Code analysis and transformation

IPA example

Simone Campanoni
simonec@eecs.northwestern.edu
Title: Practical and Accurate Low-Level Pointer Analysis

Authors:
Bolei Guo Matthew J. Bridges Spyridon Triantafyllis Guilherme Ottoni Easwaran Raman David I. August

CGO, 2005
The two problems for CATs

• Problems:
  1. Identifying memory aliases
  2. Identifying callees of indirect calls

• Solutions:
  • Solve conservatively 1 first, and then 2
  • Solve 1 and 2 at the same time

VLLPA
Alias analysis for C programs

- Usually run once at the source level (the DDG is also computed)

- Compilation passes modify the IR, so they must update the DDG
  - Add complexity to each pass
  - Updates are conservative
Alias analysis for C programs

char A[10], B[10], C[10];

foo() {
    int i;
    char *p;

    for (i=0; i<10; i++) {
        if (...) {
            p = A;
        } else {
            p = B;
        }
        C[i] = p[i];
        A[i] = ...;
    }
}

(a) Source code

Instructions 3 and 4 may access the same memory location
VLLPA:  
a low level pointer analysis for C programs

• This paper proposes an alias analysis at the IR level
  • It can be run multiple times
  • No conservative updates
  • Passes are simpler
  • No data type information (not very useful for C anyway)

• The first context-sensitive and partially flow-sensitive low-level points-to analysis algorithm
VLLPA sequence

i: p = q

- $\text{GEN}[i] = \{\}$, $\text{KILL}[i] = \{\}$
- $\text{OUT}[i] = \{(p, z) \mid (q, z) \in \text{IN}[i]\} \cup (\text{IN}[i] - \{(p, x) \text{ for all } x\})$
VLLPA sequence

- Pointer points-to sets
- Memory alias sets
- Data dependence graph
Outline

• Abstractions used

• Data-flow intra-procedural analysis

• Inter-procedural analysis

• Evaluation
Memory abstraction

- **Abstract address** = memory location at analysis time =
- **Abstract structure** = contiguous set of abstract addresses

Memory is divided into a set of abstract structures, each with a unique name
- A single abstract structure can correspond to multiple blocks at runtime
- Unbounded set of memory blocks -> finite set of abstract names

- An abstract structure is created for each global variable
int myF (int arg0, int arg1){
    int v1, v2, v3;
    ... 
    int *p = &v1;
    ...
    ... = *p
    ...
    return v1+v2+v3;
}
Memory abstraction

• Activation frame:
  • One abstract structure for each
    • Element in the incoming parameter space
    • Element in the outgoing parameter space
    • Variable in the local variable space

• Heap object allocated:
  • Named according to the context (2 call stack depth)
Abstract structures

• \(<S,o>\)
  • \(S\) is a structure name
  • \(o\) is an offset

typedef struct {
    int64_t f1;
    int64_t f2;
} myT;

void myF (void){
    myT *p = (myT *)malloc(sizeof(myT));
    int *q = &(p->f2);
    ...
}

What is the abstract address pointed by \(p\)?
\(<p,0>\)

What is the abstract address pointed by \(q\)?
\(<p,8>\)
Abstract structures

• $<S,o>$
  • $S$ is a structure name
  • $o$ is an offset

• VLLPA merges all array elements
  • $\text{myArray}[5]$ is the same location of $\text{myArray}[42]$
  • Conservative assumption
    • More aliases
    • Much faster analysis
Abstract structures, pointer aliases, and dependencies

• Two pointers alias if there is an abstract address that they can both point to

• There is a dependence between two instructions if the pointers used by them alias
Abstract structures

- \(<S, o>\)
  - \(S\) is a structure name
  - \(o\) is an offset

typedef struct {
  int64_t f1;
  int64_t f2;
} myT;

void myF (myT *p){
  int *q = &(p->f2);
  ...
}

What is the abstract address pointed by \(p\)?
What is the abstract address pointed by \(q\)?
Unknown Initial Values (UIVs)

• They encode the “unknown”

• Represent memory blocks accessible by a function, but not created by either that function or its callees

• UIVs are created for memory blocks reachable (directly or indirectly) through parameters or global variables
Unknown Initial Values (UIVs)

- For a parameter A, 
  \([A]\) represents the memory block pointed by A

```c
void myF (void *P0){
    Var1 = P0
    ...
}
```

What is the abstract address pointed by Var1?
\(<[P0],0>\)
Unknown Initial Values (UIVs)

• If \([A]\) has a field at offset \(o\), which is a pointer, then the following new UIV is created: \([A]@o\)

```c
void myF (void *P0){
    Var1 = P0  Abstract structure pointed by Var1: <[P0],0>
    ...
    Var2 = Mem[Var1+4]  What is the abstract structure pointed by Var2?
    ...
}
```

• UIVs are created lazily
Outline

• Abstractions used

• Data-flow intra-procedural analysis

• Inter-procedural analysis

• Evaluation
Main challenge

• Common memory operations (array and field accesses) are not explicit in the code
  
  \[ Vx = Vy + 10 \] \hspace{1cm} \text{my\_struct\_t *Vy = ...}
  
  \[ Vz = \text{Mem}[Vx] \] \hspace{1cm} \text{int64\_t Vz = Vy->myField;}

• The analysis has to infer whether a memory operation “looks like” a field and/or array access
Intra-procedural analysis

- Assume SSA
  - One assignment per variable. Therefore
  - For each variable, we need to maintain a single points-to set
    \( R(\text{var}) \) = mapping from a variable to a set of abstract addresses that might point to

```c
void myF (void){
  int v1, v2;
  int *p, *q;
  int *p = &v1;
  int *q = &v2;
  if (rand()) p = q;
}
```

- \( R(v1) = \{ \} \)
- \( R(v2) = \{ \} \)
- \( R(p) = \{ v1,v2 \} \)
- \( R(q) = \{ v2 \} \)
Intra-procedural analysis

• Assume SSA
  • One assignment per variable. Therefore
  • For each variable, we need to maintain a single points-to set
    \( R(\text{var}) = \) mapping from a variable to a set of abstract addresses
    that might point to

• Not flow-sensitive for pointers in memory
  • Single points-to set for each abstract memory location
    \( M(\text{addr}) = \) mapping from an abstract address to a set of abstract addresses
    that might point to

• UIVs of the function analyzed
  • \( I(f) = \) set of UIVs of function \( f \)
Intra-procedural analysis

• Modify R, M, and I with a data-flow analysis

• Var1 = Mem[Var2]
  \[ R(\text{var1}) = \{ M(<S,o>) \mid <S,o> \in R(\text{var2}) \} \]

• Mem[Var1] = Var2
  For each \(<S,o> \in R(\text{Var1}):\n    M(<S,o>) \cup M(\text{Var1}) = R(\text{Var2}) \]

• Var1 = Var2 + c
  \[ R(\text{Var1}) = \{ <S,o+c> \mid <S,o> \in R(\text{Var2}) \} \]
Intra-procedural analysis

• Var1 = Var2 + Var3
  \( R(\text{Var1}) = \{ <\text{S}, o+c> \mid <\text{S}, o> \in R(\text{Var2}) \text{ and } c = \text{infer}\_\text{offset}(\text{Var3}) \} \cup\)
  \( \{ <\text{S}, o+c> \mid <\text{S}, o> \in R(\text{Var3}) \text{ and } c = \text{infer}\_\text{offset}(\text{Var2}) \} \)

Offset assumed to follow \( i \times l + c \)
\( l \) is the size of array elements
\( c \) is a constant displacement
(non-zero if the array is a structure field)

• VarX = PHI(Var1, Var2, ... VarN)
  • \( R(\text{VarX}) = R(\text{Var1}) \cup R(\text{Var2}) \cup ... \cup R(\text{VarN}) \)
Termination

• Data-flow analysis can only add new elements in R, M, and I
  • They increase monotonically

• To ensure termination: we need an upper bound to R, M, and I
  • Finite number of abstract addresses

• Do we have these upper bounds?
Termination: unbounded UIVs?

```c
typedef struct T {
    int data; T* next;
} T;

f(T* l) {
    while (l != NULL)
        ...
    l = l->next;
}
```

If <[UIV],c> ∈ R and <[UIV]@N,c> ∈ R, then remove the latter
Termination: what about the offsets?

```c
int A[100];

g() {
    int *a = A;
    while (...) {
        ... = *a;
        ...
        a++;
    }
}
```

```
R(r2) = {<[P0],0>, <[P0],4>, <[P0],8>, ...}
```

(c) Array: source

If `<S,o1> ∈ R` and `<S,o2> ∈ R` and `o1 < o2` then remove `<S,o2>`

(d) Array: low-level

```c
A:
    reserve 400

g:
    r1 = A
    LOOP:
        r2 = φ(r1, r4)
        r3 = mem[r2]
        ...
        r4 = r2 + 4
        br (...) LOOP
```
Intra-procedural analysis

• In all equations:
  • If \([UIV],c \in R\) and \([UIV]@N,c \in R\) then remove the latter

  Elements of the same list are represented as a single abstract address

• If \(<S,o1> \in R\) and \(<S,o2> \in R\) and \(o1 < o2\) then remove \(<S,o2>\)

  Elements of the same array are represented as a single abstract address
Outline

• Abstractions used

• Data-flow intra-procedural analysis

• Inter-procedural analysis

• Evaluation
Intra-procedural analysis

• Assume SSA
  • One assignment per variable. Therefore
  • For each variable, we need to maintain a single points-to set
    \( R(\text{var}) \) = mapping from a variable to a set of abstract addresses that might point to

• Not flow-sensitive for pointers in memory
  • Single points-to set for each abstract memory location
    \( M(\text{addr}) \) = mapping from an abstract address to a set of abstract addresses that might point to

• UIVs of the function analyzed
  • \( I(f) \) = set of UIVs of function \( f \)
VLLPA main blocks

• Intra-procedural analysis:
  • Compute R, M, I for every function in isolation

• Inter-procedural analysis:
  • Propagate M, I through the call graph
  • Map abstract addresses to UIVs
  • Update the call graph
VLLPA summary

• Summary: M, I
  \( M(\text{addr}) \) = mapping from an abstract address to a set of abstract addresses that might point to
  \( I(f) \) = set of UIVs of function f

• Transfer function
SCCDAG

- We compute SCCs of the call graph

- This SCCDAG is the graph where nodes are either functions or SCCs

- An SCCDAG has no cycles
Algorithm outline

SCCDAG is traversed in reverse topological order
Unknown initial values (UIV) assumed

First iteration: indirect calls have no target
Call graph is augmented with later iterations
SCCDAG is computed from the call graph
Algorithm outline

SCCDAG traversed in reverse topological order
Unknown initial values (UIV) assumed

SCCDAG traversed in topological order to resolve UIVs and indirect calls
Algorithm outline

SCCDAG is traversed in reverse topological order
Unknown initial values (UIV) assumed

Mapping abstract addresses of $F$ to UIVs of $G$
Algorithm outline

SCCDAG traversed in reverse topological order
Unknown initial values (UIV) assumed

SCCDAG traversed in topological order to resolve UIVs and indirect calls
Algorithm outline

The now complete SCCDAG is traversed once more in topological order to compute aliases and dependences.
Outline

• Abstractions used

• Data-flow intra-procedural analysis

• Inter-procedural analysis

• Evaluation
VLLPA evaluation

• Comparing against high-level language alias analysis

• Analysis time

• Accuracy of the analysis

• Performance of the generated binary
Evaluation:
Comparing alias analyses
# Evaluation:

## Analysis Time

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Procs</th>
<th># Opers</th>
<th># Indirect Calls</th>
<th>Time (s) VLLLPA</th>
<th>Time (s) IMPACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>epicdec</td>
<td>34</td>
<td>3998</td>
<td>0</td>
<td>0.770</td>
<td>0.116</td>
</tr>
<tr>
<td>g721dec</td>
<td>26</td>
<td>2396</td>
<td>1</td>
<td>0.035</td>
<td>0.150</td>
</tr>
<tr>
<td>g721enc</td>
<td>26</td>
<td>2395</td>
<td>1</td>
<td>0.036</td>
<td>0.091</td>
</tr>
<tr>
<td>gsmdec</td>
<td>94</td>
<td>11869</td>
<td>6</td>
<td>0.129</td>
<td>0.645</td>
</tr>
<tr>
<td>gsmenc</td>
<td>94</td>
<td>11869</td>
<td>6</td>
<td>0.146</td>
<td>0.472</td>
</tr>
<tr>
<td>mpeg2dec</td>
<td>114</td>
<td>10223</td>
<td>0</td>
<td>2.150</td>
<td>0.537</td>
</tr>
<tr>
<td>adpcmenc</td>
<td>3</td>
<td>288</td>
<td>0</td>
<td>0.071</td>
<td>0.061</td>
</tr>
<tr>
<td>adpcmdec</td>
<td>3</td>
<td>284</td>
<td>0</td>
<td>0.055</td>
<td>0.030</td>
</tr>
<tr>
<td>rasta</td>
<td>436</td>
<td>42500</td>
<td>7</td>
<td>3.880</td>
<td>2.428</td>
</tr>
<tr>
<td>099.go</td>
<td>372</td>
<td>55879</td>
<td>0</td>
<td>2.087</td>
<td>1.765</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>239</td>
<td>26663</td>
<td>3</td>
<td>4.584</td>
<td>1.357</td>
</tr>
<tr>
<td>129.compress</td>
<td>18</td>
<td>1211</td>
<td>0</td>
<td>0.268</td>
<td>0.0759</td>
</tr>
<tr>
<td>130.li</td>
<td>357</td>
<td>11953</td>
<td>4</td>
<td>14.843</td>
<td>73.340</td>
</tr>
<tr>
<td>132.ijpeg</td>
<td>473</td>
<td>33780</td>
<td>644</td>
<td>2.484</td>
<td>13.899</td>
</tr>
<tr>
<td>164.gzip</td>
<td>62</td>
<td>7346</td>
<td>2</td>
<td>0.764</td>
<td>0.339</td>
</tr>
<tr>
<td>175.vpr</td>
<td>255</td>
<td>25111</td>
<td>0</td>
<td>1.328</td>
<td>1.743</td>
</tr>
<tr>
<td>176.gcc</td>
<td>2220</td>
<td>463462</td>
<td>197</td>
<td>1495.318</td>
<td>1706.950</td>
</tr>
<tr>
<td>181.mcf</td>
<td>24</td>
<td>2157</td>
<td>0</td>
<td>0.285</td>
<td>0.1383</td>
</tr>
<tr>
<td>186.crafty</td>
<td>110</td>
<td>41370</td>
<td>0</td>
<td>1.543</td>
<td>0.694</td>
</tr>
<tr>
<td>197.parser</td>
<td>324</td>
<td>22686</td>
<td>0</td>
<td>2.835</td>
<td>3.388</td>
</tr>
<tr>
<td>254.gap</td>
<td>854</td>
<td>145017</td>
<td>1281</td>
<td>643.734</td>
<td>950.64</td>
</tr>
<tr>
<td>255.vortex</td>
<td>923</td>
<td>91864</td>
<td>15</td>
<td>12.107</td>
<td>42.330</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>63</td>
<td>6725</td>
<td>0</td>
<td>0.485</td>
<td>0.2746</td>
</tr>
<tr>
<td>300.twolf</td>
<td>167</td>
<td>53950</td>
<td>0</td>
<td>1.567</td>
<td>1.136</td>
</tr>
</tbody>
</table>
Evaluation: accuracy

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Oper w/ Arcs</th>
<th>VLLPA Arcs</th>
<th>More</th>
</tr>
</thead>
<tbody>
<tr>
<td>099.go</td>
<td>13232</td>
<td></td>
<td></td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>7161</td>
<td></td>
<td></td>
</tr>
<tr>
<td>129.compress</td>
<td>329</td>
<td></td>
<td></td>
</tr>
<tr>
<td>130.li</td>
<td>3762</td>
<td></td>
<td></td>
</tr>
<tr>
<td>164.gzip</td>
<td>1953</td>
<td></td>
<td></td>
</tr>
<tr>
<td>175.vpr</td>
<td>8166</td>
<td></td>
<td></td>
</tr>
<tr>
<td>181.mcf</td>
<td>705</td>
<td></td>
<td></td>
</tr>
<tr>
<td>186.crafty</td>
<td>12026</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256.bzip2</td>
<td>1535</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Evaluation: problem of alias analysis at the source language
Evaluation: problem of alias analysis at the source language

- Accurate
- Unnecessary
- Unnecessarily Propagated

- Same deps
- Better accuracy of VLLPA

Apparent deps generated by the conservative pass updates
Evaluation:
performance of the generated binary
Improved VLLPA in HELIX-RC (ISCA 2014)
After 2014

• Approximating Flow-Sensitive Pointer Analysis Using Frequent Itemset Mining
  Vaivaswatha Nagaraj and R. Govindarajan
  CGO 2015

• ... many others

• A Collaborative Dependence Analysis Framework
  Nick P. Johnson, Jordan Fix, Taewook Oh, Stephen R. Beard, Thomas B. Jablin, and David I. August
  CGO 2017