, sau verificăm toate accesele

# SMT Solver

- În prima versiune am generat constrângeri pentru un solver Satisfiability Modulo Theories (Yices)
- Am prezentat detalii la Alpine Verification Meeting 2008, Semmering, Austria
- Era numai intraprocedural, cu multe alerte false

# PPL

- Majoritatea constrângerile pot fi reprezentate ca și combinații lineare: $c_0 + c_1 * a_1 + c_2 * a_2 + \ldots + c_n * a_n < 0$
- Folosind *Parma Polyhedra Library* putem reprezenta acest sistem de constrângeri ca și un poliedru convex
- Aritmetica modulo 2 a întregilor se poate reprezenta în domeniul $\mathbb{Z}^n$
- Problema găsirii unei soluții pentru o constrângere se reduce la intersecția dintre poliedre, $\mathbb{Z}^n$
- Însă algoritmul este în cel mai rău caz exponențial
- S-ar putea folosi domenii mai simple (diferențe finite, octagon), dar se pierde din precizie: nu toate constrângerile sunt reprezentabile și e nevoie de aproximări
- Era prea încet pentru analiză interprocedurală

# Soluția actuală

- Funcționează pentru constante, și 40% de cazuri întâlnite în practică
- E rapid, 60000 constrângeri în < 15s
- Ideea de bază
  - `umax(offset, length) == ?`
  - Dacă poate fi egal cu `offset` → am găsit o eroare
  - Dacă e mereu egal cu `length` → accesul la memorie e mereu valid
  - Similar pentru `offset+length`, și `length`
- În general problema se reduce la `UMAX(Good, Bad, Strict)`
  - `UMAX == Good && Good != Bad` → OK
  - `UMAX == Bad && (Strict || Good != Bad))` → EROARE
  - Altfel nu cunoaștem răspunsul (am putea folosi eventual un solver mai precis, și lent)
- Simplificare expresii `umax`, implementat deja în LLVM
- Scădem cele 2 valori, dacă rezultatul e constantă avem un răspuns

# Soluția actuală

- Funcționează pentru constante, și 40% de cazuri întâlnite în practică
- E rapid, 60000 constrângeri în < 15s
- Ideea de bază
  - `umax(offset, length) == ?`
  - Dacă poate fi egal cu `offset` → am găsit o eroare
  - Dacă e mereu egal cu `length` → accesul la memorie e mereu valid
  - Similar pentru `offset+length`, și `length`
- În general problema se reduce la `UMAX(Good, Bad, Strict)`
  - `UMAX == Good && Good != Bad` → OK
  - `UMAX == Bad && (Strict || Good != Bad))` → EROARE
  - Altfel nu cunoaștem răspunsul (am putea folosi eventual un solver mai precis, și lent)
- Simplificare expresii `umax`, implementat deja în LLVM
- Scădem cele 2 valori, dacă rezultatul e constantă avem un răspuns

# Soluția actuală

## Continuare

- Interval de valori posibile pentru variabile mai precis decât furnizat de LLVM
- Determinat din predicatele cu valoare de adevăr cunoscute
  - Valorile condiției din `if, while` pe calea de execuție curentă
  - Dacă putem ajunge într-un punct din program doar pe o anumită cale predicatele de pe acea cale sunt adevărate
  - Predicatele ne dau un interval nou de valori care sunt intersectate
  - De exemplu: `if (!size) return` → `size > 0`
  - Alt exemplu: `if (a < n)` → `n > 0`
- Dacă găsim un predicat care compară cele 2 părți din `umax` avem un răspuns
- Cunoscând intervalul de valori se înlocuiesc extremele în expresie, dacă obținem o constantă avem un răspuns

# Soluția actuală

## Continuare

- Interval de valori posibile pentru variabile mai precis decât furnizat de LLVM
- Determinat din predicatele cu valoare de adevăr cunoscute
  - Valorile condiției din `if, while` pe calea de execuție curentă
  - Dacă putem ajunge într-un punct din program doar pe o anumită cale predicatele de pe acea cale sunt adevărate
  - Predicatele ne dau un interval nou de valori care sunt intersectate
  - De exemplu: `if (!size) return` → `size > 0`
  - Alt exemplu: `if (a < n)` → `n > 0`
- Dacă găsim un predicat care compară cele 2 părți din `umax` avem un răspuns
- Cunoscând intervalul de valori se înlocuiesc extremele în expresie, dacă obținem o constantă avem un răspuns

# Algoritm. Contribuții

1. Slicing care lucrează cu bitcode LLVM
2. Constrângeri pentru accesele la memorie reprezentate ca expressi Scalar Evolution
3. Inlining
4. Rezolvarea constrăngerilor folosind un solver
5. Afișăm mesaje de eroare, folosind informația de depanare din bitcode ca să aflăm linia sursă originală

# Exemplu de folosire

## Exemplu

```
$ llvm-gcc -O0 -g -m32 -emit-llvm -c x.c -o foo.bc
$ boundschecker foo.bc
```

```
x.c:5: Possible invalid memory access when accessing 4 =
getelementptr i8* 1, i32 argc
```

# Rezultate

Am realizat un instrument, care:

- E rapid, capabil să analizeze program cu >100k linii de cod sub un minut
- A găsit erori în programe reale
  - Xorg ( 480 KLOC) (`#15964`)
  - ClamAV ( 110 KLOC) (`#1639`), accese valide demonstrate: 5165 / 12950 ( 40%)
- Integrabil în procesul de compilare
- Module LLVM

# Eroarea găsită în Xorg

```
struct detailed_monitor_section {
   [...]
   union {
    struct whitePoints wp[2];
    [...]
   } section;
};
static void get_whitepoint_section(Uchar *c, struct whitePoints *wp)
{
   [...]
   wp[2].white_x = WHITEX2;
}
static void get_dt_md_section([...])
{
   [...]
   get_whitepoint_section(c,det_mon[i].section.wp);
}
```

# Îmbunătățiri viitoare

- Folosirea unui solver mai precis (SMT)
- Modelarea valorilor returnate de apeluri de sistem / bibliotecă
- Îmbunătățirea algoritmului de slicing ca să reducă și mai mult dimensiunea codului
- Integrare într-un modul open source în infrastructura LLVM

# Întrebări?