IR Deserts, Decompilation Swamps and Radeco

Sushant (@sushant94)
Xvilka (@akochkov)
R2Con 2016

Intermediate languages

- Intermediate language is the language of an abstract machine designed to aid in the analysis of computer programs (Wikipedia)
- Heavily used for academic research and real world tools
- Vital for decompilation process
- Base for various kind of applications - SMT, AEG, AEP, etc

Current limitations of varios IRs

- Lack of floating point support
- Limited architecture support
- Some of them are too big for SMT solving and effective decompilation
- Some of them are written in Java or OCaml, what makes reusing them difficult
- LLVM IR was created with **compilation** in mind
- Not Invented Here

ESIL - low level IR

- Evaluable Strings Intermediate Language
- Based on RPN (Reverse Polish Notation)(for the sake of speed)
- Designed for emulation and evaluation
- Small set of the instructions (relatively small)
- Implicit specification of the side effects
- String - thus easy to read with human eye

Radeco IL (implementation)

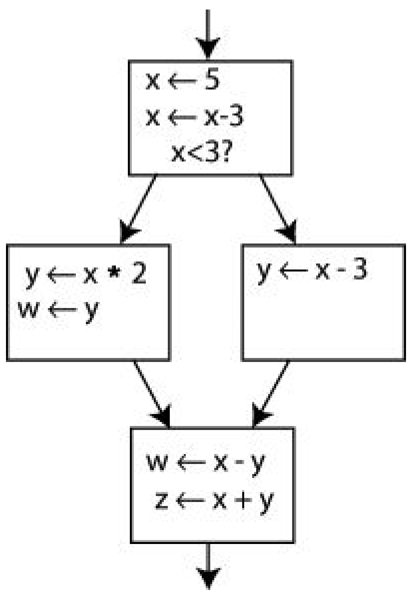
- Uses ESIL as input
- Request more metadata from radare2 (xrefs, functions, etc)
- Using r2pipe to talk to radare2
- Written in pure Rust
- Biggest part of it written during GSoC 2015 and GSoC 2016
- Authors are: Sushant Dinesh and David Kreuter

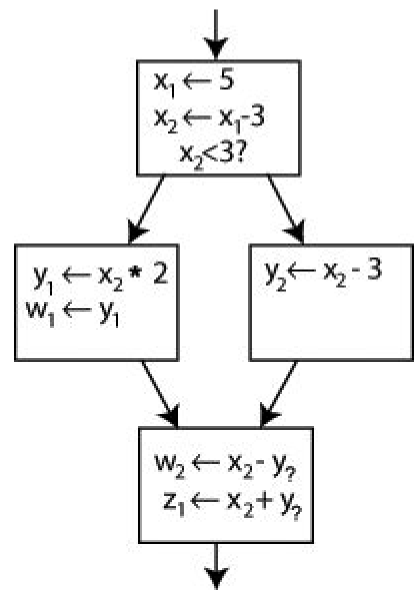
Decompilation stages - SSA

- Static Single Assignment form
- “each variable is assigned exactly once, and every variable is defined before it is used” (Wikipedia)
- Two different types of SSA: memory-based SSA and register based SSA
- Register fields and overlapping memory analysis

Decompilation stages - SSA

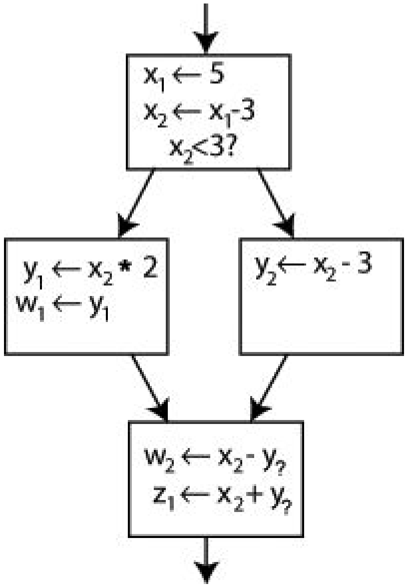
CFG before and after converting into SSA

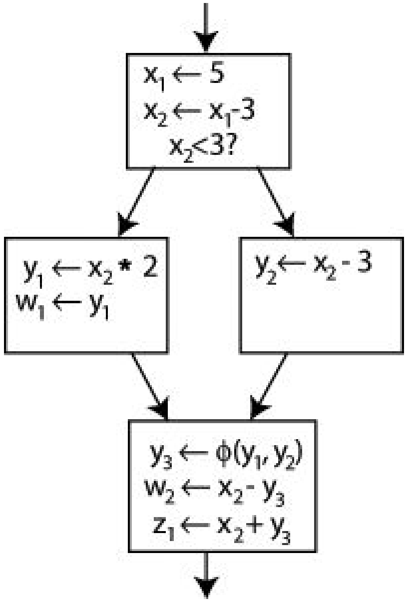




Decompilation stages - SSA

Handling conditional stages in SSA - Φ (Phi) function





Decompilation stages - SSA

Domination concept

- node **d** *dominates* a node **n** if every path from the *entry node* to **n** must go through **d**.
- A node ***d*** *strictly dominates* a node ***n*** if ***d*** dominates ***n*** and ***d*** does not equal ***n***.
- The *immediate dominator* or **idom** of a node ***n*** is the unique node that strictly dominates ***n*** but does not strictly dominate any other node that strictly dominates ***n***. Every node, except the entry node, has an immediate dominator.

Decompilation stages - SSA

Dominance frontier

The dominance frontier of a node ***d*** is the set of nodes that are “just barely” not dominated by ***d***; i.e., the set of nodes ***n***, such that:

- ***d*** dominates a predecessor ***p*** of ***n***, and
- ***d*** does not strictly dominate ***n***

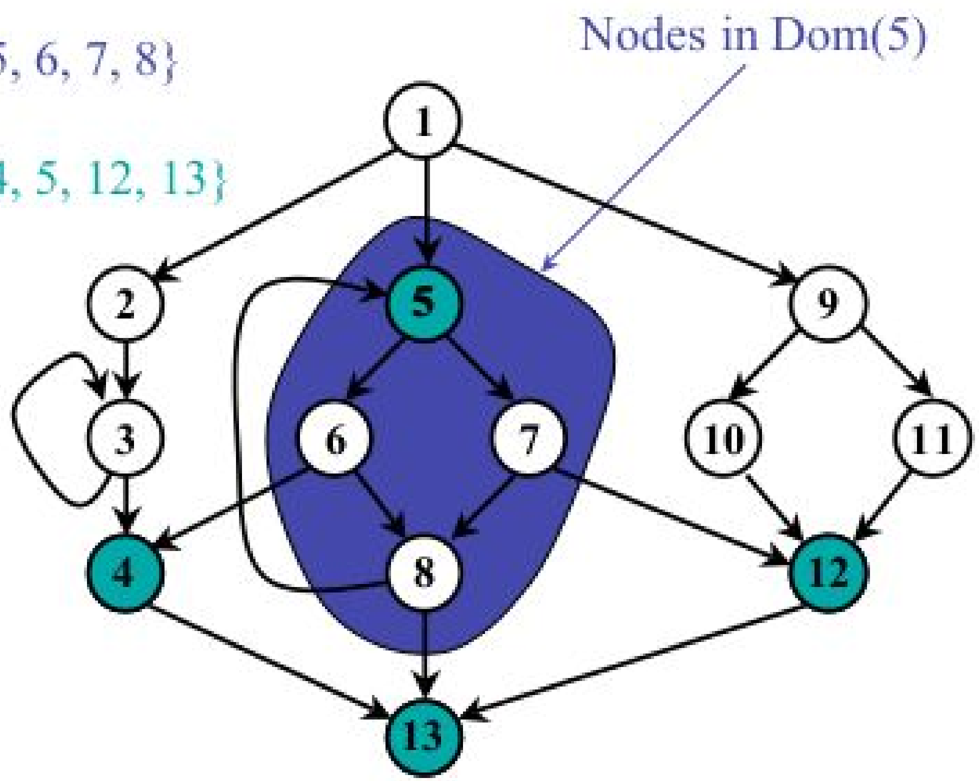
Decompilation stages - SSA

Dominance frontier

DF(d) = {n | ∃p∈pred(n), d dom p and d !sdom n}

Dom(5) = {5, 6, 7, 8}

DF(5) = {4, 5, 12, 13}



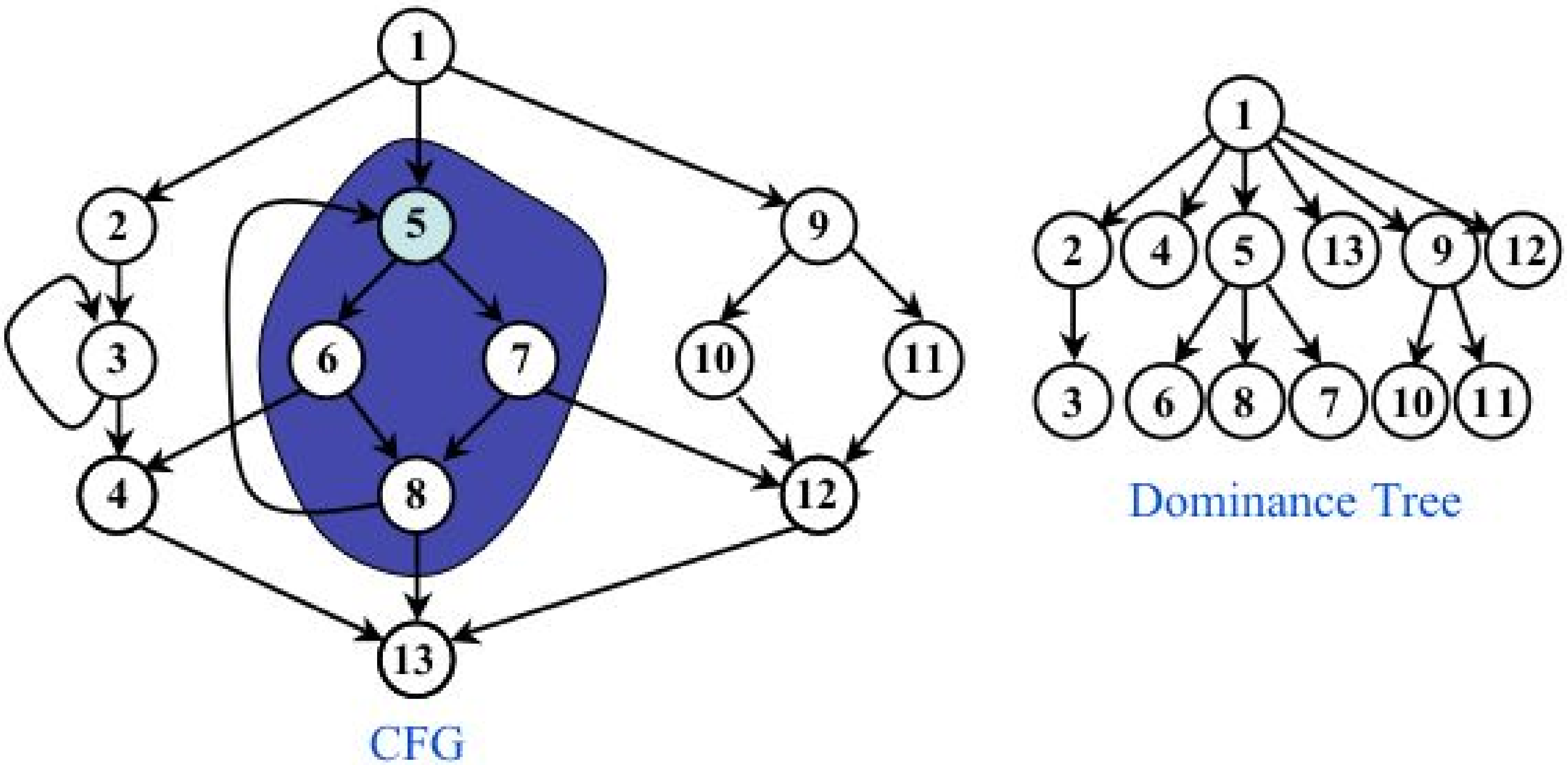
Decompilation stages - SSA

Dominator tree

- Show the dominance relation for each node
- Important step for ‘cleaning up’ SSA form
- A key for finding out correct variables names (eliminate copies of them)

Decompilation stages - SSA

Dominator tree



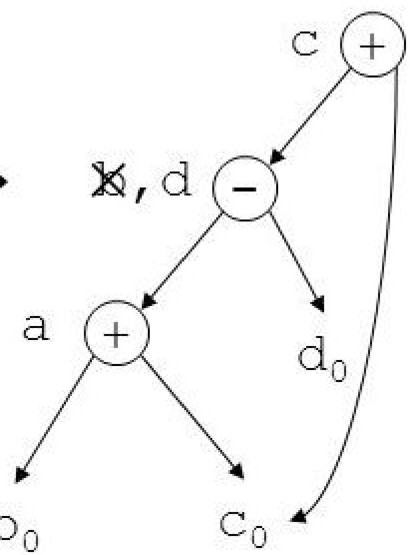
Decompilation stages - DCE

DCE == Dead Code Elimination

It removes unreachable paths from SSA tree

```
a := b + c
b := a - d
d := a - d
c := d + c
```

b is not live




```
a := b + c
d := a - d
c := d + c
```

Decompilation stages - Constant propagation

Also called as constant folding

$$
\begin{array}{lcl}
i_1 \leftarrow 1 & & \\
j_1 \leftarrow i_1 \times 4 & \Longrightarrow & j_1 \leftarrow 4 \\
\texttt{if}\ (j_1 + 1 > 2) & & \\
\quad i_2 \leftarrow 2 & & \\
\texttt{else} & & \\
\quad i_3 \leftarrow 3 & & \\
i_4 \leftarrow \phi(i_2, i_3) & \Longrightarrow & i_4 \leftarrow \phi(2) \\
k_1 \leftarrow 3 + i_4 & \Longrightarrow & k_1 \leftarrow 5
\end{array}
$$

Decompilation stages - Subexpression elimination

- Performing the compression of the dependent expressions
- Helps to shorten the output and simplify VSA stage

```
R1 = var_1

R2 = R1 + 1      =>      var_2 = var_1 + 1

var_2 = R2
```

Decompilation stages - VSA

Value Set Analysis

Tracking values not only of registers but also of bigger data objects

Collects direct integer and float values via program

Collects values by reference (by pointers)

Search the close-fitted pointers to form structures/unions

Decompilation stages - VSA

Value Set Analysis usually based on two concepts:

- Memory regions

Usually based on analysis of allocated memory

- A-locs

Finding out local and global variables - referenced directly and indirectly

Decompilation stages - VSA

```
 mov ebx, StructBuff  ; get address of first structure

.L1:
  ...
  mov al, [ebx+0] ; 1st structure field

  mov eax, [ebx+1] ; 2nd structure field
  mov eax, [ebx+5] ; 3rd structure field
  ...
  add ebx, 8 ; looping over the array of structures?
  loop .L1
```

Decompilation stages - Types propagation

- Collects all usages of V1 = V2, where V1 and V2 of the same type
- Simplest version of full fledged type inference
- Could be also done at disassembly level
- Not really needed in radeco, because of plans for full type inference

Decompilation stages - Types inference

1. Assign types to terms
2. Make function context
3. Load function prototype from external metainformation
4. Generate type contraints for every function and globally
5. Add standard types of platform into the equations
6. Add architecture-specific constraints
7. Feed those constraints to the SMT solver

Decompilation stages - Types inference

```
# include < stdlib .h >

struct LL {

    struct LL * next ;

    int handle ;

};

int close_last ( struct LL * list ) {

    while ( list->next != NULL ) {

        list = list->next ;

    }

    return close ( list->handle );

}
```

```
close_last :
        push ebp
        mov ebp , esp
        sub esp ,8
        mov edx , dword [ ebp + arg_0 ]
        jmp loc_8048402

loc_8048400 :
        mov edx , eax
loc_8048402 :
        mov eax , dword [ edx ]
        test eax , eax
        jnz loc_8048400

        mov eax , dword [ edx +4]
        mov dword [ ebp + arg_0 ] , eax
        leave
        jmp __thunk_ . close
```

Decompilation stages - External metainformation

- External system calls and shared libraries
- Debug information if presented
- Signature matching (for finding function prototypes and arguments' types)
- Reverse engineer markings at all stages
- Traces of ESIL emulation
- Memory/register dumps after the debugging

All this can be added into the decompilation engine to simplify VSA and types propagation/inference

See also

Static Single Assignment for Decompilation. Michael James Van Emmerik

Analyzing Memory Accesses in x86 Executables. Gogul Balakrishnan and Thomas Reps.

TIE: Principled Reverse Engineering of Types in Binary Programs. JongHyup Lee, Thanassis Avgerinos, and David Brumley.

DIVINE: DIscovering Variables IN Executables. Gogul Balakrishnan and Thomas Reps.

Dagger: Decompiling to IR. Christoph Erhardt.

Decompilation of LLVM IR. Simon Moll

Decompilation as search. Wei Ming Khoo.