Value-Analysis of REIL programs (Technical Report)

Sanjay Rawat, Laurent Mounier
Verimag, Grenoble France

1 Motivation

As a part of our work on *smart fuzzing*, and more specifically on *taint-analysis*, we describe here a static analysis technique to over-approximate the value contained in each register and memory location accessed at each execution point of a Reil program. The motivation is twofold:

1. To identify the (abstract) addresses of memory locations used in the program. This is mandatory to follow how tainted values are propagated during the program execution ;
2. To approximate at each execution point, the current offset in the input flows. The goal here is to mark each tainted value with its *origin* (as an offset in a given input flow).

To illustrate the first point, let us consider the following example of C program with its associated Reil translation (Fig. 1). `x`, `y` and `z` are local variables of a given procedure (stored at offsets -4, -8 and -12 from the base pointer `ebp`), and `t` is a global variable, supposed to be tainted, and those address is in register `t0`. A taint analysis performed on this program should indicate that variable `z` is not tainted at the end of the execution. However, to get this result at the Reil level, it is necessary to identify that the value written at offset -12 (line 10) comes from the memory location written at line 3. Hence, we need to know that the values of registers `t5` and `t4` are equals. This property will be provided by the value-analysis technique we describe below.

```
                                    1:   STR 3, ,t3          // t3 := 3
                                    2:   ADD ebp, -8, t4     // t4 := ebp-8
                                    3:   STM t3, ,t4         // Mem[t4] := t3
y = 3 ;                             ...
...                                 4:   LDM t0, , t1        // t1 := Mem[t0]
x = t ;                             5:   ADD ebp, -4, t2     // t2 := ebp-4
z = y ;                             6:   STM t1, ,t2         // Mem[t2] := t1
                                    7:   ADD ebp, -8, t5     // t5 := ebp-8
                                    8:   LDM t5, , t6        // t6 := Mem[t5]
                                    9:   ADD ebp, -12, t7    // t7 := ebp-12
                                    10:   STM t6, , t7       // Mem[t7] := t6
```

Figure 1: A small C program and its corresponding REIL translation

2 Intraprocedural Value-Analysis

First, we describe how this value-analysis is carried out at the procedure level.

2.1 Notations

We assume in the following that:

- The target program is a finite sequence of REIL instructions $\mathcal{I} = [i_1, i_2, \ldots \imath_n]$.
- During the execution of $\mathcal{I}$ a finite set of registers $\mathcal{R}$ and a (possibly infinite) set of memory locations $\mathcal{M}$ are accessed. The exact structure of $\mathcal{M}$ we consider is detailed in section 2.2.
- The set $\mathcal{R}$ consists in general-purpose registers $\{t_0, t_1, \ldots, t_m\}$ and Intel X86 specific registers SR (like the stack pointer `esp`, the frame pointer `ebp`, etc.). Remember that in REIL each instruction i_k *defines* (i.e., assigns) at most one register. In the following this register will be noted $reg(i_k)$.
- The initial value of the stack pointer `esp` when entering a procedure is noted $init_{esp}$. We assume that all other registers are initialized w.r.t. `esp` during the course of the procedure.

For a given register or memory location $x \in \mathcal{R} \cup \mathcal{M}$, we denote by $Val_c\,(x, i_k)$ the set of *all possible (concrete) values* of x after execution of any occurrence of instruction i_k (whatever is the execution sequence leading to i_k). In particular $Val_c\,(x, i_k) = \emptyset$ if i_k is not reachable.

2.2 Memory Model

The memory model we consider consists in a (potentially finite) set of memory cells of fixed-size (say 1 byte). Some of these cells are addressed during a program execution either for reading (using a `LDM` instruction) or for `writing` (using a `STM instruction`). In Reil each `LDM` or `STM` operation may access a fixed amount of consecutive memory cells (either 1, 2, or 4). In the following we call a *memory location* such a set of consecutive memory cells accessed at program execution.

From the source level point of view, this set of memory locations may include:

- the local variables and parameters to the procedure under consideration ;
- the global variables ;
- the dynamically allocated memory.

At the binary level, the addresses of these memory locations will be either absolute values (for global variables), or computed from the current value of the base pointer register `ebp` (for parameters and local variables), or obtained as a result of memory allocation functions (like `malloc()`).

As explained before, one of our objective is to compute *statically*, at the *binary level*, the addresses of these memory locations. This means facing the following difficulties:

- the exact value of `ebp` is unknown ;
- the value returned by a `malloc()` is unknown ;
- accessing non-scalar variables (e.g., arrays) requires some arithmetic computations ;
- distinct registers may be used to address the same memory location ;
- the set of memory locations actually accessed depends on the program execution (i.e., it is not known statically), it could even be infinite . . .
- etc.

For these reasons, our algorithm will compute *abstract addresses* of memory locations, meaning that:

- an abstract address will be defined as *offsets* w.r.t. some *reference value* (the value of *esp*, the value returned by a given call to `malloc()`, etc.) ;

- an abstract address may represent a set of possible real addresses (over-approximation) ;
- only a finite set of distinct offsets will be considered for a given reference value (widening to a special address meaning "any possible address").

Note that in general we should deal with *memory location overlapping*, meaning that writing to a memory location of size 4 located abstract address a also impacts the memory location of size 2 located at abstract address $a + 2$ (for instance).

We will leave this issue for further works, considering in the following that:

- all memory locations are of size 4 ;
- each (concrete) address is a multiple of 4.

2.3 Value computation as a forward data-flow analysis

The technique we used can be viewed as generalization of a constant propagation algorithm implemented as a data-flow analysis. It computes a set of abstract value (where each abstract value is a *superset* of some concrete values) for each register and each memory location. These abstract values are associated to the input and output of each node (i.e., instruction) of a control-flow graph (CFG) of the program. More precisely:

- The set of abstract values of each register and memory locations are gathered into a *state vector* ;
- State vectors S_{in}^n (resp. S_{out}^n) are attached to the input (resp. output) of each node n of the CFG ;
- A partial order relation is defined over the set of state vectors to form a lattice, where higher abstract values are "less precise" than lower ones ;

The algorithm computes the *more precise* state vectors associated to each node compatible with the program execution. To do so, the program semantics is abstracted by a *transfer functions* $\mathcal{F}_\rangle$ describing how the execution of each instruction i may transform a state vector. Abstract values coming from different execution paths are combined using a *merge operator* $\sqcup$, defined as a the least-upper bound in the lattice.

Formally, the final value of state vectors S_{in}^n and S_{out}^n is defined as the least fix-point of the following function, where $S0_{in}$ denotes the initial value of the state vector at the input of the entry node of the CFG[1]:

$$S^n_{in} = \begin{cases} S0_{in} \text{ if n is the entry point of the CFG} \\ \displaystyle\bigsqcup_{n' \in \text{Pred}\,(n)} S^{n'}_{out} \text{ otherwise .} \end{cases}$$

$$S^n_{out} = \mathcal{F}\,(S^n_{in})$$

This fix-point is computed iteratively, starting from the bottom value $\perp$ of the lattice. Note that a widening operator has to be used to ensure termination. In the following we give a precise definitions of the lattice and transfer function we used.

2.4 Abstract domain

The abstract domain $\mathcal{A}$ we use to represent abstract values is borrowed from the ones proposed in [6] and [1, 9] for "abstract address sets". More precisely, an abstract value will be expressed as a pair $< B, X >$, where:

- B is an (abstract) "base value", which can be either
  - an instruction (address) i_k
  - an element of the set $\{\text{Empty}, \text{None}, \text{InitEsp}, \text{Any}\}$
- X is a finite set of integers ($X \subseteq \mathbb{Z}$).

2.4.1 Abstract values

Each abstract value $< B, X >$ represents a set of concrete values $v \in Z$ defined as follows:

$$Val_c\,(< B, X >) = \begin{cases} \emptyset \text{ if } B = \text{Empty} \\ \mathbb{Z} \text{ if } B = \text{Any} \\ X \text{ if } B = \text{None} \\ \{init_{esp} + x \mid x \in X\} \text{ if } B = \text{InitEsp} \\ \{v + x \mid v \in Val_c\,(reg(i_k), i_k)\} \text{ if } B = i_k \end{cases}$$

[1]this state vector simply associates the value InitEsp to register `esp` according to our assumptions regarding register initializations.

Intuitively this means that an abstract value $< B, X >$ will be either a constant belonging to X (when B is None), or an address defined with respect to `esp` (when B is InitEsp), or with respect to the value of the destination register computed at a given instruction i_k (when B is i_k), or finally any possible value (when B is Any).

Note that, for all sets X and Y, the two following equalities holds (from the previous definition):

$$\begin{aligned} Val_c(< \text{Empty}, X >) &= Val_c(< \text{Empty}, Y >) \\ Val_c(< \text{Any}, X >) &= Val_c(< \text{Any}, Y >) \end{aligned}$$

2.4.2 Comparing abstract values

We define here some comparison relations between abstract values with respect to the set of concrete values they represent. First, we consider two main notions of *equality* between abstract values:

- a "strong equality" ($\equiv$), expressing that two abstract values represent (exactly) the same sets of concrete values;
- a "weak equality" ($\cong$), expressing that two abstract values represent non disjoint sets of concrete values.

However, abstract values of the form $< i_k, X >$ are *context-dependent*, since their associated concrete values depend on the position of the current CFG location with respect to i_k. Therefore, comparing such values needs to take into account CFG locations as well. To do so, we introduce the predicate $SVal_c(i, (p_1, p_2))$, defined in [6], indicating when two abstract values computed at location p_1 and p_2, and depending on (path dependent) concrete values computed at location i_k, can be considered as "comparable":

$SVal_c(i_k, (p_1, p_2)) =$
(i_k dominates both p_1 and p_2)
$\vee$
(either p_1 dominates p_2 or p_2 dominates p_1).

In particular we assume that $SVal_c(i_0, (l_1, l_2))$ holds for all (l_1, l_2) when i_0 is the entry point of the procedure.

The small example below illustrates a situation where predicate $SVal_c$ does not hold:

```
1. while (i < N) {
     2. t1 = x ;  //  abstract value of t1 = abstract value of x
     3. read(x) ; // abstract value of x = <3, {0}>
     4. t2 = x ;  //  abstract value of t2 = abstract value of x
     // beware, t1 and t2 may not contain the same value !
}
```

• **Strong equality ($\equiv$):**

$$< B_1, X_1 > \text{ at } l_1 \ \equiv < B_2, X_2 > \text{ at } l_2 \text{ iff} \begin{cases} B_1 = B2 = \text{Any} \\ \text{or } B_1 = B2 = \text{Empty} \\ \text{or } B_1 = B_2 = \text{None and } X_1 = X_2 \\ \text{or } B_1 = B_2 = \text{InitEsp and } X_1 = X_2 \\ \text{or } B_1 = B2 = i \text{ and } X_1 = X_2 \text{ and } SVal_c(i, (l_1, l_2)) \end{cases}$$

The following property holds:

$$< B_1, X_1 > \text{ at } l_1 \ \equiv < B_2, X_2 > \text{ at } l_2 \ \Leftrightarrow \ Val_c\,(< B_1, X_1 >) = Val_c\,(< B_2, X_2 >)$$

The definition of weak equality depends on some general assumptions on the memory model and on the way memory locations are accessed. In particular we could identify two typical situations:

- An *optimistic* assumption ($\mathcal{A}_o$), saying that distinct memory locations (stack, heap and global variables) are non overlapping, that stack or heap addresses are not known statically (i.e., they cannot be expressed by constant values in the program), and that distinct calls to `malloc` will allocate distinct memory zones:

  $\forall X.\ \forall i_k.\ Val_c(< \text{InitEsp}, X >) \neq Val_c(< \text{None}, X >) \neq Val_c(< i_k, X >)$

- A *pessimistic* assumption ($\mathcal{A}_p$), saying that none of the previous hypotheses necessarily holds:
  $\exists X.\ \exists i_k.$
  $\quad Val_c(< \text{InitEsp}, X >) \cap Val_c(< \text{None}, X >) \neq \emptyset$
  $\vee$
  $\quad Val_c(< \text{InitEsp}, X >) \cap Val_c(< i_k, X >) \neq \emptyset$
  $\vee$
  $\quad Val_c(< \text{None}, X >) \cap Val_c(< i_k, X >) \neq \emptyset$

In the following we will consider assumption $\mathcal{A}_o$ as the default one.

• **Weak equality ($\cong$) under assumption $\mathcal{A}_o$:**

$$< B_1, X_1 > \text{ at } l_1 \cong < B_2, X_2 > \text{ at } l_2 \text{ iff } \begin{cases} B_1 = \text{Any and } B_2 \neq \text{Empty} \\ \text{or } B_2 = \text{Any and } B_1 \neq \text{Empty} \\ \text{or } B_1 = B_2 = \text{None and } X_1 \cap X_2 \neq \emptyset \\ \text{or } B_1 = B_2 = \text{InitEsp and } X_1 \cap X_2 \neq \emptyset \\ \text{or } B_1 = B2 = i \text{ and} \\ \qquad SVal_c(i, (l_1, l_2)) \text{ and } X_1 \cap X_2 \neq \emptyset \end{cases}$$

- **Weak equality ($\cong$) under assumption $\mathcal{A}_o$:**

$$< B_1, X_1 > \text{ at } l_1 \cong < B_2, X_2 > \text{ at } l_2 \text{ iff } \begin{cases} B_1 = \text{Any and } B_2 \neq \text{Empty} \\ \text{or } B_2 = \text{Any and } B_1 \neq \text{Empty} \end{cases}$$

The following property holds:

$$< B_1, X_1 > \text{ at } l_1 \cong < B_2, X_2 > \text{ at } l_2 \Leftrightarrow Val_c(< B_1, X_1 >) \cap Val_c(< B_2, X_2 >) \neq \emptyset$$

We now define a (partial) order relation $\preceq$ over $\mathcal{A}$.

- **Partial order relation ($\preceq$):**

$$< B_1, X_1 > \preceq < B_2, X_2 > \text{ iff } \begin{cases} B_2 = \text{Any} \\ \text{or } B_1 = \text{Empty} \\ \text{or } B_1 = B_2 = \text{None and } X_1 \subseteq X_2 \\ \text{or } B_1 = B_2 = \text{InitEsp and } X_1 \subseteq X_2 \\ \text{or } B_1 = B2 = i \text{ and} \\ \qquad SVal_c(i, (l_1, l_2)) \text{ and } X_1 \subseteq X_2 \end{cases}$$

Intuitively, under assumption $\mathcal{A}_o$, $< B_1, X_1 > \preceq < B_2, X_2 >$ means that $< B_1, X_1 >$ is *more precise* than $< B_2, X_2 >$ (i.e.. it contains less concrete values):

$$< B_1, X_1 > \preceq < B_2, X_2 > \Leftrightarrow Val_c(< B_1, X_1 >) \subseteq Val_c(< B_2, X_2 >)$$

According to these definition, $(\mathcal{A}, \preceq)$ is a lattice those greatest element is $\top =< \text{Any}, \mathbb{Z} >$ and those least element is $\perp =< \text{Empty}, \emptyset >$.

2.4.3 Merging abstract values

We also define a *least upper-bound* operator $\sqcup$:

$$< B_1, X_1 > \sqcup < B_2, X_2 >= \begin{cases} < \text{Any}, \emptyset > \text{ if } B_1 \neq B_2 \neq \text{Empty} \\ < B_1, X_1 \cup X_2 > \text{ if } B_1 = B_2 \\ < B_1, X_1 > \text{ if } B_2 = \text{Empty} \\ < B_2, X_2 > \text{ if } B_1 = \text{Empty} \end{cases}$$

The structure of the lattice $(\mathcal{A}, \preceq)$ is depicted in Fig 2.

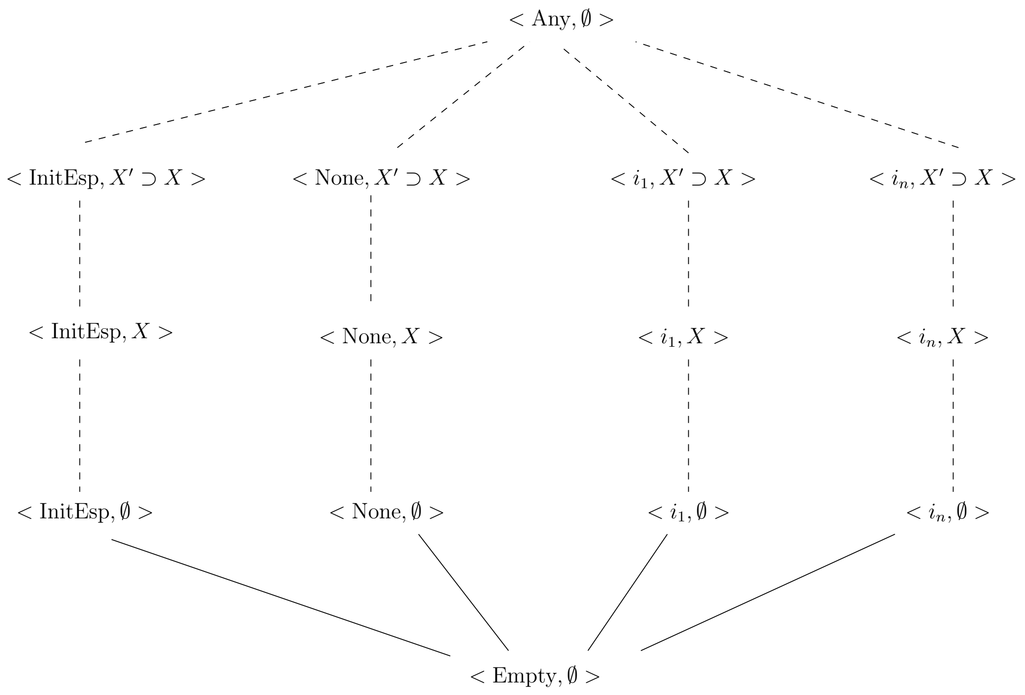
$< \text{Any}, \emptyset >$
$< \text{InitEsp}, X' \supset X >$
$< \text{None}, X' \supset X >$
$< i_1, X' \supset X >$
$< i_n, X' \supset X >$
$< \text{InitEsp}, X >$
$< \text{None}, X >$
$< i_1, X >$
$< i_n, X >$
$< \text{InitEsp}, \emptyset >$
$< \text{None}, \emptyset >$
$< i_1, \emptyset >$
$< i_n, \emptyset >$
$< \text{Empty}, \emptyset >$

Figure 2: The lattice $(\mathcal{A}, \preceq)$

2.5 The lattice of state vectors

The purpose of our data-flow analysis is to collect information about the possible values of each register and each memory locations used throughout the procedure execution. Individual registers are simply referred by their (unique) name, whereas *sets of* memory locations have to be referred by their *abstract addresses.*

A state vector S is therefore a *total function* from $(\mathcal{R} \cup \mathcal{A})$ to $\mathcal{A}$ such that $S(l) = a$ iff the value contained in l (where l is either a register or a set of memory locations) is a

Note that, for a given state vector S_{in}^{n} obtained at iteration k of the fix-point computation algorithm, $S_{in}^{n}(l) =<$ Empty$, X >$ means that the value of l *has not been computed so far* (at this iteration) when entering node n.

The relation $\preceq$ and the binary operator $\sqcup$ are now extended to state vectors:

$$\begin{array}{rcl} S_1 \preceq S_2 & \text{iff} & \forall x \in (\mathcal{M} \cup \mathcal{A}).\ S_1(x) \preceq S_2(x) \\ S_1 \sqcup S_2 & = & \{S_1(x) \sqcup S_2(x) \mid x \in (\mathcal{M} \cup \mathcal{A})\} \end{array}$$

Note that, in the lattice of state vectors ordered by $\preceq$, $\perp$ is the function which is always equal to $<$ Empty$, \emptyset >$ and $\top$ is the function which is always equal to $<$ Any$, \emptyset >$.

Remark: Alternatively, a state vector could have been defined as a *partial function* from $(\mathcal{R} \cup \mathcal{A})$ to $\mathcal{A}$, assuming that undefined elements are equal to $<$ Empty$, \emptyset >$. This definition will be used in the implementation, since it does not require $\mathcal{M} \cup \mathcal{A}$ to be known *a priori*: new state vector entries will be added only when required by the current computation. $\Box$.

$\boxed{\text{It could better to consider a special UNKNOWN value instead of Empty ?}}$

2.6 Transfer Functions $\mathcal{F}_i$

We now describe how the execution of each REIL instruction i_k transforms an input state vector S_{in} into an output state vector S_{out}. We note $S_{in}(t_i) =< B_i, X_i >$ and S_{out} is defined instruction-wised by the transfer function $\mathcal{F}_i$ associated to i_k as explained below.

Transfer between register: `STR t0, ,t1`

$$S_{out} = S_{in}[\texttt{t1} \rightarrow S_{in}(\texttt{t0})]$$

Addition or subtraction: `ADD t0, t1, t2`

$$S_{out} = S_{in}[\texttt{t2} \rightarrow \mathit{Add}(S_{in}(\texttt{t0}), S_{in}(\texttt{t1}))]$$

where the function *Add* is defined as follows:

$$Add(< B_1, X_1 >, < B_2, X_2 >) = \begin{cases} < B_1, \{x_1 + x_2 \mid x_1 \in X_1 \wedge x_2 \in X_2\} > \\ \qquad \text{if } B_1 = B_2 \text{ and } B_1 \neq \text{None and } B_1 \neq \text{Any} \\ < B_1, \{x_1 + x_2 \mid x_1 \in X_1 \wedge x_2 \in X_2\} > \text{ if } B_2 = \text{None} \\ < B_2, \{x_1 + x_2 \mid x_1 \in X_1 \wedge x_2 \in X_2\} > \text{ if } B_1 = \text{None} \\ < i_k, \{0\} > \text{ otherwise} \end{cases}$$

In case of at least one operand is "unknown" in S_{in} (i.e., equal to $<$ Empty, $\emptyset >$) then the result will be unknown as well in S_{out}. A similar definition holds for a `SUB` instruction.

Other arithmetic and logic operations: `MUL t0, t1, t2`

$$S_{out} = S_{in}[\texttt{t2} \to Mul(S_{in}(\texttt{t0}), S_{in}(\texttt{t1}))]$$

where the function Mul is defined as follows:

$$Mul(< B_1, X_1 >, < B_2, X_2 >) = \begin{cases} < \text{None}, \{x_1 \times x_2 \mid x_1 \in X_1 \wedge x_2 \in X_2\} > \text{ if } B_1 = B_2 = \text{None} \\ < i_k, \{0\} > \text{ otherwise} \end{cases}$$

Again, in case of at least one operand is "unknown" in S_{in} (i.e., equal to $<$ Empty, $\emptyset >$) then the result will be unknown as well in S_{out}. Similar definitions holds for other arithmetic or logical binary operators.

$\boxed{\text{it may be possible to be more precise for some operators ...}}$

Memory write operation: `STM t0, ,t1` (meaning `MEM[t1] := t0`)
When `t1` is "unknown" in S_{in}, then S_{out} remains equal to S_{in} (the definition of `t1` will be available at a next iteration). Otherwise, two situations may occur:

- if the value $< B_1, X_1 >$ of `t1` in S_{in} denotes *a single memory location*, then this memory location is assigned with `t0` in S_{out} (*strong update*);
- otherwise, for each element `a` of $< B_1, X_1 >$, the current value of `MEM[v]` is merged with the value of `t0` (*weak update*).

Assuming $< B_1, X_1 >= S_{in}(t1)$, this is formalized as follows:

$$S_{out} = \begin{cases} \text{if } B_1 = \text{InitEsp or } B_1 = \text{None and } |X_1| = 1 \text{ then } S_{in}[S_{in}(t1) \to S_{in}(t0)] \\ \text{if } B_1 = \text{InitEsp or } B_1 = \text{None and } |X_1| > 1 \text{ then} \\ \qquad \forall a \in S_{in}(t_1).\ S_{in}[a \to S_{in}(a) \sqcup S_{in}(t0)] \\ \text{if } B_1 = \text{Any then } \forall a \in \mathcal{A}.\ S_{in}[a \to S_{in}(a) \sqcup S_{in}(t0)] \end{cases}$$

Remark:
A better definition could be obtained by considering the set of *"may*

aliases" (according to relation $\cong$) of $< B_1, X_1 >$ in case of weak update. In principle, may aliases are defined with respect to a given position. This means that when an element of S_{in} is assigned with an abstract value of the form $< i, X >$, the location l at which this assignment takes place should stored as well. A simpler alternative is to use the relation $\approx$ (instead of $\cong$) to define may aliases ... $\Box$.

Memory read operation: `LDM t0, ,t1` (meaning `t1 := MEM[t0]`)
When `t0` is "unknown" in S_{in} then S_{out} remains equal to S_{in} (the definition of `t0` will be available at a next iteration). Otherwise, several situations may occur:

- if $B_0 =$ Any, then we are reading from "any" memory location and the content of `t1` in S_{out} will be "any" as well;
- if $S_{in}(t0)$ is "unknown", then the content of `t1` in S_{out} will be "any";
- otherwise, the value of `t1` in S_{out} is the merge of all the values `MEM[a]` for each address s belonging to $S_{in}(t0)$.

Formally:

$$S_{out} = \begin{cases} \text{if } B_0 = \text{Any then } S_{in}[t1 \rightarrow < \text{Any}, \emptyset >] \\ \text{if } B_0 = \text{InitEsp or } B_0 = \text{None and } \exists a \in S_{in}(t0).\ S_{in}(a) =< \text{Empty}, \emptyset > \text{ then} \\ \qquad S_{in}[t1 \rightarrow < \text{Any}, \emptyset >] \\ \text{if } B_0 = \text{InitEsp or } B_0 = \text{None and } \not\exists a \in S_{in}(t0).\ S_{in}(a) =< \text{Empty}, \emptyset > \text{ then} \\ \qquad \forall a \in S_{in}(t0).\ S_{in}[t1 \rightarrow S_{in}(t1) \sqcup S_{in}(a)] \end{cases}$$

All other instructions leave the input state vector unchanged (i.e., $S_{out} = S_{in}$).

2.7 Correctness of the Analysis

This dataflow analysis is *conservative*, it computes a superset of the concrete value of each register and memory location. If we note $Val_a(reg(i_k), i_k)$ the value computed by this algorithm for each This property can be formalized as follows:

$\boxed{\text{A Completer}}$

3 Some experimental results

$\boxed{\text{Add some words on the tool chain used to produce these results}}$

We give the results obtained when executing this value-analysis on two small examples of C programs.

3.1 Conditionnal statement

```
#include <stdio.h>
int main()
{
    int x, y=5, z ;
    if (y<4) {
        x=3; z=4;
    } else {
        x=4; z=3;
    } ;
    y=x+z;
    return z;
}
```

At the assembly level the address of local variables x, y and z are respectively at offsets -8, -12 and -16 from the initial value of esp when entering function main. The value of function S_{out} at the end of the main function is given below:

```
    (u'ebp', '40105601', [0]), (u'esp', 'initESP', [4]),
    ('initESP-12', 'noval', [6, 7, 8]),
    ('initESP-16', 'noval', [3, 4]),
    ('initESP-8', 'noval', [3, 4])
```

As expected, final values of x and z are $\{3, 4\}$, whereas final value of y is in $\{6, 7, 8\}$. Note that esp has been incremented by 4 during the execution (its final value is InitEsp+4), and that ebp has been initalized at (Reil) instruction number 40105601.

3.2 Iterative statement

```
#include <stdio.h>
int main()
{
    int x=0, i, y;
    for (i=0; i<4;i++) {
        y=6; x=x+i;
    }
    return 0;
}
```

At the assembly level the address of local variables x, i and y are respectively at offsets -8, -12 and -16 from the initial value of esp when entering function main. The value of function S_{out} at the end of the main function is given below:

```
(u'ebp', '40105701', [0]), (u'esp', 'initESP', [4]),
('initESP-12', 'anyval', []),
('initESP-16', 'noval', [6]),
('initESP-8', 'anyval', [])
```

The final values of x and i have been "widened" to Any, since they change at each iteration. The final value of y is 6, as expected.

4 Interprocedural Value-Analysis

4.1 Some generalities on interprocedural analysis

Several techniques can be used to perform a data-flow analysis at an interprocedural level. They can be classified according to several criteria [7]:

context sensitivity: A *context-insensitive* analysis does not distinguish between all the potential *calling contexts* (memory and register content, execution stack, etc.) of a procedure. Data-flow information inherited from all calling contexts is merged before analyzing the procedure, and data-flow information synthetized during the procedure execution is propagated back to all callers. This is not the case in a *context-sensitive* analysis.

scope: The *interprocedural effects* (i.e., the mutual influence between a caller and a callee) can be approximated in several ways. In its simplest form an interprocedural analysis could just interpret a procedure call in the most conservative way (the "worst" situation is assumed after a call, e.g., all memory locations are tainted and their value is unknown). A less extreme approach is the so-called *side effect analysis*, where only the callee's influence is computed (the caller's influence being approximated by fixed values). Finaly, a *whole program analysis* aims to take into account influences in both directions (caller $\rightarrow$ callee, and vice-versa).

underlying approach: Two broad approaches are usually considered in interprocedural analysis, the *functionnal approach*, and the *value-based approach.* The functionnal approach proceeds in two steps:

1. the computation of a context-independent and parameterized summary for each individual procedure, representing the effect of the call ;

2. the data-flow analysis of each procedure, where the effect of each call is computed by applying the corresponding summary.

   Note that this approach is not always applicable. Note also that these two steps can be combined when there is no recursive procedure (performing the analysis from the leaves to the top in the call graph, which is DAG in this case). The value based approach does not rely on summaries, but it requires the construction of a *suprer graph* (linking the CFG of each procedures). Then, when a procedure call is encountered during the analysis, the callee is analysed after transmitting all the required information. At the end of this analysis the syntethized information is propagated back to the caller.

In this work we propose to use first a whole-program, context-insensitive, functionnal approach.

Remark (calling conventions).
Dealing with interprocedural analysis at the binary level needs to take into account the *calling convention* used by x86 compilers. These conventions usually fall in four categories:

cdcl: It is the most common one in Unix environment, and it is the one used by `gcc`. All the parameters are transmitted on the stack, and the stack is "cleaned" by the caller after the call.

stdcall: It is the most common one in Windows environment when the number of argument is fixed for each function. All the parameters are transmitted on the stack, and the stack is "cleaned" by the callee at the end of its execution (instruction `RET n`).

fastcall: Same as "stdcall", apart that the first and second parameters are transmitted through registers `ecx` and `edx`.

thiscall: Used for object-oriented languages, same as "stdcall" apart that the first parameter is the current object instance (`this`), and it is transmitted register `ecx`.

In all cases the return value is transmitted through register `eax`.

4.2 Defining procedure summaries

According to our obejctives, the information we need to get from procedure summaries is:

- the number of parameters (to retrieve these parameters in the caller's context) ;
- the function side-effects, namely wat has been tainted/untainted and, what are the values that have been computed/modified for each memory locations.

Indeed, these side effects can be restricted to the memory locations that are relevant from the caller's point of view, namely:

- the callee's return value, i.e., register `eax` (we assume that other registers are left unchanged) ;
- the memory locations those address has been transmitted as an argument to the callee ;
- the global variables.

These side-effects needs to be expressed in terms of memory locations influencing the callee's behaviour, namely:

- the values transmitted as an argument to the callee ;
- the values contained in memory locations those address has been transmitted as an argument to the callee ;
- the global variables.

In an initial (and simplified) version, we may consider function summaries only in terms of *taint analysis*, i.e., what are the side effects of a procedure execution regarding the final taintedness of return value, global variables, and output parameters (memory locations those address has been transmitted as an argument). In this way, the (expensive) value analysis is used only to allow a precise taint analysis inside some specific procedures.

In a more elaborated version it could be possible to deal with side effects in terms of *value analysis* as well, if it happens to be useful . . . .

5 Comparison with related work

5.1 VSA as proposed in [3, 4, 5]

The notations used in this part are taken from [5]. The ones used in [3] slightly differ.

5.1.1 Abstract Memory Model

Address expressions At the binary level, memory locations are accessed:

- either directly, the address is then an immediate value (absolute address);
- or inderictly, the address is then of the form

  $$base + index * scale + offset$$

  where $base$ and $index$ are registers, $scale$ and $offset$ are integer constants.

Memory regions and abstract addresses

- The whole memory area is considered as 3 disjoint *memory regions*: the *globa-region* (a single region for all global data), the *AR-region* (a set of activation regions associated to each procedure [2] and the *malloc-region* (a set of dynamically allocated memory zones).
- Each concrete memory address is represented by a pair (memory-region, offset), considering that:
  - (Global, 0) represents the absolute address 0
  - for a procedure P, (AR_P, 0) represents the value of **esp** when entering procedure P
  - (malloc_L, 0) represents the address returned by a call to `malloc` at program point L.

The memory region $MemRgn$ is defined as

$$MemRgn = \{Global\} \cup Proc \cup AllocMemRgn$$

Abstract locations An abstract location (or *a-loc*) is a representation of a memory zone accessed at program execution (like a variable name at the source level). Each a-loc is identified by an (abstract) address and a size. We can distinguish between:

global a-locs: global variable accesses;

local a-locs: local variable and parameter accesses;

[2]but no more than one AR per procedure, even for recusive ones

heap a-locs: accesses to dynamic memory, note that there is one heap a-loc per heap-region (corresponding to a call to **malloc**) those size might be unknown (hence considered as ∞);

registers: to identify register accesses. Special registers are $Flags$.

5.1.2 Value Set Analysis

The main purpose of VSA is to compute both (and *simultaneously*) a "safe approximation of the set of numeric values or addresses that each a-loc holds at each program point".

value sets Value sets are used to represent set of (numeric) values. The abstract representation chosen in [5] is to use *strided-interval* (SI). An SI of size s is denoted as $s[l, u]$ and, assuming a k-bits memory, represents the following set of concrete values:

$$\{i \in [-2^{k-1}, 2^{k-1} - 1] \mid l \leq i \leq u, i \equiv l \, mod \, s\}$$

The empty SI will be noted $\perp$ and the set of SI is noted $StridedInterval$. A $ValueSet$ is a partial function associating an SI to each memory region:

$$ValueSet : MemRgn \rightarrow StridedInterval$$

Value sets form a lattice (with respect to inclusion, $\sqsubset$). Available operators are:

- meet ($vs_1 \sqcap vs_2$), join ($vs_1 \sqcup vs_2$), widening ($vs_1 \bigtriangledown vs_2$),
- addition with a constant ($vs + c$),
- dereferencing ($*(vs, s)$) meaning the set of a-locs of size s addressed by value-set vs as a pair of sets (F, P), for "fully-accessed" and "partially accessed" a-locs.

abstract environment The main purpose of VSA is to associate an "abstract environment" ($AbsEnv$) to each program location, where an $AbsEnv$ is a mapping associating a value-set to each a-loc. $a - locs[R]$ denotes the set of a-locs belonging to a memory region R, and $AlocEnv[R]$ denotes the set of values associated each a-loc of R:

$$AlocEnv[R] : a - locs[R] \rightarrow ValueSet$$

An *AbsEnv* is then defined as the product of the following mapings:

$$\begin{aligned} & (register \to ValueSet \cup \{true, false, maybe\}^3) \\ \times\ & (\{Global\} \to AlocEnv[Global]) \\ \times\ & (Proc \to AlocEnv[Proc]) \\ \times\ & (AllocMemRgn \to AlocEnv[AllocMemRgn]) \end{aligned}$$

Note that maping for *Proc* and *AllocMemRgn* are partial mappings, since a procedure may be not active, or a `malloc` instruction not executed yet.

5.1.3 Intraprocedural VSA

It is a data-flow analysis those objective is to compute the *AbsEnv* associated to each node of a program CFG. It is based on a (classical) working-list forward least fix-point computation algorithm, based on a transfer function associated to each instruction. Specific features are:

- *Conditions* are taken into account. In the CFG, both instructions and conditions are associated to edges (and not to vertices). The transfer fuction associated to a condition allows to restrict (using operator $\sqcap$) the current *AbsEnv* wrt this condition. Note that retrieving the "high level" predicate (e.g, $x \geq 5$) associated to a branch or jump instruction from the binary may not be trivial.
- Widening is applied at at least one edge of every cycle in the CFG (" la Bourdoncle"). The choice of this edge is explained in a specific paper.
- Due the fact that size and overlapping of a-locs is taken into account, the transfer function associated to memory reads and writes is more complex. In particular:
  - for each memory write, there's a distinction between two main cases:
    - if the write address corresponds to a single fully-accessed a-locs which is neither in a recursive procedure nor in the heap, then this a-loc is *strongly updated* with the new value set ;
    - otherwise, each fully-accessed a-locs is *weakly updated* (the new value set is $\sqcup$-ned with the current one), and each partally-accesed a-locs is set to "top" (any possible value).
  - for each memory read, the same distinction occurs:
    - if there's no partially-accessed a-locs in the read address, then value read is a $\sqcup$ of all the fully-accessed a-locs ;
    - otherwise the value read is "top".

Two versions of intraprocedural VSA have been proposed.

version 1 [3]. The set of a-locs considered during the VSA is computed *a priori* using the so-called "semi-naive algorithm" implemented by IDAPro. This algorithm looks for explicit (i.e., based on syntactic patterns) memory accesses:

- either through an offset relative to `ebp` or `esp` (e.g., `[ebp + ...]`, `[esp + ...]`)
- or through absolute addresses in case of global variables (e.g., `[190098]`)

An a-loc is then a memory region laying between two such addresses (it has a given size). The value sets stored in the memory locations identified (only) by this fixed set of a-locs are computed during the VSA.

This approach suffers from some drawbacks:

- "indirect" memory accesses are ignored (e.g., `[eax]`, `[eax + ...]`)
- the memory layout considered is sometimes too coarse, ignoring the underlying data structure (array, records). As a consequence, VSA results can be (too) over-approximated: whenever an a-loc representing an array slice is accessed, the slice is considered as a whole (even if only one cell is accessed). This is also the case when computing dependencies between a-locs.

version 2 [8, 5]. To improve version 1, the idea is to proceed in an iterative way (following an abstraction-refinement scheme). It relies on a combination between VSA and ASI (Aggregate Structure Identification). ASI allows to identify the structure od data aggregates used in a program trough the data patterns used to access them:

Ex: accessing every four bytes at offets -40, -32, ... -8 means accessing a 4-bytes field of an array of 8-bytes structures.

The principle is the following:

- a first VSA is performed from the a-locs provided by IDAPro ;
- data-access patterns are generated from the results of this VSA
- apply ASI to refine the initial set of a-locs
- run VSA again on this new set
- etc., until no more a-locs are discovered.

Rk: it is not clear what happens when the value of a register used for a memory accessed is found equal to "ANY" by the 1st round of VSA ... (probably no possible ASI from this information)

The underlying problem is to find a good trade-off between accuracy and efficiency (number of a-locs discoverd wrt execution time). Another problem is that some operations (like memcpy, used for instance to initialise dynamicaly allocated memory) "breaks" the whole memory zone into a 1-byte array. To solve these two problems some indications can be given by the user.

5.1.4 Interprocedural VSA

Two versions are described in [5], a context-insensitive and a context-sensitive one.

The context-insensitive solution It follows a *value-based* approach, which does not need to compute procedure summaries (but it means that each procedure will be entirely analyzed at each iteration). The basic idea is to consider a so-called *super-graph*, which extends the notion of CFG to the whole program, and to perform a intra-procedural like data-flow analysis from this super-graph. The main difficulty is to correctly handle *parameter binding* (i.e., relating formal parameters to actual ones). Details of this approach are given below.

Super-graph. Each call instruction is splitted into two nodes, named `call` and `end-call`. To express the control flows corresponding to procedure calls, edges are added between each `call` nodes and the corresponding `enter` node (of the procedure being called), and similarly between each `exit` node and all possible corresponding `end-call` nodes (of each potential callers). Note that in [5] (direct) edges are also added between each corresponding `call` and `end-call` nodes, with the identity transfer function[4]

Parameter bindings. Special operations are performed when dealing with `call` $\rightarrow$ `enter` and `exit` $\rightarrow$ `end-call` edges. In the former case, the content of `esp` is initialized at offset 0 (base offset for the callee), and each a-loc defined in the callee is initilialized with its value as computed in the caller's environment. This is performed for arguments, registers and local variables (???). In the latter case, `esp` is restored, and each a-loc defined in the caller is assigned with its value as computed in the callee's environment. Note that weak updates are used in case of recursive prpcedures or memory overlapings.

[4]to take into account the case where *the propagation quiesces in the callee.*

Interprocedural VSA. It is similar to a classical intraprocedural analysis performed on a CFG, apart that the extra-edges are taken into account (by applying the transfer functions described above).

The context-sensitive solution $\boxed{\text{to be completed}}$

5.1.5 Main differences with our work

Clearly the main concern of this work is to staticaly compute a *safe* over-approximation of the possible value of each memory location accessed from any (*arbitrary*) binary program. Our objective is less ambitious: we need a VSA only for making the taint analysis more precise, by keeping track of the addresses used in memory read/write operations whenever it's possible, while preserving the scalability of our analysis. This leads to some differences:

- taking into account size and overlapings of memory zones ;
- being more or less "dynamic" (?): a single abstract address for each memory chunk accessed (a-loc), whereas we start a "new" a-loc whenever we are lost in the course of a procedure ...

$\boxed{\text{This latest point needs to be clarified, some refinements/extensions are given in [5] \dots}}$

5.2 Statically-directed dynamic automated test generation [2]

$\boxed{\text{to be completed}}$

6 What is missing in this current version

- summarize the general assumption we are considering: expected structure of the stack, no recursive procedures (yet ?), symbolic jumps handled by IDAPro (no more), etc.
- size of the memory transfers to be taken into account
- (scalable) interprocedural analysis
- taint analysis

- optimization by taking into account only registers and statements involved in address computations (?)
- use of *widening* and *narrowing* operators to make the fix-point computation more efficient still keeping accurate enough results ...
- etc.

$\boxed{\text{to be completed ..}}$

References

[1] Wolfram Amme, Peter Braun, Eberhard Zehendner, and François Thomasset. Data Dependence Analysis of Assembly Code. Research Report RR-3764, INRIA, 1999.

[2] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 12–22, New York, NY, USA, 2011. ACM.

[3] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *In CC*, pages 5–23. Springer-Verlag, 2004.

[4] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32:23:1–23:84, August 2010.

[5] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32:23:1–23:84, August 2010.

[6] Saumya Debray and Robert Muth. Alias analysis of executable code. In *In POPL*, pages 12–24, 1998.

[7] U. Khedker, A. Sanyal, and B. Karkare. *Data flow analysis: theory and practice.* Taylor and Francis, 2009.

[8] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. *ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation*, page 100, 2006.

[9] Am Wolfram, Peter Braun, François Thomasset, and Eberhard Zehendner. Data dependence analysis of assembly code. *Int. J. Parallel Program.*, 28:431–467, October 2000.