Static binary analysis with a domain specific language

Julien Vanegue

EKOPARTY 2008

02/10/2008

Index

I. OVERVIEW

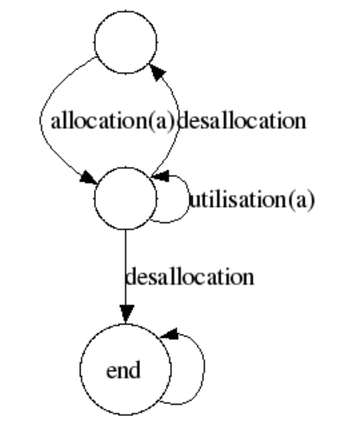
Static analysis

- Static analysis is determination of dynamic properties of program without executing it.
- Properties of interest can be about control flow (program paths) or data flow.
- Static analysis can be performed on source code level when available.
- We will focus here on binary level static analysis.

Typestate checking

- Typestate checking is the study of program's resources state evolution.
- Example: state of a memory zone, state of file descriptors..
- Ex: verify that a file is never closed twice.
- Ex: verify that a memory chunk is never freed twice.

Finite-state automata for correct use of memory allocation



Program transformation

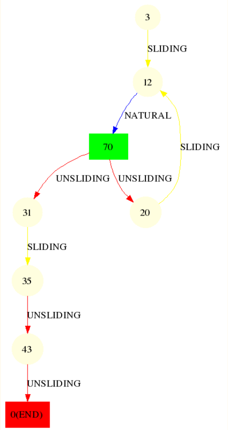
- A mechanism of successive derivation of program's structure.
- Transformations can put in light properties of interest for the transformed program.
- Ex: from a low-level intermediate representation (ASM) to an IR suitable for data-flow analysis (SSA).
- The VISITOR design pattern is very well adapted for implementing such transformations.

SPARC assembly

```
loop3: stb aa, [s + B]
       stb bb, [s + A]
       add aa, bb, aa
       and aa, 255, aa
       ldsb [s + aa], aa
       ldsb [data + I], bb

       xor aa, bb, aa
       stb aa, [data + I]

       add I, 1, I

       cmp I, data_len
       bne loop3
       nop
```

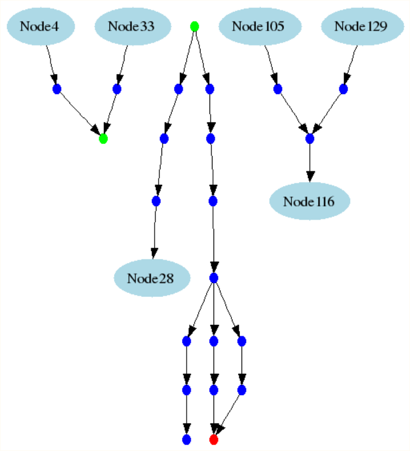
Control-flow

- Control-flow represents the sequentiality of execution between instructions's blocks.



Data-flow analysis

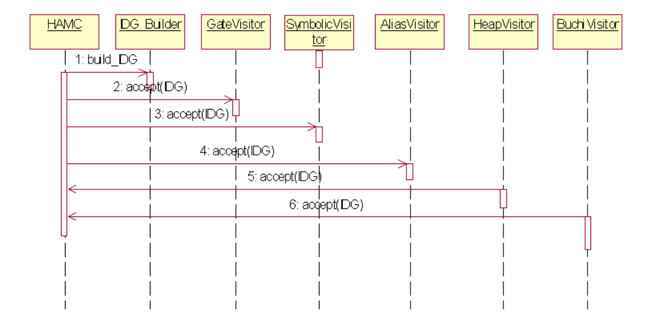
- Data flows represent dependences between variables of the program.



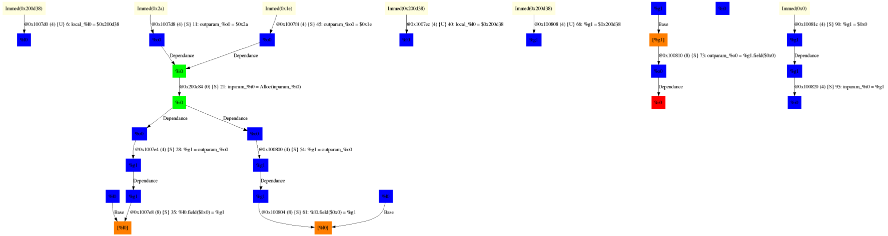
Common transformations

- Hierarchy of transformations.
- Architecture backend.
- Gate Visitor : interprocedural data-flow propagation.
- Alias Visitor: resolving problems due to variables with multiple names.
- Heap Visitor: determining the correctness of the program for heap properties.

Transformation hierarchy



Aliasing problems: graphical representation



Heap Visitor

- We reason about future states of a variable (forward analysis)
- Backward analysis is complementary and allows to reason about ancestors (dependences) of the variables.
- Examples : Memory leak in forward analysis, heap corruption in backward analysis.

II. RESULTS

Results

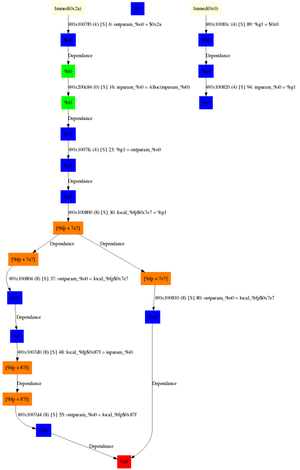
- Detection of multiple disallocations.
- Detection of memory leaks.
- Detection of heap corruptions.

Dummy source code

```
void titi(char *ptr)
{
  free(ptr);
}

int main()
{
  char *a;

  a = malloc(42);
  titi(a);
  free(a);
  return (0);
}
```

Detection of double free

and its data-flow



Detected error : double free

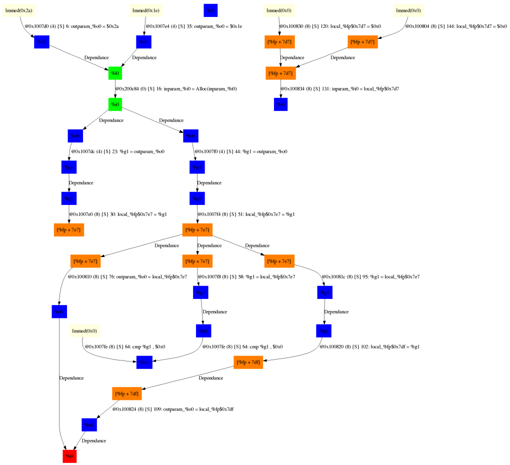
Analysis for variable : [%fp + 7e7] (double free)
* FIRST FREE:
@0x200c84 (0) S 16: inparam_%i0 = Alloc(inparam_%i0)
@0x1007fc (4) S 23: %g1 = outparam$_{\%}o0$
@0x100800 (8) S 30: local_%fp_0x7e7 = %g1
@0x100804 (8) S 37: outparam_%o0 = local_%fp_0x7e7
@0x1007d0 (8) S 48: local_%fp_0x87f = inparam_%i0
@0x1007d4 (8) S 55: outparam_%o0 = local_%fp_0x87f
@0x200ca4 (0) S 65: inparam_%i0 = Free(inparam_%i0)
* SECOND FREE:
@0x200c84 (0) S 16: inparam_%i0 = Alloc(inparam_%i0)
@0x1007fc (4) S 23: %g1 = outparam_%o0
@0x100800 (8) S 30: local_%fp_0x7e7 = %g1
@0x100810 (8) S 80: outparam_%o0 = local_%fp_0x7e7

@0x200ca4 (0) S 65: inparam_%i0 = Free(inparam_%i0)

Source code

```
int main()
{
  char *a;
  char *b;

  a = malloc(42);
  a = malloc(30);
  if (a)
    return (0);
  free(a);
  free(a);
  return (0);
}
```

Data-flow


Immed(0x2a)
Immed(0x1e)
Immed(0x0)
Immed(0x0)
@0x1007d0 (4) {S} 6: outparam_%o0 = $0x2a
@0x1007e4 (4) {S} 35: outparam_%o0 = $0x1e
@0x100830 (8) {S} 120: local_%fp$0x7d7 = $0x0
@0x100804 (8) {S} 144: local_%fp$0x7d7 = $0x0
%o0
%o0
[%fp + 7d7]
[%fp + 7d7]
Dependance
Dependance
Dependance
Dependance
%i0
[%fp + 7d7]
@0x200c84 (0) {S} 16: inparam_%i0 = Alloc(inparam_%i0)
@0x100834 (8) {S} 131: inparam_%i0 = local_%fp$0x7d7
%i0
%i0
Dependance
Dependance
%o0
%o0
@0x1007dc (4) {S} 23: %g1 = outparam_%o0
@0x1007f0 (4) {S} 44: %g1 = outparam_%o0
%g1
%g1
Dependance
Dependance
%g1
%g1
@0x1007e0 (8) {S} 30: local_%fp$0x7e7 = %g1
@0x1007f4 (8) {S} 51: local_%fp$0x7e7 = %g1
[%fp + 7e7]
[%fp + 7e7]
Dependance
Dependance
Dependance
[%fp + 7e7]
[%fp + 7e7]
[%fp + 7e7]
@0x100810 (8) {S} 76: outparam_%o0 = local_%fp$0x7e7
@0x1007f8 (8) {S} 58: %g1 = local_%fp$0x7e7
@0x10081c (8) {S} 95: %g1 = local_%fp$0x7e7
%o0
%g1
%g1
Dependance
Dependance
Immed(0x0)
%g1
%g1
@0x1007fe (8) {S} 64: cmp %g1 , $0x0
@0x1007fe (8) {S} 64: cmp %g1 , $0x0
@0x100820 (8) {S} 102: local_%fp$0x7df = %g1
%icc
[%fp + 7df]
Dependance
Dependance
[%fp + 7df]
@0x100824 (8) {S} 109: outparam_%o0 = local_%fp$0x7df
%o0
Dependance
%i0

Detected errors : double free

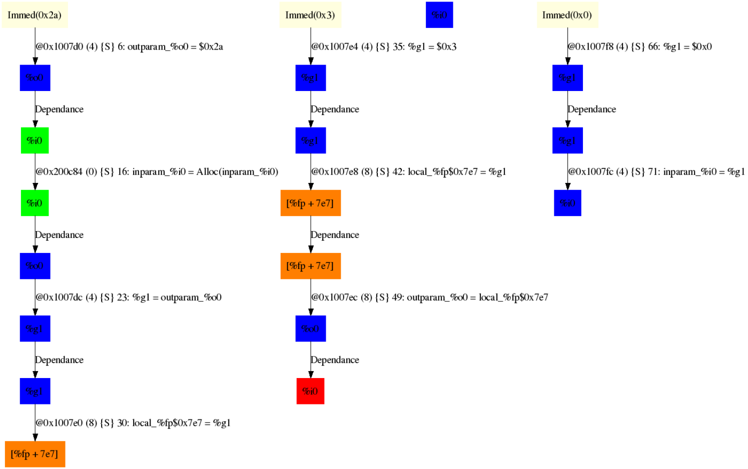
```
* Analysis for variable : [%fp + 7e7]
* PATH TO FREE1:
@0x200c84 (0) S 16: inparam_%i0 = Alloc(inparam_%i0)
@0x1007f0 (4) S 44: %g1 = outparam_%o0
@0x1007f4 (8) S 51: local_%fp_0x7e7 = %g1
@0x100810 (8) S 76: outparam_%o0 = local_%fp_0x7e7
@0x200ca4 (0) S 86: inparam_%i0 = Free(inparam_%i0)
* PATH TO FREE2:
@0x200c84 (0) S 16: inparam_%i0 = Alloc(inparam_%i0)
@0x1007f0 (4) S 44: %g1 = outparam_%o0
@0x1007f4 (8) S 51: local_%fp_0x7e7 = %g1
@0x10081c (8) S 95: %g1 = local_%fp_0x7e7
@0x100820 (8) S 102: local_%fp_0x7df = %g1
@0x100824 (8) S 109: outparam_%o0 = local_%fp_0x7df
@0x200ca4 (0) S 86: inparam_%i0 = Free(inparam_%i0)
* MISSING FREE:
@0x200c84 (0) S 16: inparam_%i0 = Alloc(inparam_%i0)
@0x1007dc (4) S 23: %g1 = outparam_%o0
@0x1007e0 (8) S 30: local_%fp_0x7e7 = %g1
```

Detection of heap corruptions

Source code

```
int main()
{
  char *a;

  a = malloc(42);

  a =( char *) 3;
  free(a);

  return 0;
}
```

Data-flow


Immed(0x2a)
@0x1007d0 (4) {S} 6: outparam_%o0 = $0x2a
%o0
Dependance
%i0
@0x200c84 (0) {S} 16: inparam_%i0 = Alloc(inparam_%i0)
%i0
Dependance
%o0
@0x1007dc (4) {S} 23: %g1 = outparam_%o0
%g1
Dependance
%g1
@0x1007e0 (8) {S} 30: local_%fp$0x7e7 = %g1
[%fp + 7e7]
Immed(0x3)
@0x1007e4 (4) {S} 35: %g1 = $0x3
%g1
Dependance
%g1
@0x1007e8 (8) {S} 42: local_%fp$0x7e7 = %g1
[%fp + 7e7]
Dependance
[%fp + 7e7]
@0x1007ec (8) {S} 49: outparam_%o0 = local_%fp$0x7e7
%o0
Dependance
%i0
%i0
Immed(0x0)
@0x1007f8 (4) {S} 66: %g1 = $0x0
%g1
Dependance
%g1
@0x1007fc (4) {S} 71: inparam_%i0 = %g1
%i0

Detected errors : heap corruptions

```
* Analysis for variable : [%fp + 7e7]
* MISSING FREE:
@0x200c84 (0) S 16: inparam_%i0 = Alloc(inparam_%i0)
@0x1007dc (4) S 23: %g1 = outparam_%o0
@0x1007e0 (8) S 30: local_%fp_0x7e7 = %g1
* FREE OF UNALLOCATED VARIABLE:
@0x1007e4 (4) S 35: %g1 = 3
@0x1007e8 (8) S 42: local_%fp_0x7e7 = %g1
@0x1007ec (8) S 49: outparam_%o0 = local_%fp_0x7e7

@0x200ca4 (0) S 59: inparam_%i0 = Free(inparam_%i0)
```

III. IMPLEMENTATION

Syntax for type declaration

type op = proc%4 len:uint ptr:*uchar type:uint size:uint content:uint
regset:int prefix:int imm:int baser:int indexr:int address_space:int scale:int name:*char

type instr = proc%4 instr:int type:int prefix:int spdiff:int flags:int

ptr_instr:*uchar annul:int prediction:int nb_op:int op1:op op2:op op3:op len:int

Syntax for variables declaration

```
Builtin-types: byte, short, long
type struct0 = field0:int
type struct1 = field1:int field2:int
type struct2 = field3:struct1 field4:long field5:struct1
type struct3::struct1 = field6:byte

long mylong = 42
struct0 mystruct0 = field0:42
struct0 mystruct0b = struct0(field0:42)
struct1 mystruct1 = (field1:42 field2:43)
struct2 mystruct2 = (field3(field1:42, field2:43), field4:44, field5:$mystruct1)
struct3 mystruct3 = field1:42 field2:43 field6:0xFF
```

Walking the CFG in ERESI

cfg-analyse.esh:

```
set $block $1
set $visitedblocks[$block.curaddr] 1
blocktransform $block

foreach $nextblock in $block.outlist

set $found $visitedblocks[$nextblock.curaddr]
cmp $found 1
je next
cfg-analyze $nextblock

next: forend
```

Using commands of ERESI-PT fragment (program transformation)

```
rewrite $mystruct

case SrcType1() into DstType1()

case SrcType2() into DstType2()::DstType3()

case SrcType3()
pre pre-commands;
into DstType4()
post post-commands;

case SrcType5() – DstType1()    ?    post-commands

endrwt
```

Attribute	Description
CALLPROC	The instruction performs a call (link) to a procedure
IMPBRANCH	The instruction performs a branch-always to another basic block
CONDBRANCH	The instruction performs a conditional branch to another basic block
RETPROC	The instruction returns from a procedure
ARITH	The instruction performs an arithmetic or logic operation
BITTEST	The instruction performs a bit test
BITSET	The instruction changes bit-level values
INT	The instruction triggers an interruption
LOAD	The instruction reads at a memory location
STORE	The instruction writes at a memory location
ASSIGN	The instruction performs an assignment
COMPARISON	The instruction performs a comparison
USEFLAG	The instruction use the content of status flags
WRITEFLAG	The instruction modifies the content of status flags
PROLOG	The instruction is part of a function prolog
EPILOG	The instruction is part of a function epilog
STOP	The instruction stops the process
NOP	The instruction does nothing

Table: Semantic attributes of assembly instructions in ERESI

Definition of semantics flags for machine instructions

```
# Attributes for ASM instructions
define b ASM_TYPE_IMPBRANCH
define cb ASM_TYPE_CONDBRANCH
define c ASM_TYPE_CALLPROC
define i ASM_TYPE_INT
define r ASM_TYPE_RETPROC
define p ASM_TYPE_PROLOG
define cmp ASM_TYPE_COMPARISON
define bs ASM_TYPE_BITSET
define a ASM_TYPE_ASSIGN
define wm ASM_TYPE_STORE
define rm ASM_TYPE_LOAD
define e ASM_TYPE_EPILOG
define s ASM_TYPE_STOP
define n ASM_TYPE_NOP
define ar ASM_TYPE_ARITH
define wf ASM_TYPE_WRITEFLAG
define i-r i r
define ar-wf ar wf
define ar-wm ar wm
```

ELIR Operation	Description
Call	Direct Call (link) to a procedure
IndCall	Indirect Call (link) to a procedure
Branch	Direct Branch to a basic block
IndBranch	Indirect Branch to a basic block
Return	Return from a procedure
Interrupt	Trap or interruption
Stop	End of program
Nop	No operation
TernopR3	Arithmetic operation between 2 registers with result in third register
TernopRI	Arithmetic op between register and immediate
AssignIR	Assignment of an immediate value into a register
AssignIM	Assignment of an immediate value into the memory
AssignRR	Assignment of a register value into another register
AssignRM	Assignment of a register value into the memory
AssignMR	Assignment of a memory value into a register
CmpRI	Comparison between immediate value and register
CmpRR	Comparison between two registers
BitTest	Test of bit-level values in a register
BitSet	Set of bit-level values in a register
Prolog	Stack frame allocation for a new procedure
Epilog	Stack frame destruction for the current procedure

Definition of ELIR in the ERESI meta-language

```
# Types of ELIR operands
type Reg = id:int
type Immed = val:long
type Mem = base:Reg off:Immed name:string
type Ins = uflags:Immed addr:Immed

# Types of ELIR instructions
type IndBranch::Ins = dst:Reg
type Branch::Ins = dst:Immed
type Call::Ins = dst:Immed
type IndCall::Ins = dst:Reg
type Interrupt::Ins = dst:Immed
type Return::Ins = dst:Immed
type TernopR3::Ins = dst:Reg src1:Reg src2:Reg
type TernopRI::Ins = dst:Reg rsrc:Reg isrc:Immed
type AssignIR::Ins = dst:Reg src:Immed
type AssignIM::Ins = dst:Mem src:Immed
type AssignMR::Ins = dst:Reg src:Mem
type AssignRM::Ins = dst:Mem src:Reg
type BitSet::Ins = src:Immed dst:Reg
type CmpRI::Ins = fst:Immed snd:Reg
```

Transformation of machine code to ELIR

```
set $curblock $1
set $curaddr $curblock.vaddr

foreach $instr of $instrlists[$curaddr]

rewrite $instr

ld,ldd,ldub,ldx,lduw,lduh,ldsw,ldsb
case instr(type:a-rm)
into AssignMR(addr:$curaddr src(name:$instr.op1.name base(id:$instr.op1.baser)
off(val:$instr.op1.imm))
dst(id:$instr.op2.baser), uflags:0)

st,stb,stw,sth,std,stx
case instr(type:a-wm)
into AssignRM(addr:$curaddr src(id:$instr.op1.baser) dst(name:$instr.op2.name
base(id:$instr.op2.baser)
off(val:$instr.op2.imm)) uflags:0)
```

What is ELIR useful to ?

- Make us able to analyze machine code independently of the architecture
- Simplify further data-flow analysis by making memory accesses more explicits in the analyzed program
- More generally: simplify analysis by making explicit all effects of assembly instructions (especially useful on CISC architecture like Pentium)

Data dependence analysis of ELIR programs

```
set $curb $1
set $curaddr $curb.vaddr
set $ilist $instrlists[$curaddr]

foreach $instr in $ilist

rewrite $instr

case TernopRI() post use $instr.rsrc $curb.registry; def $instr.dst $curb.registry

case AssignIR() post def $instr.dst $curb.registry

case AssignIM() post use $instr.dst.base $curb.registry; def $instr.dst.name
$curb.registry

case AssignMR() post use $instr.src.base $instr.src.name $curb.registry; def $instr.dst
$curb.registry
```

Questions

- ?