Side-Effect Analysis of Assembly Code

Andrea Flexeder, Michael Petter and Helmut Seidl

Technische Universität München, Boltzmannstrasse 3, 85748 Garching, Germany,
`{flexeder,seidl, petter}@cs.tum.edu`,
WWW home page: `http://www2.cs.tum.edu/~{flexeder,seidl,petter}`

Abstract. In this paper we present a light-weight interprocedural side-effect analysis on assembly code. We represent the modifying potential of a procedure f by classifying all write accesses, occurring within f, relative to the parameter registers. In particular our approach is the first to accurately handle reference parameters. We demonstrate the usefulness of this approach by integrating this analysis into our assembly analyser and provide an evaluation of the precision of our approach. Approximately 50 per cent of all procedures can be statically shown to have side-effects.

1 Introduction

The prevalent binary analysis frameworks, e.g. CodeSurfer/$x86$ [23], BitBlaze [28] or Jakstab [16] reach their limits when analysing large assemblies. All of these approaches rely on the (approximate) call-string approach [27] and thus suffer from imprecision when limiting call-string length. Analysing binaries consisting of more than $700,000$ assembler instructions with these call-string based approaches will be prohibitively expensive to be applied in practise. On the one hand, analyses can treat procedure calls conservatively, i.e. a procedure call invalidates the values of *all* the local variables whenever a pointer to its stack frame escapes to another execution context. On the other hand, analyses can make standard assumptions about a procedure call, e.g. that a procedure does only write to its own stack frame or to the heap. While these assumptions hold for restricted program classes, e.g. safety-critical software, this is not the case for general-purpose software as our experiments show. Therefore, we propose a *side-effect analysis* which computes a tight approximation of write accesses of procedures to the stack. In contrast to the existing binary analysis frameworks, we propose a light-weight form of functional approach [27] to interprocedural analysis which still produces useful results. In our framework, the effect of a procedure is represented by all possible parameter register relative write accesses. The following example illustrates the side-effect of a procedure which is passed a local address.

Example 1. In the following C code fragment, a pointer to array `c` is passed as parameter to procedure `f`. Within `f`, the first four elements of the array are set to zero. The corresponding PPC [14] assembly is given to the right. In PPC assembler the stack pointer is given by register `r1`. The array access is realised by passing a stack address, i.e., address `r1+12`, in register `r3` to procedure `f` (cf. instruction `0x5C`). After the call to `f` within `main`, a conservative analysis of `main` therefore must assume that all local variables of `main` may have been modified by the procedure call. Thus, all information about the values of local variables after the procedure call `f()` is lost.

```
f(int a[]){
  int j;
  for(j=0;j<4;j++)
    a[j] = 0;
}

main(){
  int c[4];
  int b = 13;
  f(c);
}
```

```
//main(){...
//int b = 13;
50:  li    r2,13
54:  stw   r2,8(r1)
//f(c);
58:  addi  r2,r1,12
5C:  mr    r3,r2
60:  bl    0x00
//}...
```

```
//f(int a[]){
00:  stwu    r1,-32(r1)
04:  stw     r3,24(r1)
//...

//a[j] = 0;
14:  lwz     r2,8(r1)
18:  mulli   r2,r2,4
1C:  mr      r9,r2
20:  lwz     r2,24(r1)
24:  add     r9,r9,r2
28:  li      r2,0
2C:  stw     r2,0(r9)
//...
```

In order to retain the value of the local variable `b` in `main`, we have to determine the side-effect of `f`. The side-effect identifies all stack locations whose value may be modified through the procedure call and thus must be invalidated after instruction `0x60`. Accordingly, we inspect the memory write accesses occurring in procedure `f`. The only memory write access in `f` happens at instruction `0x2C` and modifies the memory cells addressed by the expressions `{r3,r3+4,r3+8,r3+12}`. This set describes the *modifying potential* of procedure `f`. By matching actual to formal parameters, we conclude that within procedure `main` after the call to `f`, the memory locations `{r1+12,r1+16,r1+20,r1+24}` of `main` are modified, while local `b` (i.e., the memory cell with address `r1+8`) remains untouched by `f()`. □

Related Work Side-effect analysis has intensively been studied for high-level languages, e.g., the purity analysis for Java programs [25], the side-effect analysis for C/C++ programs [4,18], or the context-sensitive pointer analysis for C programs [31,11]. The approach of Cooper et al. [5] relies on graph algorithms for solving the alias-free side-effect analysis problem introduced by Banning [3] in a flow-insensitive manner. All these techniques, however, are not directly applicable to low-level code where there is no clear distinction between an integer and an address.

For the analysis of low-level code, Debray et al. present an interprocedural, flow-sensitive pointer alias analysis of $x86$ executables, which, however, is context-insensitive [8]. They abstract the value of each register by an *address descriptor*, i.e., a set of possible congruence values with respect to a program instruction. They make no distinction between two addresses which have the same lower-order k bits. Since they do not track any memory content they suffer from information loss, whenever a register is assigned a value from memory. Moreover if different definitions of the same register reach the same join point, then this register is assumed to take any value. A context-sensitive low-level points-to analysis is presented in [15]. This approach is only partially flow-sensitive: the values of registers are handled flow-sensitively using SSA form, while memory locations are treated flow-insensitively (tracking only a single points-to set for each memory location). The notion of *UIV*s (unknown initial values) is introduced in order to represent all those memory locations that are accessible by the procedure but do not belong to the current stack frame of the procedure or to the stack frames of its callees. Their aim is to improve on compiler optimisations, such as e.g. load and store reorderings, and thus a crude treatment of local memory is sufficient. In contrast, we track local memory locations context-sensitively as well. The most advanced approach in the area of analysis of assembly code is provided by Reps et al. [23]. They propose a flow-sensitive framework for analysing memory accesses in $x86$ executables. For dealing with procedures, they rely on the call-string approach (CSA)[27]. In order overcome the context-insensitivity of CSA-0 and the impracticability of increasing the length of call-strings they apply techniques from Cooper et al. [5] in order to determine the set of memory locations that may be modified by each procedure. This information is used in order to improve on the precision when combining the information from different call sites. In contrast, our algorithm adheres to the functional approach of program analysis and thus does not suffer from the limitations of call-strings of bounded length.

A stack analysis for $x86$ executables is addressed in [19]. There, the authors aim at verifying that a function leaves the stack in its original state after the function returns. In order to identify such *well-behaving* functions, use-depth and kill-depth analyses are introduced. By means of use-depth information they estimate the maximal stack height from which a function may read a value, while kill-depth information tells the maximal height of the runtime stack to which the function may write a value. While addressing a related problem, the applied techniques are different. In particular, our approach does not suffer from a loss of precision when dealing with recursive procedures.

Contributions We propose a light-weight side-effect analysis based on the functional approach to interprocedural analysis which

- provides context-sensitive side-effects by computing an overapproximation of the set of register-relative memory locations which are modified by a procedure call.
- allows analysing large binaries with more than $700,000$ assembler instructions reasonably fast and also provides significant side-effects.
- can be used to lose less information after procedure calls in sound intraprocedural analyses.

Overview In the next Section 2 we present the concrete semantics for our side-effect analysis. In Section 3 we first describe how to embed the side-effect of a procedure into an intraprocedural analysis, then present how to compute the modifying potential of a procedure and finally provide a correctness proof of our approach. Additionally we describe the implementation of the side-effect analysis in our assembly analyser and evaluate its results in Section 4, before we conclude in Section 5.

2 The Concrete Semantics

For the analysis, we assume that programs are given as a collection of procedures q where each q is represented by a finite control flow graph G_q. The control flow graph of a given assembly can be extracted via methods provided by, e.g., the program analysis frameworks [12,24,17]. Every control flow graph G_q consists of:

- a finite set N_q of *program points* of the procedure q,
- a finite set $E_q \subseteq (N_q \times \mathit{Label} \times N_q)$ of *edges*, where *Label* denotes a program instruction,
- a unique entry point $s_q \in N_q$ for procedure q,
- a unique exit point $r_q \in N_q$ for procedure q and
- a designated procedure $main$ where program execution always starts.

Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ denote the set of registers the assembly operates on, with k the number of registers. We assume that all the registers $\mathbf{X}$ are global and thus possibly serve for passing parameters.

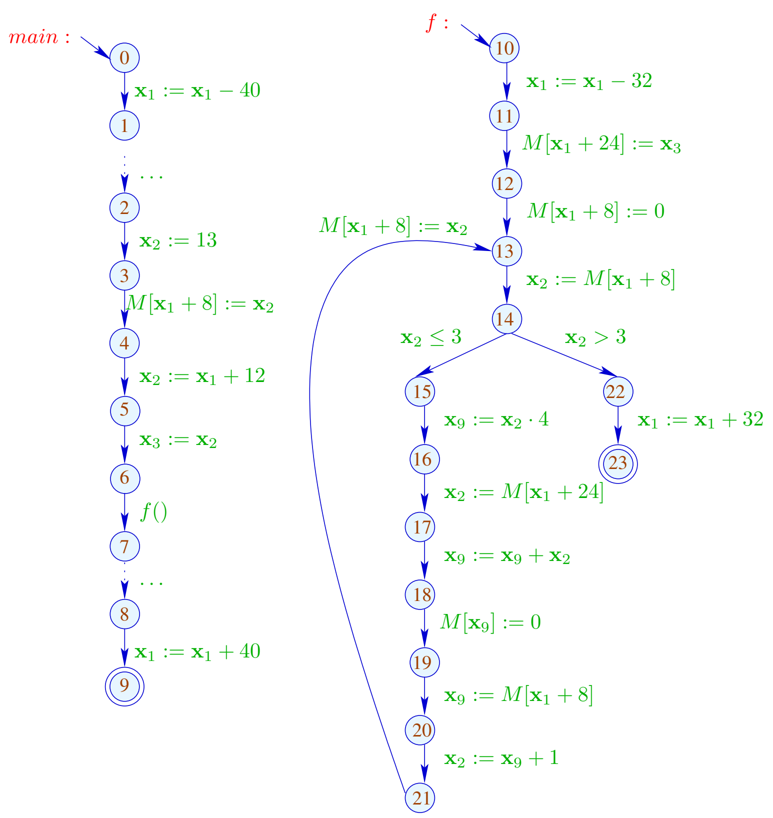
The concrete effect of every processor instruction within our analysis is modelled by a sequence of the following constructs:

- stm: assignment statements and memory access instructions,
- guards and
- $q()$: procedure calls.

Each edge in the control flow graph therefore is annotated by one of these constructs. For the analysis, we consider assignments of the form $\mathbf{x}_i := t$ ($\mathbf{x}_i$ a register, t an expression) and guards providing comparisons of the values of two registers, i.e.,$\mathbf{x}_j \bowtie \mathbf{x}_k$, and the value of a register with a constant, i.e. $\mathbf{x}_j \bowtie c$, for every comparison operator $\bowtie$. Furthermore, we use non-deterministic assignments of the form $\mathbf{x}_i :=?$ to represent instructions possibly modifying register $\mathbf{x}_i$ in an unknown way. Non-deterministic assignments represent a safe description of those instructions which are not handled precisely, but have an impact on the values of registers, as for instance bit operations. Memory access instructions are of the form $\mathbf{x}_i := M[t]$ or $M[t] := \mathbf{x}_i$ ($\mathbf{x}_i$ a register, t an expression). For simplicity, here we only consider memory access instructions applied to words, i.e., 4 bytes of memory simultaneously. That means that we exclude instructions operating on multiple registers simultaneously. The formalism can be extended, though, to deal with variable-length memory access instructions and multi-register arithmetic as well. This format subsumes assembler programs in three-address form as is provided by some processors, e.g., PowerPC[14], or by a transformation into a low-level intermediate representation, e.g., REIL [10].

When dealing with procedure calls, we distinguish procedure-global from procedure-local memory locations. A dedicated register of the processor serves as *stack pointer*, i.e., holds the top-of-stack for the current procedure. In our formalisation, this register is given by $\mathbf{x}_1$. Accordingly, the single edge starting at the entry point of the control flow graph for a procedure q is assumed to be annotated with an instruction $\mathbf{x}_1 := \mathbf{x}_1 - c_q$ for some constant c_q. This instruction allocates the local stack space for the current invocation of q. Likewise, the single edge reaching the exit point of the control flow graph for q is annotated with the instruction $\mathbf{x}_1 := \mathbf{x}_1 + c_q$ for the same constant c_q. This instruction deallocates the local stack space. For simplicity, we rule out intermediate increments or decrements of the stack pointer, and thus do not consider *dynamic* stack allocation here. An approach to deal with dynamic stack allocation is sketched in [13]. Figure 1 illustrates our control flow representation of Example 1.

For the concrete semantics, the memory $\mathbf{S}$ of the program is divided into the disjoint addresses spaces $\mathbf{S}_L$ for the *stack* or local memory, and $\mathbf{S}_G$ for global memory. The set of global addresses is given by $\mathbf{S}_G = \{(G, z) \mid z \in \mathbb{Z}\}$, and the set of stack



$main:$
0
$\mathbf{x}_1 := \mathbf{x}_1 - 40$
1
$\dots$
2
$\mathbf{x}_2 := 13$
3
$M[\mathbf{x}_1 + 8] := \mathbf{x}_2$
4
$\mathbf{x}_2 := \mathbf{x}_1 + 12$
5
$\mathbf{x}_3 := \mathbf{x}_2$
6
$f()$
7
$\dots$
8
$\mathbf{x}_1 := \mathbf{x}_1 + 40$
9
$f:$
10
$\mathbf{x}_1 := \mathbf{x}_1 - 32$
11
$M[\mathbf{x}_1 + 24] := \mathbf{x}_3$
12
$M[\mathbf{x}_1 + 8] := 0$
$M[\mathbf{x}_1 + 8] := \mathbf{x}_2$
13
$\mathbf{x}_2 := M[\mathbf{x}_1 + 8]$
14
$\mathbf{x}_2 \leq 3$
$\mathbf{x}_2 > 3$
15
22
$\mathbf{x}_9 := \mathbf{x}_2 \cdot 4$
$\mathbf{x}_1 := \mathbf{x}_1 + 32$
16
23
$\mathbf{x}_2 := M[\mathbf{x}_1 + 24]$
17
$\mathbf{x}_9 := \mathbf{x}_9 + \mathbf{x}_2$
18
$M[\mathbf{x}_9] := 0$
19
$\mathbf{x}_9 := M[\mathbf{x}_1 + 8]$
20
$\mathbf{x}_2 := \mathbf{x}_9 + 1$
21

Fig. 1. Control flow representation of Example 1.

addresses is given by $\mathbf{S}_L = \{(L, z) \mid z \in \mathbb{Z}\}$. A program state γ then is given by a triple $\gamma = \langle \rho, f, \lambda \rangle$ where

- $\rho : \mathbf{X} \to \mathbf{S}$ provides the contents of registers.
- $f = (x_r, x_{r-1}, \dots, x_0)$ for $x_r < x_{r-1} < \dots < x_0$ is the *frame structure* of the current stack where $x_0 = \mathsf{Max} + 1$ and x_r equals the least address on the current stack. Here Max is the maximal height the stack can grow. Thus in particular, $x_r = \rho(\mathbf{x}_1)$. Note that here the stack grows *downward* from numerically higher addresses towards zero.
- λ provides the contents of memory locations. Thus, λ assigns values to the set $\mathbf{S}_G$ of global addresses as well as to the current set of local addresses. The values of global addresses are restricted to be global addresses only. The set of allowed local addresses is restricted to the set $\mathbf{S}_L(x_r) = \{(L, z) \mid x_r \leq z < x_0\}$. Their values are restricted to elements from the set $\mathbf{S}_G \cup \mathbf{S}_L(x_r)$ only.

In order to obtain a uniform representation, concrete integer values are embedded into the *global* address space, i.e., the plain value 5 as well as the global address 5 are represented by the pair $(G, 5)$. Remark that the labels L and G allow to inherently distinguish between local and global address space.

We consider address arithmetic w.r.t. a given frame structure $f = (x_r, x_{r-1}, \dots, x_0)$. We restrict the arithmetic operations on local addresses such that starting from a local address from a stack frame $[x_i, x_{i-1} - 1]$, arithmetic is only legal if it produces a local address within the range $[x_i, x_0]$. The interval $[x_i, x_0]$ represents the set of all currently valid local addresses. Accordingly, the arithmetic operations of addition, subtraction, multiplication and boolean comparisons on elements from $\{L, G\} \times \mathbb{Z}$ w.r.t. a given frame structure $f = (x_r, x_{r-1}, \dots, x_0)$ are defined by:

$$
\begin{aligned}
&(L, c_1) +_f (L, c_2) = \mathsf{undefined}\\
&(G, c_1) +_f (G, c_2) = (G, c_1 + c_2)\\
&(L, c_1) +_f (G, c_2) = (G, c_2) +_f (L, c_1) = \begin{cases} (L, c_1 + c_2) & \text{if } x_r \le c_1 \Rightarrow x_r \le c_1 + c_2\\ \mathsf{undefined} & \text{otherwise}\end{cases}\\
&(L, c_1) -_f (L, c_2) = (G, c_1 - c_2)\\
&(G, c_1) -_f (G, c_2) = (G, c_1 - c_2)\\
&(L, c_1) -_f (G, c_2) = \begin{cases} (L, c_1 - c_2) & \text{if } x_r \le c_1 \implies x_r \le c_1 - c_2\\ \mathsf{undefined} & \text{otherwise}\end{cases}\\
&(G, c_1) -_f (L, c_2) = \mathsf{undefined}\\
&(G, c_1) \cdot_f (G, c_2) = (G, c_1 \cdot c_2)\\
&(L, c_1) \cdot_f (G, c_2) = (G, c_2) \cdot_f (L, c_1) = \mathsf{undefined}\\
&(L, c_1) \cdot_f (L, c_2) = \mathsf{undefined}\\
&(L, c_1) \bowtie (L, c_2) \begin{cases} \mathsf{true} & \text{if } c_1 \bowtie c_2\\ \mathsf{false} & \text{otherwise}\end{cases}\\
&(G, c_1) \bowtie (G, c_2) \begin{cases} \mathsf{true} & \text{if } c_1 \bowtie c_2\\ \mathsf{false} & \text{otherwise}\end{cases}\\
&(G, c_1) \bowtie (L, c_2) = \mathsf{undefined}\\
&(L, c_2) \bowtie (G, c_1) = \mathsf{undefined}
\end{aligned}
$$

for every comparison operator $\bowtie$. If an undefined value occurs, we assume that an exception is thrown and the program execution is aborted. For the analysis, we only consider non-aborting program executions and flag warnings if an abortion cannot be excluded. Statements and guards induce transformations of (sets of) states. A single processor instruction s on a given program state $\langle \rho, f, \lambda \rangle$ returns a set of program states which is defined by:

$$
\begin{aligned}
[\![\mathbf{x}_i := t]\!]\langle \rho, f, \lambda \rangle &= \{\langle \rho \oplus \{\mathbf{x}_i \mapsto [\![t]\!](\rho, f)\}, f, \lambda \rangle\}\\
[\![\mathbf{x}_i := ?]\!]\langle \rho, f, \lambda \rangle &= \{\langle \rho \oplus \{\mathbf{x}_i \mapsto a\}, f, \lambda \rangle \mid a \in \mathbf{S}_G\}\\
[\![\mathbf{x}_i := M[t]]\!]\langle \rho, f, \lambda \rangle &= \{\langle \rho \oplus \{\mathbf{x}_i \mapsto \lambda([\![t]\!](\rho, f))\}, f, \lambda \rangle\}\\
[\![M[t] := \mathbf{x}_i]\!]\langle \rho, f, \lambda \rangle &= \{\langle \rho, f, \lambda \oplus \{\lambda([\![t]\!](\rho, f)) \mapsto \rho(\mathbf{x}_i)\}\rangle\}\\
[\![\mathbf{x}_j \bowtie \mathbf{x}_k]\!]\langle \rho, f, \lambda \rangle &= \{\langle \rho, f, \lambda \rangle \mid \rho(\mathbf{x}_j) \bowtie \rho(\mathbf{x}_k) = \mathsf{true}\}\\
[\![\mathbf{x}_j \bowtie c]\!]\langle \rho, f, \lambda \rangle &= \{\langle \rho, f, \lambda \rangle \mid \rho(\mathbf{x}_j) \bowtie (G, c) = \mathsf{true}\}
\end{aligned}
$$

with $\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k \in \mathbf{X}$ where $i \neq 1$, t an expression and arbitrary $c \in \mathbb{Z}$. Here, the operator $\oplus$ adds new argument/value pairs to a function. Moreover, the evaluation function $[\![t]\!](\rho, f)$ takes an expression t and returns the value of t in the context of register assignment ρ w.r.t. a given frame structure f.

$$[\![t]\!](\rho, f) = \begin{cases} [\![t_1]\!](\rho, f) \,\Box_f\, [\![t_2]\!](\rho, f) & \text{if } t = t_1 \,\Box\, t_2 \\ \rho(\mathbf{x}_i) & \text{if } t = \mathbf{x}_i \\ (G, c) & \text{if } t = c \end{cases}$$

with $\Box = \{+, -, \cdot\}$ and $\Box_f = \{+_f, -_f, \cdot_f\}$, respectively. Consider, e.g., a memory access $\mathbf{x}_i := M[t]$. Then for every state $\langle \rho, f, \lambda \rangle$ the value a of expression t is determined. Given that a is a valid memory address of λ, the content $\lambda(a)$ of the memory location a is assigned to register $\mathbf{x}_i$.

Next, we describe the effect of a procedure call $q()$. According to our convention, a stack pointer decrement instruction reserves a stack region for the local variables of the procedure. For the concrete semantics, this means that for a procedure call $q()$ the stack is extended by the stack frame of q. According to our assumptions, the only assignments to variable $\mathbf{x}_1$ are of the form $\mathbf{x}_1 := \mathbf{x}_1 + c$ for some $c \in \mathbb{Z}$ at the first and last control flow edge of control flow graphs only. There, these assignments allocate or deallocate the local stack frame for the current instance of the procedure. The newly allocated memory cells are uninitialised and therefore have arbitrary values. Accordingly, we define for $c > 0$ and $f = (x_r, \ldots, x_0)$:

$$\begin{aligned}[\![\mathbf{x}_1 := \mathbf{x}_1 - c]\!]\langle \rho, f, \lambda \rangle = \{ \langle & \rho \oplus \{\mathbf{x}_1 \mapsto \rho(\mathbf{x}_1) -_f (G, c)\}, \, (x_r - c, x_r, \ldots, x_0), \\ & \lambda \oplus \{(L, z) \mapsto a \mid z \in [x_r - c, x_r); a \in \mathbf{S}_G\} \rangle \mid x_r \geq c \}\end{aligned}$$

After execution of the body of the procedure, the current stack frame is deallocated. Assume that $f = (x_{r+1}, x_r, \ldots, x_0)$ and $c > 0$. Then the effect of the last instruction of the procedure epilogue which increments the stack pointer again, for state $\langle \rho, f, \lambda \rangle$ is given by:

$$\begin{aligned}[\![\mathbf{x}_1 := \mathbf{x}_1 + c]\!]\langle \rho, f, \lambda \rangle = \{ & \langle \rho \oplus \{\mathbf{x}_1 \mapsto \rho(\mathbf{x}_1) +_f (G, c)\}, (x_r, \ldots, x_0), \lambda_{|\mathbf{S}_G \cup \mathbf{S}_L(x_r)} \rangle \\ & \mid x_r = x_{r+1} + c \}\end{aligned}$$

After executing the last instruction of procedure q, the stack frame of q has been popped. In particular, the stack pointer again points to the top of the stack. With $\lambda_{|M}$ we denote the restriction of λ to the domain M.

Any procedure q transforms the set of program states before the procedure call to the set of program states after the procedure call to q. Following the approach, e.g., of [21,22], we characterise the effect of a procedure p by means of a constraint system E over transformers operating on program states Γ:

$$\begin{array}{lll} E(s_p) & \supseteq \mathsf{Id} & s_p \text{ start point of procedure } p \\ E(v) & \supseteq E(r_q) \circ E(u) & (u, q(), v) \text{ a call edge} \\ E(v) & \supseteq [\![s]\!] \circ E(u) & (u, s, v) \text{ with } s \in \mathsf{stm} \cup \mathsf{guards} \end{array}$$

with r_q the return point of procedure q. Here, Id denotes the singleton mapping defined by $\mathsf{Id}(y) = \{y\}$ and $\circ$ denotes the composition of transformations of type $\Gamma \to 2^{\Gamma}$. This composition is defined by:

$$(f \circ g)(y) = \bigcup \{f(y') \mid y' \in g(y)\}.$$

The effect of a procedure q is given by the effect accumulated at its return point $E(r_q)$.

The set of *attained program states* when reaching program point u is given by the least solution of the following system of inequations R. The effect of a call to procedure q is given by the application of $E(r_q)$ to the set of program states, valid immediately before the procedure call.

$$\begin{array}{lll} R(s_{main}) \supseteq \mathit{\Gamma}_0 & & \\ R(v) \supseteq [\![s]\!]\,(R(u)) & & (u, s, v) \text{ with } s \in \mathsf{stm} \cup \mathsf{guards} \\ R(v) \supseteq E(r_q)(R(u)) & & (u, q(), v) \text{ a call edge} \end{array}$$

Here, $\mathit{\Gamma}_0$ denotes the set of start states of the program. At program start, the stack is empty. This means that the stack pointer $\mathbf{x}_1$ has the value $(L, \mathsf{Max} + 1)$. Thus,

$$\mathit{\Gamma}_0 = \{\langle \rho, (\mathsf{Max} + 1), \lambda\rangle \mid \rho : \mathbf{X} \to \mathbf{S}_G \mid \rho(\mathbf{x}_1) = (L, \mathsf{Max} + 1), \lambda : \mathbf{S}_G \to \mathbf{S}_G\}$$

Moreover, *application* of a function $T : \mathit{\Gamma} \to 2^{\mathit{\Gamma}}$ to a set $Y \subseteq \mathit{\Gamma}$ is defined by:

$$T(Y) = \bigcup\{T(y) \mid y \in Y\}$$

Since the right-hand sides in both constraint system E and constraint system R denote monotonic functions, these systems of inequations have unique least solutions.

3 Analysis of Side-Effects

This section consists of three parts: Firstly, we describe how to embed the side-effect information of a procedure into any intraprocedural value analysis. Secondly, we present an interprocedural analysis which computes the modifying potential of each procedure. And, thirdly, we prove the correctness of our side-effect analysis.

3.1 Effect Integration

Our goal is to determine for every procedure q its *modifying* potential, i.e., the set of local memory cells whose contents are possibly modified during an invocation of q. For that, we consider the register values at a procedure call as the arguments of the procedure. The modifying potential $[\![q]\!]^\natural$ therefore, is represented by two components (X, M) where:

- $X \subseteq \mathbf{X}$ is a subset of registers whose values after the call are equal to their values before the call. This set should always contain $\mathbf{x}_1$.
- $M : \mathbf{X} \to 2^{\mathbb{Z}}$ is a mapping which for each register $\mathbf{x}_i$ provides a (super)set of all $\mathbf{x}_i$-relative write accesses to the local memory. In a concrete implementation, the analysis may compute with particular sets of offsets only, such as intervals [20,6] or strided intervals [1].

Given the modifying potential $[\![q]\!]^\natural$ of all procedures q in the program, a value analysis can be constructed which determines for every program point, for all registers and local memory locations a superset of their respective values. For that, we are interested in two kinds of values:

- *absolute values*, i.e., potential addresses in the global memory. These are represented as $(\mathbf{x}_0, z)$. Register $\mathbf{x}_0$ is used for accessing the segment of global memory. In our framework, we assume $\mathbf{x}_0$ to be hard-wired to the value $(G, 0)$.
- *stack pointer offsets*, i.e., local memory locations which are addressed relative to $\mathbf{x}_1$. These are represented as $(\mathbf{x}_1, z)$.

Thus, we consider the value domain $\mathbb{V} = 2^{\{\mathbf{x}_1, \mathbf{x}_0\} \times \mathbb{Z}}$ where the greatest element $\top$ is given by the set $\{\mathbf{x}_1, \mathbf{x}_0\} \times \mathbb{Z}$.
Let $\mathbf{S}^\sharp{}_q := [0, c_q]$ denote the set of all procedure-local memory cells of procedure q where the constant c_q is provided by the initial control flow edge of procedure q.
An abstract program state w.r.t. procedure q is then given by the triple $\langle \rho^\sharp, c_q, \lambda^\sharp \rangle$, with:

- $\rho^\sharp : \mathbf{X} \to \mathbb{V}$ which assigns the value $\{(\mathbf{x}_1, 0)\}$ to the stack pointer $\mathbf{x}_1$, and to each register $\mathbf{x}_i \in \mathbf{X}$ different from $\mathbf{x}_1$ a set of its possible values.
- c_q denotes the size of the stack frame of procedure q. This constant is provided by the single edge starting at the entry point of q.
- $\lambda^\sharp : \mathbf{S}^\sharp{}_q \to \mathbb{V}$ which assigns every local memory location from the stack frame of q a set of its possible values.

Thus, our analysis determines for every program point u of a procedure q a set of program states $\Gamma^\sharp$ of the form $\langle \rho^\sharp, c_q, \lambda^\sharp \rangle$. Again, $R^\sharp(u)$ is defined as the least solution of the following constraint system:

$$\begin{array}{lll}
[\mathrm{R}^\sharp 0]\ R^\sharp(s_q) \sqsupseteq \Gamma^\sharp_0 & & s_q \text{ start point of procedure } q \\
[\mathrm{R}^\sharp 1]\ R^\sharp(v) & \sqsupseteq [\![s]\!]^\sharp(R^\sharp(u)) & (u, s, v) \text{ with } s \in \mathsf{stm} \cup \mathsf{guards} \\
[\mathrm{R}^\sharp 2]\ R^\sharp(v) & \sqsupseteq [\![p]\!]^\natural \ @^\sharp\ (R^\sharp(u)) & (u, p(), v)
\end{array}$$

where $\Gamma^\sharp_0$ sets $\mathbf{x}_1$ to the set $\{(\mathbf{x}_1, 0)\}$ and all other registers and local memory locations to the full set $\{\mathbf{x}_0, \mathbf{x}_1\} \times \mathbb{Z}$ of values. Thus:

$$\Gamma^\sharp_0 = \{\gamma^\sharp : \langle \mathbf{X} \to \top, 0, \bot \rangle \mid \gamma^\sharp(\mathbf{x}_1) = \{(\mathbf{x}_1, 0)\}\}$$

where $\bot$ denotes the empty mapping. The transformers $[\![s]\!]^\sharp$ are the abstract effects of edge labels, $[\![p]\!]^\natural$ represents the modifying potential of procedure p, and $@^\sharp$ is the application of a modifying potential to a given assignment of registers and local addresses to sets of values. For registers $\mathbf{x}_i$, we have:

$$((X, M)\ @^\sharp\ \langle \rho^\sharp, c_q, \lambda^\sharp \rangle)(\mathbf{x}_i) = \begin{cases} \top & \text{if } \mathbf{x}_i \notin X \\ \rho^\sharp(\mathbf{x}_i) & \text{if } \mathbf{x}_i \in X \end{cases}$$

For a local offset a,

$$((X, M)\ @^\sharp\ \langle \rho^\sharp, c_q, \lambda^\sharp \rangle)(a) = \top$$

if there exists some $\mathbf{x}_i \in \mathbf{X}, d \in M(\mathbf{x}_i), (\mathbf{x}_1, b) \in \rho^\sharp(\mathbf{x}_i)$ such that $a = b + d$.
Otherwise,

$$((X, M)\ @^\sharp\ \langle \rho^\sharp, c_q, \lambda^\sharp \rangle)(a) = \lambda^\sharp(a)$$

i.e., remains unchanged. In case that for $\mathbf{x}_i \in \mathbf{X}, d \in M(\mathbf{x}_i)$ and $(\mathbf{x}_1, b) \in \rho^\sharp(\mathbf{x}_i)$, $b < d$, i.e., the offset $b + d$ is *negative*, a potential access to a local memory location outside the stack is detected. In this case, we again flag a warning and abort the analysis.

The abstract transformers $[\![s]\!]^\sharp$ are identical to the abstract transformers which we use for an auxiliary intraprocedural value analysis when computing the modifying potential of procedures.

Example 2. Recall the program from Example 1 and its corresponding control flow representation in Figure 1. According to our effect description the modifying potential of procedure f is given by $[\![f]\!]^\natural = (\{\mathbf{x}_1\}, \{(\mathbf{x}_3, 0), (\mathbf{x}_3, 4), (\mathbf{x}_3, 8), (\mathbf{x}_3, 12)\})$ at program point 23. At program point 6, directly before the procedure call, we have the following register assignment: $(\mathbf{x}_1, 8) \mapsto \{(\mathbf{x}_0, 13)\}; \mathbf{x}_2 \mapsto \{(\mathbf{x}_1, 12)\}; \mathbf{x}_3 \mapsto \{(\mathbf{x}_1, 12)\}$. Embedding the side-effect information of f into the intraprocedural analysis of procedure main, then directly after the procedure call $f()$ the value of the stack location $(\mathbf{x}_1, 8)$ remains unchanged. Thus we arrive at the following register assignment for program point 7: $(\mathbf{x}_1, 8) \mapsto \{(\mathbf{x}_0, 13)\}$. □

Next we describe how to compute the modifying potential of a procedure.

3.2 Effect Computation

Assume that we are given a mapping μ which assigns to each procedure g an approximation of its modifying potential, i.e., $\mu(g) = (X_g, M_g)$ where $X_g \subseteq \mathbf{X}$ and $M_g : \mathbf{X} \to 2^{\mathbb{Z}}$. Relative to μ, we perform for a given procedure g, an intraprocedural *value* analysis which determines for every program point u of g and all registers $\mathbf{x}_i \in \mathbf{X}$ as well as all g-local memory cells $a \in [0, c_g]$, sets $\rho^\natural(u)(\mathbf{x}_i), \lambda^\natural(u)(a) \subseteq 2^{(\mathbf{X} \cup \{\mathbf{x}_0\}) \times \mathbb{Z}}$ of values relative to the values of registers at procedure entry. Again c_g denotes the stack frame size of procedure g. Relative to the mappings $\langle \rho^\natural(u), \lambda^\natural(u) \rangle$, u a program point of g, the modifying potential of g is given by $\mathsf{eff}_g(\mu) = (X', M')$ where

$$\begin{aligned} X' &= \{\mathbf{x}_i \in \mathbf{X} \mid \rho^\natural(r_g)(\mathbf{x}_i) = \{(\mathbf{x}_i, 0)\}\} \\ M' &= (\mathbf{X} \times 2^{\mathbb{Z}}) \cap \big(\bigcup\{[\![t]\!]^\natural(\rho^\natural \mid (u, M[t] := \mathbf{x}_j, \_) \in E_g)\} \cup \\ &\qquad \{(\mathbf{x}_k, z + z') \mid (u, f(), \_) \in E_g, (\mathbf{x}_k, z) \in \rho^\natural(u)(\mathbf{x}_{k'}), (\mathbf{x}_{k'}, z') \in M_f\}\big) \end{aligned}$$

Note that (X', M') monotonicly depends on μ, only if the mappings $\rho^\natural(u)$ as well as $\lambda^\natural(u)$ monotonicly depend on μ. Thus, given such a monotonic value analysis of the functions $\rho^\natural, \lambda^\natural$, the modifying potential can be determined as the least (or some) solution of the constraint system:

$$[\![g]\!]^\natural \sqsupseteq \mathsf{eff}_g((\emptyset, \emptyset)), \qquad \text{for all procedures } g$$

It remains to construct the constraint system whose least solution characterises the mappings $\rho^\natural(u) : \mathbf{X} \to \bar{\mathbb{V}}$ and $\lambda^\natural : [0, c_g] \to \bar{\mathbb{V}}$ w.r.t. procedure g with stack frame size c_g. Now, $\bar{\mathbb{V}}$ is the complete lattice $2^{(\mathbf{X} \cup \{\mathbf{x}_0\}) \times \mathbb{Z}}$. Note that the greatest element $\top$ of $\bar{\mathbb{V}}$ is given by $\top = (\mathbf{X} \cup \{\mathbf{x}_0\}) \times \mathbb{Z}$.

The abstract arithmetic operations are obtained by first considering single elements $(\mathbf{x}_i, c)$. We define:

$$\begin{aligned} (\mathbf{x}_i, c_1) \odot (\mathbf{x}_0, c_2) &= \{(\mathbf{x}_i, c_1 \odot c_2)\} \\ (\mathbf{x}_0, c_2) \odot (\mathbf{x}_i, c_1) &= \{(\mathbf{x}_i, c_1 \odot c_2)\} \\ (\mathbf{x}_i, c_1) \odot (\mathbf{x}_j, c_2) &= \top \end{aligned}$$

for $\odot \in \{+, \cdot\}$, while for subtraction we define:

$$\begin{aligned}
(\mathbf{x}_i, c_1) - (\mathbf{x}_i, c_2) &= \{(\mathbf{x}_0, c_1 - c_2)\} \\
(\mathbf{x}_i, c_1) - (\mathbf{x}_0, c_2) &= \{(\mathbf{x}_i, c_1 - c_2)\} \\
(\mathbf{x}_i, c_1) - (\mathbf{x}_j, c_2) &= \top
\end{aligned}$$

for $i \neq j$. These definitions then are lifted to abstract operators $\odot^\natural$ on sets of such elements, i.e., to $\bar{\mathbb{V}}$ by:

$$S_1 \odot^\natural S_2 = \bigcup \{s_1 \odot s_2 \mid s_1 \in S_1, s_2 \in S_2\}$$

The definition of these abstract operators gives rise to an abstract evaluation $[\![t]\!]^\natural$ of expressions t w.r.t. to a given register assignment $\rho^\natural$.

$$[\![t]\!]^\natural(\rho^\natural) = \begin{cases} [\![t_1]\!]^\natural(\rho^\natural) \odot^\natural [\![t_2]\!]^\natural(\rho^\natural) & \text{if } t = t_1 \odot t_2 \\ \rho^\natural(\mathbf{x}_i) & \text{if } t = \mathbf{x}_i \\ \{(\mathbf{x}_0, c)\} & \text{if } t = c \end{cases}$$

The mappings $\rho^\natural(u), \lambda^\natural(u)$ for a procedure g then can be characterised by the least solution of the following constraint system:

$$\begin{array}{lll}
[\mathrm{R}^\natural_\mu 0] & R^\natural_\mu(s_g) \sqsupseteq \langle \{\mathbf{x}_i \mapsto \{(\mathbf{x}_i, 0)\} \mid \mathbf{x}_i \in \mathbf{X} \cup \{\mathbf{x}_0\}\}, 0, \bot\rangle & \\
[\mathrm{R}^\natural_\mu 1] & R^\natural_\mu(v) \sqsupseteq [\![s]\!]^\natural(R^\natural_\mu(u)) & (u, s, v) \text{ with } s \in \mathsf{stm} \cup \mathsf{guards} \\
[\mathrm{R}^\natural 2] & R^\natural_\mu(v) \sqsupseteq (R^\natural_\mu(r_f))@^\natural(R^\natural_\mu(u)) & (u, f(), v) \text{ a call edge}
\end{array}$$

At procedure start, all registers $\mathbf{x}_i \in \mathbf{X}$ are mapped to their symbolic values $\{(\mathbf{x}_i, 0)\}$ (constraint $[\mathrm{R}^\natural 0]$).

The effect of the instruction decrementing the stack pointer is that a new stack frame $\mathbf{S}^\sharp{}_g$, i.e., the local memory locations $\mathbf{S}^\sharp{}_g$, for procedure g is allocated. Thus, we have:

$$[\![\mathbf{x}_1 := \mathbf{x}_1 - c]\!]^\natural(\langle \rho^\natural, c', \lambda^\natural\rangle) = \langle \rho^\natural, c, \{a \mapsto \top \mid a \in [0, c]\}\rangle$$

The effect of the instruction incrementing the stack pointer is that the stack frame $\mathbf{S}^\sharp{}_g$ for procedure g is deallocated. Thus, the set of variables is restricted to registers only:

$$[\![\mathbf{x}_1 := \mathbf{x}_1 + c]\!]^\natural(\langle \rho^\natural, c, \lambda^\natural\rangle) = \langle \rho^\natural, 0, \bot\rangle$$

The second constraint $[\mathrm{R}^\natural 1]$ handles assignments to other registers, memory accesses and guards. For assignments, we have:

$$\begin{aligned}
[\![\mathbf{x}_i := ?]\!]^\natural(\langle \rho^\natural, c, \lambda^\natural\rangle) &= \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \top\}, c, \lambda^\natural\rangle \\
[\![\mathbf{x}_i := t]\!]^\natural(\langle \rho^\natural, c, \lambda^\natural\rangle) &= \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto [\![t]\!]^\natural(\rho^\natural)\}, c, \lambda^\natural\rangle
\end{aligned}$$

with $i \neq 1$. The effect of a *memory read access* instruction in procedure g on a state $\langle \rho^\natural, c, \lambda^\natural\rangle$ is given by:

$[\![\mathbf{x}_i := M[t]]\!]^\natural(\langle \rho^\natural, c, \lambda^\natural\rangle) =$

$$\begin{cases} \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \top_G\}, c, \lambda^\natural \rangle & \text{if } [\![t]\!]^\natural(\rho^\natural) \subseteq \{\mathbf{x}_0\} \times \mathbb{Z} \\ \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \bigcup\{\lambda^\natural(c') \mid (\mathbf{x}_1, c') \in [\![t]\!]^\natural(\rho^\natural)\}\}, c, \lambda^\natural \rangle & \text{if } [\![t]\!]^\natural(\rho^\natural) \subseteq \{\mathbf{x}_1\} \times [0, c] \\ \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \top\}, c, \lambda^\natural \rangle & \text{otherwise} \end{cases}$$

with $i \neq 1$. The *global top* $\top_G$ describes the set of all possible global addresses: $\top_G = \mathbf{x}_0 \times \mathbb{Z}$.

Since we do not track the values of global variables, in case of a memory access to a global memory location, variable $\mathbf{x}_i$ may be assigned every possible global value. If the evaluation of a memory access expression yields that a local variable $(\mathbf{x}_1, c)$ is addressed which belongs to the stack frame of the current procedure, its value is assigned to register $\mathbf{x}_i$. For all other cases the value of $\mathbf{x}_i$ is overapproximated by the top element.

For a *memory write* instruction the abstract effect function is defined by:

$$[\![M[t] := \mathbf{x}_j]\!]^\natural(\langle \rho^\natural, c, \lambda^\natural \rangle) =$$

$$\begin{cases} \langle \rho^\natural, c, \lambda^\natural \oplus \{c' \mapsto \rho^\natural(\mathbf{x}_j)\}\rangle & \text{if } \{(\mathbf{x}_1, c')\} = [\![t]\!]^\natural(\rho^\natural) \wedge c' \in [0, c] \\ \langle \rho^\natural, c, \lambda^\natural \oplus \{c' \mapsto (\lambda^\natural(c') \cup \rho^\natural(\mathbf{x}_j)) \mid (\mathbf{x}_1, c') \in [\![t]\!]^\natural(\rho^\natural), c' \in [0, c]\} & \text{otherwise} \end{cases}$$

with $j \neq 1$. If the accessed memory location denotes a single local variable of the current stack frame the value of variable $\mathbf{x}_j$ is assigned to the corresponding local memory location. If the evaluation of a memory access expression yields a set of possibly accessed local memory locations, all their values are extended by the value of variable $\mathbf{x}_j$. In all the other cases, none of the local memory locations may receive new values. If an element $(\mathbf{x}_1, c')$ is found in $[\![t]\!]^\natural(\rho^\natural)$ with $c' \notin [0, c]$, we issue a warning and abort the analysis.

Guards are used for restricting the sets of possible values of registers. Sets of possible values, however, can only be compared if they refer to the same base register. First let us consider the guard $\mathbf{x}_i \bowtie c$. If $\rho^\natural(\mathbf{x}_i) = \{\mathbf{x}_0\} \times S$ and $c \in \mathbb{Z}$, let $S' = \{s \in S \mid s \bowtie c\}$. Then we set:

$$[\![\mathbf{x}_i \bowtie c]\!]^\natural(\langle \rho^\natural, c', \lambda^\natural \rangle) = \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \{\mathbf{x}_0\} \times S'\}, c', \lambda^\natural \rangle$$

Likewise, for a guard $\mathbf{x}_i \bowtie \mathbf{x}_j$, if $\rho^\natural(\mathbf{x}_i) = \{\mathbf{x}_k\} \times S_1$ and $\rho^\natural(\mathbf{x}_j) = \{\mathbf{x}_k\} \times S_2$, then we set

$$S_1' = \{s \in S_1 \mid \exists s_2 \in S_2 : s \bowtie s_2\}$$
$$S_2' = \{s \in S_2 \mid \exists s_1 \in S_1 : s_1 \bowtie s\}$$

and define

$$[\![\mathbf{x}_i \bowtie \mathbf{x}_j]\!]^\natural(\langle \rho^\natural, c', \lambda^\natural \rangle) = \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \{\mathbf{x}_k\} \times S_1', \mathbf{x}_j \mapsto \{\mathbf{x}_k\} \times S_2'\}, c', \lambda^\sharp \rangle$$

In all other cases, guards have no effect on the register assignment.

3.3 Correctness

The correctness proof is based on a description relation $\Delta \subseteq \Gamma \times \Gamma^\sharp$ between concrete and abstract states. A description relation is a relation with the following property: $s \,\Delta\, s_1^\sharp \;\wedge\; s_1^\sharp \sqsubseteq s_2^\sharp \Rightarrow s \,\Delta\, s_2^\sharp$ for $s \in \Gamma$ and $s_1^\sharp, s_2^\sharp \in s^\sharp$.
Here, the concrete state $\langle \rho, f, \lambda \rangle$ is described by the abstract state $\langle \rho^\sharp, c, \lambda^\sharp \rangle \in \Gamma^\sharp$ if

- $f = (x_r, x_{r-1}, \ldots, x_0)$ with $x_{r-1} - x_r = c$;
- $\rho(\mathbf{x}_i) \in \gamma_{x_r}(\rho^\sharp(\mathbf{x}_i))$ for all $\mathbf{x}_i$;
- $\lambda(L, a) \in \gamma_{x_r}(\lambda^\sharp(a - x_r))$.

Here, the concretisation γ_x replaces the tagging register $\mathbf{x}_0$ with G, while symbolic local addresses $(\mathbf{x}_1, b)$ are translated into $(L, x - b)$.

Additionally, we require a description relation between the transformations induced by same-level concrete computations and abstract states from $R^\natural_\mu$. Assume that $T : \Gamma \to 2^\Gamma$ is a transformation which preserves frame structures, i.e., $f = f'$ for all $\langle \rho', f', \lambda' \rangle \in T(\langle \rho, f, \lambda \rangle)$. Then T is described by $\langle \rho^\natural, c, \lambda^\natural, X, M \rangle$ if for all $\langle \rho', f', \lambda' \rangle \in T(\langle \rho, f, \lambda \rangle)$,

- $f = (x_r, x_{r-1}, \ldots, x_0)$ where $x_{r-1} - x_r = c$;
- $\rho'(\mathbf{x}_i) \in \{\rho(\mathbf{x}_j) + (G, a) \mid (\mathbf{x}_j, a) \in \rho^\natural(\mathbf{x}_i)\}$ for all $\mathbf{x}_i$;
- $\lambda'(L, a) \in \{\rho(\mathbf{x}_j) + (G, a) \mid (\mathbf{x}_j, a) \in \lambda^\natural(a - x_r)\}$ for all $a \in [x_r, x_{r-1})$;
- $\rho(\mathbf{x}_i) = \rho'(\mathbf{x}_i)$ for all $\mathbf{x}_i \in X$;
- If $\lambda(L, b) \neq \lambda'(L, b)$ for $b \geq x_{r-1}$, then $(L, b) = \rho(\mathbf{x}_i) + (G, a)$ for some $(\mathbf{x}_i, a) \in M$.

Here, we have assumed that $\rho(\mathbf{x}_0)$ always returns $(G, 0)$. The description relation for transformers is preserved by function composition, execution of individual statements as well as least upper bounds. Therefore, we obtain by induction on the fixpoint iterates:

Theorem 1. Correctness.

1. *Assume that $E, R^\natural_\mu$ denote the least solutions of the constraint systems for the concrete effects of procedures and their side effects, respectively. Then for every program point u, the transformer $E(u)$ is frame preserving, and $E(u)\ \Delta\ R^\natural_\mu(u)$.*
2. *Let $R, R^\sharp$ denote the least solutions of the constraint systems for the collecting semantics and abstract reachability, respectively. Then for every program point u, $R(u)\ \Delta\ R^\sharp(u)$.*

Local Addresses Escaping to the Heap

Although the description of the collecting semantics excludes those programs, where e.g. local addresses may be written into global memory, our abstract approach is able to detect such situations. Therefore, the modifying potential (X, M) of a procedure can be enriched by an additional component η, where

$$\eta : (\mathbf{X} \cup \{\mathbf{x}_0\}) \to 2^{\mathbf{X}}$$

η is a mapping which assigns to each register $\mathbf{x}_i$ the subset of registers $\mathbf{x}_j$ such that $\mathbf{x}_j$-relative addresses may escape to the global memory if $\mathbf{x}_i$ is a global address. Provided an abstract value analysis which maps registers to some abstract values ($\rho^\natural$) and local memory locations to some abstract values ($\lambda^\natural$), the effect computation for η is given by:

$$\begin{aligned}\eta'(\mathbf{x}_i) = \{&\mathbf{x}_k \in \mathbf{X} \mid \exists (u, M[t] := \mathbf{x}_j, \_) \in E_g, \exists z, z' \in \mathbb{Z}.(\mathbf{x}_k, z) \in V_\mu(u)(\mathbf{x}_j) \\ &\wedge (\mathbf{x}_i, z') \in [\![t]\!]^\natural(V_\mu(u))\} \cup \\ &\{\mathbf{x}_j \in \mathbf{X} \mid (u, f(), \_) \in E_g, (\mathbf{x}_i, z) \in V_\mu(u)(\mathbf{x}_k), \mathbf{x}_{k'} \in \eta_f(\mathbf{x}_k), \\ &(\mathbf{x}_j, z') \in V_\mu(u)(\mathbf{x}_{k'})\}\end{aligned}$$

Now the modifying potential of a procedure g, i.e. the triple (X, M, η), can be determined as the least solution of the constraint system:

$$[\![g]\!]^\natural \sqsupseteq \mathsf{eff}_g((\emptyset, \bot, \emptyset)), \qquad \text{for all procedures } g$$

The component η serves as another soundness check. If a register $\mathbf{x}_i$ may hold a global address, i.e., $\rho^\natural(\mathbf{x}_i)$ contains an element $(\mathbf{x}_0, z)$, and $\mathbf{x}_j \in \eta(\mathbf{x}_i)$ is found such that $\rho^\natural(\mathbf{x}_j)$ contains a stack address, i.e., an element $(\mathbf{x}_1, z')$, then some reference to the stack may have escaped to the global memory. In this case, we flag a warning and again abort the analysis.

4 Experimental Results

We have implemented the side-effect analysis in our assembly analyser VoTUM [30]. Our benchmark suite consists of statically linked PowerPC assemblies of publicly available programs such as the cryptography library **openSSL**, and the HTTP server **thttpd**. Moreover, we experimented with the four larger programs basename, vdir, chmod, chgrp from the Unix GNU Coreutils package, and also ran the prototype on **gzip** from SPECint. The vehicle control **control**[29] is generated from SCADE and thus is very restrictive in the use of pointers and does not use dynamic memory [9]. The binaries of these programs have sizes between 0.6 and 3.8 MB and have been compiled with gcc version 4.4.3 at optimisation levels $O0$ and $O2$ and are statically linked with glibc. We conducted our experiments on a 2.2 GHz quad-core machine equipped with 16 GB of physical memory where the algorithm occupies just a single core. We have implemented our side-effect analysis with the enhancement of Section 3.3. Additionally, we track the values of global memory locations with static addresses. We have evaluated the significance of our side-effect analysis by means of:

- The percentage of side-effect free procedures, i.e. those procedures that have no effect to the stack frame of other procedures, in the binary (Column **SE_free**).
- The absolute number of procedures where locals may escape to the heap (Column **Esc**).
- The percentage of locals that have to be invalidated after a procedure call and in parentheses the absolute number of call sites with side-effects (Column **Inv**).
- The absolute number of procedures where a safe handling of the link register and the stack pointer could not be verified (Columns **LR** and **SP**).
- The percentage of procedures where our analysis infers unbounded side-effects (Column **Top**). In contrast to our theoretical conception from Section 3 we follow a less restrictive approach in our implementation: in case of an unbounded side-effect our analyser reports a warning and proceeds by setting all the values of the registers of the caller to unknown values except for the stack pointer and the link register.

Evaluating our experimental results, we have: Column **SE_free** reveals that $20\% - 50\%$ of all the procedures in the binary *have side-effects*. However, side-effect information concerns only the local variables of up to 2% of all call sites (Column **Inv**). Embedding the side-effects into the caller invalidates a third of the caller‘s locals in average. Column **Esc** illustrates that stack addresses rarely escape to global memory. This is the case for approximately 0.5% of all procedures in our benchmark suite. For at most 2% of all the

Program	Size	Instr	Procs	Calls	SE_free	Esc	Inv	LR	SP	Top	M	T
openSSL_O0	3.8MB	769511	6710	38188	78%	27	22%(549)	152	21	4%	15168	2298
thttpd_O0	0.8MB	196493	1200	7947	48%	2	27% (115)	28	17	9%	6157	1189
basename_O0	0.7MB	168232	916	5303	53%	5	33% (89)	8	5	8%	3387	1417
chgrp_O0	0.8MB	197260	1119	6338	51%	5	22% (192)	11	5	8%	4264	2052
chmod_O0	0.7MB	179415	1027	5767	52%	1	38% (96)	24	12	7%	3609	1739
vdir_O0	0.9MB	221665	1307	6988	56%	3	25% (165)	45	15	7%	5329	1060
gzip_O0	0.8MB	166213	1078	5954	53%	2	30%(112)	30	10	7%	3924	1858
control_O0	0.6MB	162530	819	4948	49%	5	31%(112)	9	8	9%	3284	2197
openSSL_O2	2.9MB	613882	6234	43407	72%	6	30%(486)	464	20	4%	15396	5594
thttpd_O2	0.8MB	189034	1150	5706	48%	1	19%(348)	44	13	9%	5149	1906
basename_O2	0.7MB	139271	907	5386	51%	1	30%(61)	23	6	8%	3326	1555
chgrp_O2	0.7MB	164420	1083	6522	49%	5	23%(199)	7	7	7%	4086	1234
chmod_O2	0.7MB	148702	989	6005	50%	5	29%(123)	8	7	7%	3749	1629
vdir_O2	0.8MB	177022	1200	7106	51%	2	35%(80)	40	7	7%	5152	1304
gzip_O2	0.7MB	162380	1028	6323	52%	1	33%(113)	28	6	8%	3960	1841
control_O2	0.6MB	161184	819	4989	49%	5	33%(81)	8	5	9%	3286	2380

Size: the binary file size; **Instr**: the number of assembler instructions;
Procs: the number of procedures; **Calls**: the number of procedure calls;
T: the absolute running time in seconds; **M**: the peak memory consumption in MB.

procedures our analysis could not verify the integrity of the organisational stack cells, cf. Columns **LR** and **SP**. These procedures are either related to error handling or require relational domains to be analysed precisely. For instance when the bound for iterating over an array is provided as the parameter of a procedure, domains like polyhedra [7] or symbolic intervals [26] have to be used. For at most 9% of all procedures our analysis inferred an unbounded side-effect (Column **Top**). In order to achieve more precision in these cases, we have to instantiate our side-effect analysis with more sophisticated domains [26,7] and better widening strategies (for sets of values). For our benchmark programs, the analyser consumes between 3 and 15 GB memory and analysis time is between 20 minutes up to 1.5 hours. In order to rate the light-weightness of our approach, we want to mention that the control flow reconstruction analysis of that large binaries has approximately the same running time.

Summarising, for most of the procedures our side-effect analysis provides tight bounds for parameter relative write accesses even with the simplistic instantiation of our value analysis with sets of values. Since approximately 50% of all procedures have side-effects, for a sound and precise analysis it is mandatory to consider side-effect information.

5 Conclusion

We have presented a reasonably fast interprocedural analysis of assembly code for inferring the side-effects of procedure calls onto the runtime stack. Such an analysis contributes in refining arbitrary analyses by identifying possible modifications in the stack frame of the caller through a procedure call. Our analysis allows to verify in how far basic assumptions for the analysis of the assembly code are met. Still, precision could be significantly enhanced, if the analysis tracked relational invariants for values. Our approach can be extended to a *purity analysis* which not only tracks modifications to the stack but also to the heap. In order to deal with assembly code generated from, e.g. object-oriented programs, it also seems inevitable to combine our methods with simple forms of heap analysis such as [2].

References

1. G. Balakrishnan and T. Reps. Recovery of Variables and Heap Structure in x86 Executables. Technical report, University of Wisconsin, Madison, 2005.
2. G. Balakrishnan and T. Reps. Recency-Abstraction for Heap-Allocated Storage. In *13th International Static Analysis Symposium, SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 221–239. Springer, 2006.
3. J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29–41, New York, NY, USA, 1979. ACM.
4. J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, USA, 1993. ACM.
5. K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 57–66, New York, NY, USA, 1988. ACM.
6. P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, pages 269–295. Springer-Verlag, Germany, 1992.
7. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *5th Ann. ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–97, 1978.
8. S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–24, New York, NY, USA, 1998. ACM.
9. F.-X. Dormoy and E. Technologies. *SCADE 6 A Model Based Solution For Safety Critical Software Development*, 2008. `http://www.esterel-technologies.com/technology/WhitePapers/`.
10. T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis, 2009. `http://www.zynamics.com/downloads/csw09.pdf`.
11. M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.
12. A. Flexeder, B. Mihaila, M. Petter, and H. Seidl. Interprocedural control flow reconstruction. In *The Eight ASIAN Symposium on Programming Languages and Systems, (APLAS 2010)*, pages 188–203. Springer Verlag, 2010.
13. A. Flexeder, M. Petter, and H. Seidl. Analysis of executables for WCET concerns. Technical Report, Institut für Informatik, 2008. Available online http://www2.in.tum.de/ flexeder/report38.pdf.

14. B. Frey. *PowerPC Architecture Book, Version 2.02*, November 2005. `http://www.ibm.com/developerworks/systems/library/es-archguide-v2.html`.
15. B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August. Practical and Accurate Low-Level Pointer Analysis. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.
16. J. Kinder and H. Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, volume 5123 of *LNCS*, pages 423–427. Springer, 2008.
17. J. Kinder, H. Veith, and F. Zuleger. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, volume 5403 of *Lecture Notes in Computer Science*, pages 214–228. Springer, Jan 2009.
18. W. Landi, B. G. Ryder, and S. Zhang. Interprocedural Modification Side Effect Analysis With Pointer Aliasing. In *In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, 1993.
19. C. Linn, S. Debray, G. Andrews, and B. Schwarz. Stack Analysis of x86 Executables, 2004. `www.cs.arizona.edu/~debray/Publications/stack-analysis.pdf`.
20. R. E. Moore and F. Bierbaum. *Methods and Applications of Interval Analysis (SIAM Studies in Applied and Numerical Mathematics) (Siam Studies in Applied Mathematics, 2.).* Soc for Industrial & Applied Math, 1979.
21. M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 330–341, 2004.
22. M. Müller-Olm and H. Seidl. Upper Adjoints for Fast Inter-procedural Variable Equalities. In *17th European Symposium on Programming (ESOP)*, 2008.
23. T. Reps and G. Balakrishnan. Improved Memory-Access Analysis for x86 Executables. In *Compiler Construction, 17th International Conference, CC*, Lecture Notes in Computer Science, pages 16–35. Springer, 2008.
24. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 100–111. ACM, 2006.
25. A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. In *VMCAI*, pages 199–215. Springer-Verlag, 2005.
26. S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program Analysis Using Symbolic Ranges. In *Static Analysis, 14th International Symposium*, volume 4634 of *Lecture Notes in Computer Science*, pages 366–383. Springer, 2007.
27. M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Application*, pages 189–234, 1981.
28. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.
29. Sicherheitsgarantien Unter REALzeitanforderungen, 2010. `http://www.sureal-projekt.org/`.
30. *VoTUM*, 2010. `http://www2.in.tum.de/votum`.
31. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 1995. ACM.