Static Analysis Of Binary Code With Memory Indirections Using Polyhedra*

Clément Ballabriga[1], Julien Forget[1], Laure Gonnord[2], Giuseppe Lipari[1], and Jordy Ruiz[1]

[1] CRIStAL (Univ. Lille, CNRS, Centrale Lille, UMR 9189), Lille, France
firstname.lastname@univ-lille.fr
[2] Univ. Lyon, LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), Lyon, France
laure.gonnord@ens-lyon.fr

Abstract. In this paper we propose a new abstract domain for static analysis of binary code. Our motivation stems from the need to improve the precision of the estimation of the Worst-Case Execution Time (WCET) of safety-critical real-time code. WCET estimation requires computing information such as upper bounds on the number of loop iterations, unfeasible execution paths, etc. These estimations are usually performed on binary code, mainly to avoid making assumptions on how the compiler works. Our abstract domain, based on polyhedra and on two mapping functions that associate polyhedra variables with registers and memory, targets the precise computation of such information. We prove the correctness of the method, and demonstrate its effectiveness on benchmarks and examples from typical embedded code.

1 Introduction

In real time systems, checking that computations complete before their deadlines under all possible contexts is a crucial activity. Worst-Case Execution Time (WCET) analysis consists in computing an upper bound to the longest execution path in the code. It is usually performed on the binary code, because it needs information on the low-level instructions executed by the hardware processor.

In this paper, we propose a static analysis of binary code based on abstract interpretation using a polyhedra-based abstract domain. Our motivation is the need to enhance existing WCET analysis by improving the computation of upper bounds on the number of iterations in loops. However, our abstract domain has other potential applications (not developed in this paper), such as buffer-overflow analysis, unfeasible paths analysis or symbolic WCET computation [1].

Most analyses by abstract interpretation proposed in the literature are performed on *source code*. On the contrary, as it is usually the case for WCET analysis, we propose to analyze *binary code*. There are several important advantages in performing static analysis of binary code: 1) we analyze the code that actually runs on the machine, hence no need for additional assumptions on how

* Partially funded by the French National Research Agency (ANR), Corteva project.

```c
#define SIZE 10
foo(int offset) {
  int i, bound = 10;
  int tab[SIZE];
  if ((offset > SIZE) || (offset < 0))
    return -1;
  tab[offset] = 100;
  for (i = 0; i < bound; i++);
}
```

examples/netcode.c

Fig. 1. Network-inspired benchmark

the compiler works; 2) in presence of undefined behaviors (of source code), the analysis is more accurate; 3) we can perform the analysis even without access to the source code.

The main problem is that, in higher-level representations, the variables, addresses and values are well identified. In binary code, the notion of program variable is lost, so we can only analyze processor registers and memory locations. We propose to identify the subset of registers and memory locations to be represented in the abstract state as the analysis progresses. This representation enables us to design a relational analysis on binary code, which is the main contribution of the paper.

1.1 Motivating example

As a motivating example, we present a snippet of C code, inspired from packet processing network drivers in Figure 1[3]. We remind however that our methodology addresses (disassembled) binary code.

The `send_request` function sends a request in some application-layer protocol that runs over UDP/IP. Lines 12-13 build a packet composed of a variable-length IP header, a fixed-length UDP header, and a variable-length UDP payload (some operations on IP or UDP fields have been omitted). Note that the starting address of the UDP header depends on the size of the IP header (`h1->hdr_len`). At line 17, we call the function responsible for putting the useful data (payload) into the packet. At line 18, the packet is sent using the `send_packet` function, which belongs to the lower-level network layer API. This function does not take the packet size as parameter, since it can be deduced from the header: in lines 2-3, the function parses the packet to obtain the UDP payload size, and the UDP checksum is computed by iterating over the payload.

To automatically compute a bound on the number of iterations of the loop at line 4, the analysis has to discover that `udp_l` equals `udp_size` (due to line 16). This can be done with an appropriate use of a *relational abstract domain*. However, very few of the existing analyses running on binary code use a relational

[3] The original bench listing is available here: `https://pastebin.com/C5UPYRx3`

domain, and to the best of our knowledge, none support relations between addresses that are not know statically (`udp_l`, `udp_size`). Let us emphasize that such a use of pointers and memory buffers is typical of many embedded systems: for instance in network packet processing, but also in many device drivers.

1.2 Contribution

The contributions of the paper are:

- A new *relational* abstract domain POLYMAP, which consists of a polyhedron and two mappings that track the correspondence between data locations (registers or memory) and polyhedra variables;
- An abstract interpretation procedure, which computes abstract states of POLYMAP for a small assembly language, and which we prove to be sound;
- An experimental evaluation of our prototype called Polymalys. It implements the previous procedure and computes upper bounds to loop iterations. We compare Polymalys with other existing tools on a set of classic benchmarks.

2 Language definition

In this section, we define the analyzed language, called `MEMP`, a simplified assembly language where we focus on memory indirection operators.

2.1 Syntax

In order to simplify the presentation, we make the following assumptions: all data locations have the same size, memory accesses are aligned to the word size, there are no integer overflows, and function calls are inlined (these limitations could be lifted using for instance [2,3]). We also reduce the set of instructions to a minimum (Polymalys actually supports the ARM A32 instruction set). The syntax of `MEMP` is defined in Figure 2. A program is a sequence of labeled instructions. Instructions operate on registers, labels or constants. Concerning memory instructions, if r contains value c, then $*(r)$ denotes the content at address r (below, we overload the notation and also denote $*(c)$ for this content). OP^c denotes the concrete semantics of operation OP. `RAND` emulates undefined registers, to represent e.g. function parameters. Other instructions are directly commented in the figure (on the left of each instruction).

2.2 Formal semantics

The small-steps semantics of `MEMP` is defined below. The semantics of data and arithmetic/logic operations is defined in Figure 3 by function $\xrightarrow{i}$, which operates in a context $(\mathcal{R}, *)$ consisting of two mappings where:

- $\mathcal{R} : R \nrightarrow \mathbb{Z}$ is the *registers content*, which maps registers to their values. We assume that it is initially empty;

Programs (P)	$::= l_1 : I_1,\ l_2 : I_2,\ \ldots,\ l_n$: `END`	
Labels (L)	$::= \{l_1, l_2, \ldots\}$	
Registers (R)	$::= \{r_1, r_2, \ldots\}$	
Constants (C)	$::= \{c_1, c_2, \ldots\}$	
Instructions (I)	$::=$	
$r_1 \leftarrow \mathrm{OP}^c(r_2, r_3)$	|	`OP r1 r2 r3`
$r \leftarrow c$	|	`SET r c`
Emulate undefined r	|	`RAND r`
$r_1 \leftarrow *(r_2)$	|	`LOAD r1 r2`
$*(r_1) \leftarrow r_2$	|	`STORE r1 r2`
Branch to l if $r = 0$	|	`BR r l`
Halt	|	`END`

Fig. 2. Syntax of `MEMP`

– $* : \mathbb{Z} \nrightarrow \mathbb{Z}$ is the *memory content*, which maps memory addresses to their values. We assume that it is also initially empty. Note that integer wrapping could be used to restrain addresses to be in $\mathbb{N}$ instead of $\mathbb{Z}$ [2].

For a given mapping m, we denote $m[x : y]$ the mapping m' such that $m'(x) = y$ and, for every register $x' \neq x$, $m'(x') = m(x')$. In other words, $m[x : y]$ denotes a single mapping substitution (or mapping addition if x was previously unmapped). We also denote $m \backslash (x_1 : x_2)$ the mapping such that the association $x_1 : x_2$ is removed from m.

$$\frac{}{(\texttt{SET r c}, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}[r : c], *)} \qquad \frac{c = random()}{(\texttt{RAND r}, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}[r : c], *)}$$

$$\frac{\mathcal{R}(r_2) = c_2 \qquad \mathcal{R}(r_3) = c_3 \qquad c_1 = \mathrm{OP}^c(c_2, c_3)}{(\texttt{OP r}_1\ \texttt{r}_2\ \texttt{r}_3, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}[r_1 : c_1], *)}$$

$$\frac{\mathcal{R}(r_2) = c_2 \qquad *(c_2) = c_1}{(\texttt{LOAD r}_1\ \texttt{r}_2, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}[r_1 : c_1], *)} \qquad \frac{\mathcal{R}(r_1) = c_1 \qquad \mathcal{R}(r_2) = c_2}{(\texttt{STORE r}_1\ \texttt{r}_2, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}, *[c_1 : c_2])}$$

Fig. 3. Semantics of data and arithmetic operations.

The semantics of control flow operations is defined in Figure 4, by the function $\xrightarrow{c}$, which adds a program counter *pc* to the previous context. We use $\xrightarrow{e}$ to denote the last transition of the program.

3 Abstract domain

The abstract domain we propose is based on the *polyhedral abstract domain* [4], to which we add information to track relations between polyhedra variables and registers or memory addresses.

$$\frac{P[pc] = \texttt{BR r l} \qquad \mathcal{R}(r) \neq 0}{P \vdash (pc, \mathcal{R}, *) \xrightarrow{c} (pc+1, \mathcal{R}, *)} \qquad \frac{P[pc] = \texttt{BR r l} \qquad \mathcal{R}(r) = 0}{P \vdash (pc, \mathcal{R}, *) \xrightarrow{c} (l, \mathcal{R}, *)}$$

$$\frac{P[pc] = \texttt{END}}{P \vdash (pc, \mathcal{R}, *) \xrightarrow{e} (\mathcal{R}, *)} \qquad \frac{P[pc] = I \qquad I \notin \{\texttt{END}, \texttt{BR}\} \qquad (I, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}', *')}{P \vdash (pc, \mathcal{R}, *) \xrightarrow{c} (pc+1, \mathcal{R}', *')}$$

Fig. 4. Semantics of control-flow operations.

3.1 Polyhedra

A *polyhedron* p denotes a set of points in a $\mathbb{Z}$ vector space bounded by linear constraints (equalities or inequalities). More formally, let $|S|$ denote the cardinality of set S. Let C_n denote the set of linear constraints in $\mathbb{Z}^n$ on the set of variables $\mathcal{V}_n$, where $|\mathcal{V}_n| = n$. Then $\langle c_1, c_2, ..., c_m \rangle$ denotes the polyhedron p consisting of all the vectors in $\mathbb{Z}^n$ that satisfy constraints c_1, c_2, ..., c_m, where $c_i \in C_n$ for $1 \leq i \leq m$ (n and m are unrelated). We denote $dim(p) = n$ the *dimension* of p. In the rest of the paper, the term *variable* implicitly refers to polyhedron variables. We denote:

- $\mathcal{P}$ the set of polyhedra;
- $s \in p$ when s (with $s \in \mathbb{Z}^{dim(p)}$) satisfies the constraints of polyhedron p;
- $p \sqsubseteq_\diamond p'$ iff $\forall s \in p, s \in p'$;
- $p'' = p \sqcup_\diamond p'$ the *convex hull* of p and p';
- $p'' = p \sqcap_\diamond p'$ the union of the constraints of p and p';
- $vars(p)$ the set of variables of p, where $|vars(p)| = dim(p)$ by definition;
- $proj(p, x_1 \dots x_k)$ the projection of p on space $x_1 \dots x_k$, with $k < |dim(p)|$;
- $p[x_i/x_j]$ the substitution of variable x_j by x_i in p.

3.2 Abstract States

In polyhedral analysis of source code, variables of the polyhedra are related to variables of the source code. In our case, polyhedra variables are related to registers and memory contents. We use the term *data location* to refer indistinctly to registers or memory addresses. Let $\mathcal{V}$ denote the set of polyhedra variables.

The set of abstract states POLYMAP is defined as $\mathcal{A} = \mathcal{P} \times (R \nrightarrow \mathcal{V}) \times (\mathcal{V} \nrightarrow \mathcal{V})$. An abstract state $a \in \mathcal{A}$, with $a = (p, \mathcal{R}^\sharp, *^\sharp)$, consists of a polyhedron p, a *register mapping* $\mathcal{R}^\sharp$ and an *address mapping* $*^\sharp$. We have $\mathcal{R}^\sharp(r) = v$ iff variable v represents the value of register r in p. We have $*^\sharp(x_1) = x_2$ iff variable x_2 represents the value at the memory address represented by variable x_1. We denote $vars_R(p)$ the codomain of $\mathcal{R}^\sharp$ (i.e. register content variables), $vars_A(p)$ the domain of $*^\sharp$ (i.e. address variables) and $vars_C(p)$ the codomain of $*^\sharp$ (i.e. address content variables). Sets $vars_R(p)$, $vars_A(p)$ and $vars_C(p)$ are disjoint and are all subsets of $vars(p)$.

Example 1. In the following abstract state, register r_0 contains value 2, and address 2 contains value 1:

$$(\{x_1 = 2, x_2 = x_1, x_3 = 1\}, \{r_0 : x_1\}, \{x_2 : x_3\})$$

The usual operators on the abstract domain (inclusion, join and widening), and its least and greatest elements are presented in Section 4.4.

3.3 Aliasing

In a general sense, *aliasing* occurs in a program when a data location can be accessed through several symbolic names. As we will see in Section 4, aliases play an important role in our analysis. In fact, we introduce mechanisms that prevent their occurrence in the abstract state (see Section 4.2), so as to simplify the analysis. We define the *aliasing relation* between two variables x_1 and x_2 of a polyhedron p as follows:

- Cannot alias: whenever $\langle x_1 = x_2 \rangle \cap p = \emptyset$;
- May alias: whenever $\langle x_1 = x_2 \rangle \cap p \neq \emptyset$;
- Must alias, denoted $x_1 \equiv x_2$: whenever $p \sqsubseteq_\diamond \langle x_1 = x_2 \rangle$.

The aliasing relation between a register r and a variable x is defined by the aliasing relation between $\mathcal{R}^\sharp(r)$ and x. Similarly, the aliasing relation between two registers r_1, r_2 is defined by the aliasing relation between $\mathcal{R}^\sharp(r_1)$ and $\mathcal{R}^\sharp(r_2)$.

To avoid ambiguities with notations on constraints, let $same(x_1, x_2)$ denote the fact that x_1 and x_2 are the same polyhedron variables (not just equivalent variables). There is no need to check register aliases, because a single register cannot be mapped to two different variables ($\mathcal{R}^\sharp$ is a function). The absence of aliases can thus be stated as follows.

Definition 1. *Let $s = (p, \mathcal{R}^\sharp, *^\sharp)$ be an abstract state. We say that s is* alias free *iff:*

$$\forall x_1, x_2 \in vars_A(p), x_1 \equiv x_2 \Rightarrow same(x_1, x_2)$$

4 Computing abstract states

Our analysis follows the abstract interpretation framework proposed in [4], adapted to our setting with non-local control-flow, following the technique proposed in Astrée [5] and MOPSA [6]. An important singularity of our analysis is that polyhedral variables are progressively created or removed during the analysis. Whenever a new polyhedron variable is introduced, we assume it is a fresh variable that has never been used at any other point during the analysis.

4.1 Interpretation algorithm

We use $(p', [r_i : x_i], [x_j : x_k])(\cdot)$ as a shorthand for $\lambda(p, \mathcal{R}^\sharp, *^\sharp).(p \sqcap_\diamond p', \mathcal{R}^\sharp[r_i : x_i], *^\sharp[x_j : x_k])$, and denote $-$ when a state component remains unchanged. Procedures to compute the join ($\sqcup$), widening ($\triangledown$) and *antialias* of abstract states, and the transfer function $(I)^\sharp$ of instruction I are detailed in the remainder of this section. The complete interpretation procedure is described in Algorithm 1.

It applies to a program P of `MEMP`. During the interpretation, we keep a subset L of labels of interest. Abstract values are stored in a map M from labels to abstract values. We assume that loop header labels L_W of P have previously been identified using an existing analysis (e.g. Tarjan's algorithm [7]). Figure 5 reports a running example of this analysis, that will be used throughout the rest of the section.

Algorithm 1 INTERPRET(P)

```
 1: procedure UPDATE(ℓ, a, L)                          ▷ Auxiliary procedure
 2:     a ← antialias(a)
 3:     if ℓ ∈ L_W then                                ▷ Check if l is a loop header
 4:         new ← M[l] ▽ (M[l] ⊔ a)
 5:     else
 6:         new ← M[ℓ] ⊔ a
 7:     end if
 8:     if new ⋢ M[ℓ] then                             ▷ Abstract value for ℓ changed, propagate
 9:         M[ℓ] ← new; L ← L ∪ ℓ
10:     end if
11: end procedure
12:
13: for all (ℓ, I) ∈ P do                             ▷ Start of main procedure
14:     M[ℓ] ← ⊥                                      ▷ Begin with empty abstract states
15: end for
16: M[ℓ_1] ← ⊤; L ← {ℓ_1}                             ▷ Program starting label
17: while L ≠ ∅ do                                    ▷ Fixpoint iteration
18:     Pick and remove ℓ from L
19:     with BR r ℓ'
20:         UPDATE(ℓ', (⟨r = 0⟩, −, −)(M[ℓ]), L)      ▷ Branching case
21:         UPDATE(ℓ + 1, (−, −, −)(M[ℓ]), L)         ▷ Not branching case
22:     with END
23:         skip
24:     with _
25:         UPDATE(ℓ + 1, ((P[ℓ])♯)(M[ℓ]), L)         ▷ Abstract semantics of I
26: end while
27: return M
```

4.2 Anti-aliasing

Whenever updating an abstract state, we immediately remove aliases (line 2), because the absence of aliases significantly simplifies the analysis in places where we need to check the equivalence of two variables (`LOAD`, `STORE`, $\sqcup$ and $\triangledown$). In practice, aliases are introduced when encountering a conditional branching (see Section 4.3). We remove an alias using procedure *antialias*, which relies on the procedure *Merge* defined below. It is based on the following observation: if two

```
1: RAND r0          5: ADD r3 r0 r1      9: STORE r3 r2
2: RAND r7          6: STORE r3 r1      10: LOAD r6 r3
3: SET r1 4         7: SUB r5 r7 r1     11: END
4: SET r2 5         8: BR r5 10
```

Label	*Polyhedron*	*Registers*	*Memory*
5	$p_1 = \langle x_1 = 4, x_2 = 5\rangle$	$\mathcal{R}_1^\sharp = \{r_0 : x_0, r_1 : x_1, r_2 : x_2, r_7 : x_7\}$	
6	$p_2 = p_1 \sqcap_\diamond \langle x_3 = x_0 + x_1\rangle$	$\mathcal{R}_2^\sharp = \mathcal{R}_1^\sharp[r_3 : x_3]$	
7	$p_3 = p_2 \sqcap_\diamond \langle x_4 = x_3, x_5 = x_1\rangle$	$\mathcal{R}_2^\sharp$	$*_1^\sharp = \{x_4 : x_5\}$
8	$p_4 = p_3 \sqcap_\diamond \langle x_8 = x_7 - x_1\rangle$	$\mathcal{R}_3^\sharp = \mathcal{R}_2^\sharp[r_5 : x_8]$	$*_1^\sharp$
10 (from 9)	$p_5 = p_4 \sqcap_\diamond \langle x_9 = x_2\rangle$	$\mathcal{R}_3^\sharp$	$*_2^\sharp = \{x_4 : x_9\}$
$10'$ (from 8)	$p_6 = p_4 \sqcap_\diamond \langle x_8 = 0\rangle$	$\mathcal{R}_3^\sharp$	$*_1^\sharp$
$unify(10, 10')$	$p_7 = p_6[x_9/x_5]$	$\mathcal{R}_3^\sharp$	$*_3^\sharp = \{x_4 : x_9\}$
$10 \sqcup 10'$	$p_8 = p_2 \sqcap_\diamond \langle x_4 = x_3, x_8 = x_7 - x_1, x_1 \leq x_9 \leq x_2\rangle$	$\mathcal{R}_3^\sharp$	$*_3^\sharp$
11	$p_8 \sqcap_\diamond \langle x_{10} = x_9\rangle$	$\mathcal{R}_3^\sharp[r_6 : x_{10}]$	$*_3^\sharp$

Fig. 5. Running example of analysis

addresses are equal, then the values stored at these addresses must be equal too. Let x_1, x_2 be two variables of $vars_A(p)$ such that: $\neg same(x_1, x_2) \wedge x_1 \equiv x_2$.

$$Merge((p, \mathcal{R}^\sharp, *^\sharp), x_1, x_2) = (p', \mathcal{R}^\sharp, *^{\sharp'}) \quad \begin{array}{l} \text{with } p' = (p[x_1/x_2])[*^\sharp(x_1)/\, *^\sharp(x_2)] \\ \text{and } *^{\sharp'} = *^\sharp \setminus (x_2 : *^\sharp(x_2)) \end{array}$$

Function $antialias : \mathcal{A} \to \mathcal{A}$ applies *Merge* for each pair of distinct equivalent address variables of an abstract state.

Example 2. In state a below, address x_2 is an alias on address x_1. Thus, x_4 must be equal to x_3, so $Merge(a, x_1, x_2)$ replaces x_2 by x_1 and x_4 by x_3. In the result, x_3 is constrained by the original constraints of x_3 and x_4, and the memory mapping $x_2 : x_4$ is discarded.

$$a = (\langle x_1 = x_2, x_3 \geq 4, x_4 \leq 5\rangle,\ -,\ *^\sharp = \{x_1 : x_3, x_2 : x_4\})$$
$$Merge(a, x_1, x_2) = (\langle 4 \leq x_3 \leq 5\rangle,\ -,\ *^{\sharp'} = \{x_1 : x_3\})$$

4.3 Transfer functions

We now define the constraints generated for the analysis of each instruction of our language. We denote $(I)^\sharp : \mathcal{A} \to \mathcal{A}$ the transfer function of instruction I.

Binary operation If the relation $r_1 = \mathrm{OP}^c(r_2, r_3)$ is linear, we map the target register to a new variable, subject to the corresponding linear constraint in the

polyhedron. The memory mapping is unchanged. Otherwise, the target register is mapped to a new unconstrained variable.

$$(\texttt{OP r}_1\ \texttt{r}_2\ \texttt{r}_3)^\sharp = \begin{cases} (\langle x = \mathrm{OP}^c(\mathcal{R}^\sharp(r_2), \mathcal{R}^\sharp(r_3))\rangle,\ [r_1 : x],\ -)(\cdot) & \text{if } \mathit{linear}(\mathrm{OP}^c) \\ (-,\ [r_1 : x],\ -)(\cdot) & \text{otherwise} \end{cases}$$

Example 3. In Figure 5, at label 6 (i.e. the label immediately following the `ADD` operation) we introduce the constraint $x_3 = x_0 + x_1$ and the register mapping $\mathcal{R}_1^\sharp(r_3) = x_3$.

Set The impact of the immediate load instruction is straightforward:

$$(\texttt{SET r}_1\ \texttt{c})^\sharp = (\langle x = c\rangle,\ [r_1 : x], -)(\cdot)$$

Rand The random instruction maps a register to an unconstrained variable:

$$(\texttt{RAND r}_1)^\sharp = (-,\ [r_1 : x], -)(\cdot)$$

Load If the input state contains a memory address variable that is equivalent to the load address (note that for alias free states, if such a variable exists, it is unique), then in the output state the value of the destination register is the value of the memory value mapped to this address. Otherwise, the value of the destination register is undefined:

$$(\texttt{LOAD r}_1\ \texttt{r}_2)^\sharp = \begin{cases} (\langle x = *^\sharp(a)\rangle,\ [r_1 : x],\ -)(\cdot) & \text{if } a \equiv r_2 \\ (-,\ [r_1 : x],\ -)(\cdot) & \text{otherwise} \end{cases}$$

Example 4. In Figure 5, at label 10 we have $x_4 \equiv r_3$ and $*^\sharp(x_4) = x_9$, so at label 11 we introduce the constraint $x_{10} = x_9$ and the mapping $\mathcal{R}_3^\sharp[r_6] = x_{10}$.

Store Again, we need to consider the impact of aliases. If there exists an address variable equivalent to the target register, then there already exists a memory mapping for this address. The previous content at this address is replaced by the content of the source register (see *Replace* below). Otherwise, we create a new memory mapping (see *Create* below). An alias free state contains at most one address variable that must-alias with r_1. It may however contain several may-alias address variables a'. For each such a', this means that a' either equals r_1, which requires a *Replace*, or is different from r_1, which has no impact. We apply operator $\sqcup$ on both cases to manage this uncertainty, and add the constraints for each may-alias address (see *May* below).

$$(\texttt{STORE r}_1\ \texttt{r}_2)^\sharp = \begin{cases} \lambda s.\mathit{Replace}(a)(\mathit{May}(s)) & \text{if } \exists a \in \mathit{vars}_A(p), a \equiv r_1 \\ \lambda s.\mathit{Create}(\mathit{May}(s)) & \text{otherwise} \end{cases}$$

With ($\bigcirc$ denotes function composition):

$$\begin{aligned} \mathit{Replace}(a) &= (\langle x = \mathcal{R}^\sharp(r_2)\rangle,\ -,\ [a : x])(\cdot) \\ \mathit{Create} &= (\langle x_i = \mathcal{R}^\sharp(r_1), x_j = \mathcal{R}^\sharp(r_2)\rangle, -, [x_i : x_j])(\cdot) \\ \mathit{May} &= \bigcirc_{\{a \in A \mid a \text{ may-alias } r_1\}} \lambda s.(\mathit{Replace}(a)(s) \sqcup s) \end{aligned}$$

Example 5. In Figure 5, at label 7, we create a new memory mapping $*_1^\sharp(x_4) = x_5$ and we introduce the constraints $x_4 = x_3$, $x_5 = x_1$.

Example 6. In Figure 5, at label 10, when coming from label 9, we replace a previous mapping, x_4 is mapped to x_9 (instead of x_5 previously), and we introduce the constraint $x_9 = x_2$.

Branching In Algorithm 1, when branching to a target label (ℓ') the branching condition holds ($r = 0$). We add no constraint for the otherwise case because it cannot be encoded using a linear relation.

Example 7. In Figure 5, at label 10, when coming from label 6, we add the constraint $x_8 = 0$.

4.4 Abstract domain operators, least and greatest elements

Our analysis introduces new variables and removes old ones as it progresses. There is no predefined correspondence between variables and data locations, because the set of data locations used by the program is unknown a priori. As a consequence, it may happen that two abstract states use different variables to designate the same data location. This implies that to compare two states we first need to check whether some variables of the two states actually correspond to the same data location. This verification relies on a *unification* procedure, presented below. Unification is used for inclusion testing, and also in the join and widening operators.

Unification Unification checks for the equivalence of two variables in two polyhedra, p_1 and p_2. Intuitively, we try to express each variable as a linear expression of a well-chosen set of variables to conveniently check their equivalence.

Let $V_c = \mathit{vars}(p_1) \cap \mathit{vars}(p_2)$ and $p' = \mathit{proj}(p_1, V_c) \sqcup_\diamond \mathit{proj}(p_2, V_c)$. We denote $\mathit{npiv}(p')$ the set of non-pivot variables discovered by Gauss-Jordan elimination performed on the system of equality constraints of p' (we exclude inequalities). Then, $\mathit{npiv}(p')$ is such that, in p':

- no variable in $\mathit{npiv}(p')$ is equivalent to a linear expression of other variables of $\mathit{npiv}(p')$;
- each variable in $\mathit{vars}(p') \backslash \mathit{npiv}(p')$ is equivalent to a linear expression of variables from $\mathit{npiv}(p')$.

Let $\mathit{linexpr}(x, p_1, \mathit{npiv}(p'))$ denote the linear expression representation of variable $x \in \mathit{vars}(p_1)$ in terms of variables in $\mathit{npiv}(p')$, represented as the vector of the linear expression coefficients. Let C' be the constraint system of $\mathit{proj}(p_1, x \cup \mathit{npiv}(p'))$. If C' contains an equality constraint involving x, then computing $\mathit{linexpr}(x, p_1, \mathit{npiv}(p'))$ is straightforward. Otherwise, the empty vector is returned. If several (non-equivalent) equality constraints appear, we arbitrarily pick one. Note that, even though our unification can miss equivalent variables, this does not jeopardize the soundness of the analysis (see Section 5.3 and in particular Lemma 3).

Algorithm 2 describes our unification procedure. We directly modify the second state to unify it with the first one. First, we compute set of non-pivot variables (line 4). Then, we check for the equivalence of address variables according to their linear expression representation, and we perform variable substitutions in p'_2, $\mathcal{R}_2^{\sharp'}$ and $*_2^{\sharp'}$ in case of equivalence (line 8). Register unification is simpler, we just replace the bindings in $\mathcal{R}_2^{\sharp'}$ by those of $\mathcal{R}_1^{\sharp}$ (line 12).

Algorithm 2 $\mathit{unify}((p_1, \mathcal{R}_1^{\sharp}, *_1^{\sharp}), (p_2, \mathcal{R}_2^{\sharp}, *_2^{\sharp}))$

1: $(p'_2, \mathcal{R}_2^{\sharp'}, *_2^{\sharp'}) \leftarrow (p_2, \mathcal{R}_2^{\sharp}, *_2^{\sharp})$
2: $V_c \leftarrow \mathit{vars}(p_1) \cap \mathit{vars}(p_2)$ ▷ common variables
3: $p' \leftarrow \mathit{proj}(p_1, V_c) \sqcup_\diamond \mathit{proj}(p_2, V_c)$
4: $B \leftarrow \mathit{npiv}(p')$
5: **for all** $(x_i, x_j) \in \mathit{vars}_A(p_1) \times \mathit{vars}_A(p_2)$ **do**
6: $\quad v_i = \mathit{linexpr}(x_i, p_1, B)$; $v_j = \mathit{linexpr}(x_j, p_2, B)$
7: $\quad$ **if** $v_i \neq []$ and $v_j \neq []$ and $v_i = v_j$ **then** ▷ variables are equivalent
8: $\quad\quad$ Replace x_j by x_i and $*^{\sharp}(x_j)$ by $*^{\sharp}(x_i)$ in p'_2, $\mathcal{R}_2^{\sharp'}$, and $*_2^{\sharp'}$
9: $\quad$ **end if**
10: **end for**
11: **for all** $r \in \mathit{Dom}(\mathcal{R}_1^{\sharp}) \cap \mathit{Dom}(\mathcal{R}_2^{\sharp})$ **do** ▷ variables are trivially equivalent
12: $\quad$ Replace $\mathcal{R}_2^{\sharp}(r)$ by $\mathcal{R}_1^{\sharp}(r)$ in p'_2, $\mathcal{R}_2^{\sharp'}$, and $*_2^{\sharp'}$
13: **end for**
14: **return** $(p'_2, \mathcal{R}_2^{\sharp'}, *_2^{\sharp'})$

Example 8. In Figure 5, when computing $\mathit{unify}(10, 10')$, s_1 corresponds to the state of 10 and s_2 to the state of $10'$. A possible set of non-pivot variables is $\{x_0, x_7\}$. In s_1 (and in s_2), we have $x_4 - x_0 + 0 \cdot x_7 - 4 = 0$, so $\mathit{linexpr}(x_4) = [1;\ -1;\ 0;\ -4]$ (corresponding, respectively, to the coefficients of x_4, x_0, x_7, and the constant). Since $*_2^{\sharp}(x_4) = x_9$ (in s_1) and $*_1^{\sharp}(x_4) = x_5$ (in s_2), we replace x_5 by x_9 in s_2.

Inclusion Let us now define formally the partially ordered set $(\mathcal{A}, \sqsubseteq)$. Given two functions f and g, we denote $f \subseteq g$ when $\mathit{Dom}(f) \subseteq \mathit{Dom}(g)$ and $\forall x \in \mathit{Dom}(f) : f(x) = g(x)$. Introducing new mappings in $\mathcal{R}^{\sharp}$ or $*^{\sharp}$ (i.e. enlarging

their domains) actually removes feasible concrete states, thus we define abstract states inclusion as follows (see lemma 4 for more details):

Definition 2. *Let* $a_1 = (p_1, \mathcal{R}_1^\sharp, *_1^\sharp)$ *and* $a_2 = (p_2, \mathcal{R}_2^\sharp, *_2^\sharp)$*. The ordering operator* $\sqsubseteq$ *is defined as follows:*

$$a_1 \sqsubseteq a_2 \Leftrightarrow p_1' \sqsubseteq_\diamond p_2 \wedge \mathcal{R}_2^\sharp \subseteq \mathcal{R}_1^{\sharp'} \wedge *_2^\sharp \subseteq *_1^{\sharp'}$$
$$\text{with } ({p_1}', \mathcal{R}_1^{\sharp'}, *^{\sharp'}) = \mathit{unify}(a_2, a_1)$$

There exists several equivalent representations of the greatest and least elements of $(\mathcal{A}, \sqsubseteq)$. We define them as follows:

Definition 3. *The greatest element of* $(\mathcal{A}, \sqsubseteq)$ *is denoted* $\top$*, with* $\top = (\langle\rangle, \emptyset, \emptyset)$.

Definition 4. *The least element of* $(\mathcal{A}, \sqsubseteq)$ *is denoted* $\bot$ *and defined as* $\bot = (p_\bot, \mathcal{R}_\bot^\sharp, *_\bot^\sharp)$*, where* $p_\bot$ *is the empty polyhedron and* $\mathcal{R}_\bot^\sharp$*,* $*_\bot^\sharp$ *are such that every data location is mapped to a variable.*

Join Algorithm 3 describes our join procedure. It unifies the input states (line 1), then computes the convex hull on the unified states (line 2). Then, if a memory location or register is bound in one input state and unbound in the other, it is unbound in the result state.

Algorithm 3 $(p_1, \mathcal{R}_1^\sharp, *_1^\sharp) \sqcup (p_2, \mathcal{R}_2^\sharp, *_2^\sharp)$

```
1: (p2', R2#', *2#') = unify((p1, R1#, *1#), (p2, R2#, *2#))
2: p ← p1 ⊔◇ p2'
3: R# ← ∅; *# ← ∅
4: for all r ∈ Dom(R1#') do
5:     if R1#(r) = R2#'(r) then R#(r) ← R1#(r) end if
6: end for
7: for all a ∈ Dom(*1#') do
8:     if *1#(a) = *2#'(a) then *#(a) ← *1#(a) end if
9: end for
10: return (p, R#, *#)
```

Example 9. In Figure 5, when computing $10 \sqcup 10'$, we obtain identical register and memory mappings for 10 and $\mathit{unify}(10, 10')$. The convex hull $p_5 \sqcup_\diamond p_7$ groups the constraints on x_9 ($x_1 \leq x_9 \leq x_2$) and lifts those on x_8.

Widening Due to the presence of loops, the widening operator $\triangledown$ is used to ensure that our analysis reaches a fixpoint. $\triangledown$ is defined just like $\sqcup$, except that we use a polyhedra widening operator $\triangledown_\diamond$ in place of $\sqcup_\diamond$.

4.5 Loop bounds

To compute loop bounds, for each loop header label ℓ we create a "virtual" register r_ℓ, to count the number of iterations of ℓ. We instrument the program so that the register r_ℓ is set to 0 when entering loop ℓ, and incremented at each iteration of ℓ (which is fairly classic, see e.g. [8]).

Finally, let P a program of `MEMP` and $M = \mathit{interpret}(P)$. Let ℓ_e be the label of instruction `END` in P. Let $(p_f, \mathcal{R}^\sharp_f, *^\sharp_f) = M[\ell_e]$. Then the loop bound for a loop header ℓ is computed as $\max(p_f, \mathcal{R}^\sharp_f[r_\ell])$ (where $\max(p, x)$ denotes the greatest value of variable x satisfying the constraints of p).

5 Soundness

In this section, we prove the soundness of our analysis. We first establish a set of important lemmas on our abstract domain operators, and then prove soundness with respect to the concretization function.

5.1 Join

Operator $\sqcup$ is not commutative. We establish that it does however compute an upper bound of its operands, with respect to our inclusion definition (Lemma 1). The proof is based on two auxiliary properties on mapping inclusions:

Property 1. Let $a_1 = (p_1, \mathcal{R}^\sharp_1, *^\sharp_1)$, $a_2 \in \mathcal{A}$, $a_3 = (p_3, \mathcal{R}^\sharp_3, *^\sharp_3) = a_1 \sqcup a_2$. We have:

$$(p_1 \sqsubseteq_\diamond p_3) \wedge (\mathcal{R}^\sharp_3 \subseteq \mathcal{R}^\sharp_1) \wedge (*^\sharp_3 \subseteq *^\sharp_1)$$

Proof. Considering Algorithm 3: $(p_1 \sqsubseteq_\diamond p_3)$ follows from line 2, $(\mathcal{R}^\sharp_3 \subseteq \mathcal{R}^\sharp_1)$ from line 5, and $(*^\sharp_3 \subseteq *^\sharp_1)$ from line 8. □

Property 2. Let a_1, a_2, $a'_1 \in \mathcal{A}$, with $a'_1 = (p'_1, \mathcal{R}^{\sharp'}_1, *^{\sharp'}_1) = \mathit{unify}(a_2, a_1)$. Then:

$$(\mathcal{R}^\sharp_2 \subseteq \mathcal{R}^\sharp_1) \wedge (*^\sharp_2 \subseteq *^\sharp_1) \Rightarrow (\mathcal{R}^\sharp_2 \subseteq \mathcal{R}^{\sharp'}_1) \wedge (*^\sharp_2 \subseteq *^{\sharp'}_1)$$

Proof. Obvious from Algorithm 2.

Lemma 1. *Let a_1, $a_2 \in \mathcal{A}$. We have:* $(a_1 \sqsubseteq a_1 \sqcup a_2) \wedge (a_2 \sqsubseteq a_1 \sqcup a_2)$.

Proof. Polyhedron inclusion follows from the polyhedra join operator. We must also prove the inclusion of register and memory mappings (after unification).

Case for a_1 follows from Properties 1 and 2. Concerning the case for a_2, let $a_3 = a_1 \sqcup a_2$. When computing a_3, a variable v of a_2 falls into one of three categories: 1) v is also in $\mathit{vars}(p_1)$, it remains in a_3; 2) v is equivalent to a variable v_1 of $\mathit{vars}(p_1)$, it is replaced by v_1 in a_3 (Algorithm 2, line 8); 3) otherwise, it is removed (Algorithm 3). Then, let $a'_2 = \mathit{unify}(a_3, a_2)$. When computing a'_2, variables that fell in category 2 at the previous step (when computing a_3) will be replaced by their equivalent in a_3, because they fall again in category 2. Thus we obtain $\mathcal{R}^\sharp_3 \subseteq \mathcal{R}^{\sharp'}_2$, $*^\sharp_3 \subseteq *^{\sharp'}_2$, which concludes the proof. □

5.2 Widening

Lemma 2 establishes that operator $\triangledown$ is indeed a widening operator.

Property 3. Let $a_1, a_2 \in \mathcal{A}$. We have: $(a_1 \sqcup a_2) \sqsubseteq (a_1 \triangledown a_2)$.

Proof. The property holds because $\sqcup$ and $\triangledown$ use the same unification procedure, and because we assume that $\triangledown_\diamond$ is a valid polyhedra widening operator. □

Property 4. Let $a_1 = (p_1, \mathcal{R}_1^\sharp, *_1^\sharp)$, $a_2 \in \mathcal{A}$, $a_3 = (p_3, \mathcal{R}_3^\sharp, *_3^\sharp) = a_1 \triangledown_\diamond a_2$. We have: $(p_1 \sqsubseteq_\diamond p_3) \wedge (\mathcal{R}_3^\sharp \subseteq \mathcal{R}_1^\sharp) \wedge (*_3^\sharp \subseteq *_1^\sharp)$

Proof. Same as for Property 1.

Property 5. Let $(b_n)_{n\in\mathbb{N}}$ be a non decreasing infinite sequence in $\mathcal{A}$. Then, the sequence $a_0 = b_0$ and $a_{n+1} = a_n \triangledown b_{n+1}$ converges in a finite number of steps.

Proof. Thanks to Property 4, and considering that there is a finite quantity of data locations, there exists $N \in \mathbb{N}$ such that for all $i > N$, $\mathcal{R}_{i+1}^\sharp = \mathcal{R}_i^\sharp$ and $*_{i+1}^\sharp = *_i^\sharp$. Thus, $a_{i+1} = (p_i \triangledown_\diamond q_{i+1}, \mathcal{R}_i^\sharp, *_i^\sharp)$, where q_{i+1} is the polyhedron of b_{i+1} and p_i that of a_i.

Assuming that $\triangledown_\diamond$ is a valid polyhedra widening operator, there exists $m > N$ such that $p_{m+1} = p_m$. Since $m > N$ we also have $\mathcal{R}_{m+1}^\sharp = \mathcal{R}_m^\sharp$ and $*_{m+1}^\sharp = *_m^\sharp$, which concludes the proof. □

Lemma 2. *Operator $\triangledown$ is a widening operator.*

Proof. Follows from Properties 3 and 5.

5.3 Concrete and abstract states

Let $\mathcal{C} = ((R \nrightarrow \mathbb{Z}) \times (\mathbb{Z} \nrightarrow \mathbb{Z}))$ denote the set of concrete states (pairs of registers contents and memory contents). Data locations are mapped to values in a concrete state, while they are mapped to polyhedra variables in the abstract state. The concretization function γ relates data location values to data location variables as follows:

Definition 5. *Let $a = (p, \mathcal{R}^\sharp, *^\sharp)$ be an abstract state. The concretization function γ is defined as follows:*

$$\gamma : \mathcal{A} \longrightarrow \mathcal{P}(\mathcal{C})$$

$$(p, \mathcal{R}^\sharp, *^\sharp) \longmapsto \Big\{ (*, \mathcal{R}) \mid \exists f : Dom(*^\sharp) \to Dom(*),$$

$$\left(\bigsqcap_{r \in Dom(\mathcal{R}^\sharp)}^{\diamond} \langle \mathcal{R}^\sharp(r) = \mathcal{R}(r) \rangle \quad \bigsqcap_\diamond \bigsqcap_{x \in Dom(*^\sharp)}^{\diamond} \langle x = f(x), *^\sharp(x) = *(f(x)) \rangle \right) \sqsubseteq_\diamond p \Big\}$$

More intuitively, we build a polyhedron p' with the following constraints: 1) register values of the concrete state ($\mathcal{R}(r)$) must be equal to the corresponding variable in the abstract state ($\mathcal{R}^\sharp(r)$); 2) we try to find a function f that maps address variables to addresses ($x = f(x)$), then the content of each address variables ($*^\sharp(x)$) must be equal to the memory value ($*(f(x))$). If $p' \sqsubseteq_\diamond p$ then the concrete state satisfies the constraints of p and belongs to the concretization.

Example 10.

$$\begin{aligned} a &= (\{1 \leq x_1 \leq 2, x_2 = x_1, x_3 = 1\}, \{r_0 : x_1\}, \{x_2 : x_3\}) \\ \gamma(a) &= \{(\{r_0 = 1\}, \{*(1) = 1\}\}), && (f(x_2) = 1) \\ &\quad (\{r_0 = 2\}, \{*(2) = 1\})\} && (f(x_2) = 2) \end{aligned}$$

Let $\xrightarrow{c}{}^*$ denote the transitive closure of $\xrightarrow{c}$. The soundness of our abstract interpretation is established as follows:

Theorem 1. *Let P be a `MEMP` program. Let $M = Interpret(P)$. Then, for any concrete state s_{init}: $(P \vdash (l_1, s_{init}) \xrightarrow{c}{}^* (\ell, s)) \Longrightarrow (s \in \gamma(M[\ell]))$*

Proof. The proof of soundness follows from the structure of Algorithm 1, and from the following lemmas, which establish the soundness of each operator used in the algorithm.

Lemma 3. *Let a_1, $a_2 \in \mathcal{A}$. We have: $\gamma(a_1) = \gamma(unify(a_2, a_1))$.*

Proof. Let $a_1' = unify(a_2, a_1)$. Since we assume that a_1 and a_2 are alias free (recall Section 4.2), any two non-equivalent variables in a_1 are also replaced by non-equivalent variables in a_1' (or unchanged). Thus a_1' is a simple renaming of a_1, and so a_1 and a_1' have the same concretization. □

Lemma 4. *Let a_1, $a_2 \in \mathcal{A}$. We have: $(a_1 \sqsubseteq a_2) \Rightarrow \gamma(a_1) \subseteq \gamma(a_2)$*

Proof. Let $s \in \gamma(a_1)$. Let $a_1' = (p_1', \mathcal{R}_1^{\sharp'}, *_1^{\sharp'}) = unify(a_2, a_1)$. From Lemma 3, $s \in \gamma(a_1')$, thus there exists a function f for s satisfying the property of Definition 5 with $a = a_1$. Now, assume that $p_1' \sqsubseteq_\diamond p_2 \wedge \mathcal{R}_2^\sharp \subseteq \mathcal{R}_1^{\sharp'} \wedge *_2^\sharp \subseteq *_1^\sharp$ (i.e. $a_1 \sqsubseteq a_2$). Then there exists a function f' for s that satisfies Definition 5, with $a = a_2$: just take f' such that it is the restriction of f to $Dom(*_2^\sharp)$. So $s \in \gamma(a_2)$. □

Lemma 5. *Let a_1, $a_2 \in \mathcal{A}$. We have: $\gamma(a_1) \cup \gamma(a_2) \subseteq \gamma(a_1 \sqcup a_2)$.*

Proof. From Lemma 1 and Lemma 4.

Lemma 6. *Let a_1, $a_2 \in \mathcal{A}$. We have: $\gamma(a_1) \cup \gamma(a_2) \subseteq \gamma(a_1 \triangledown a_2)$*

Proof. From Lemma 5, Lemma 4 and Property 3.

Lemma 7. *Let $a \in \mathcal{A}$. We have: $\gamma(a) \subseteq \gamma(antialias(a))$*

Proof. Let $(p, \mathcal{R}^\sharp, *^\sharp) = a$. Let x_1, $x_2 \in vars_A(p)$ be such that $\neg same(x_1, x_2) \wedge x_1 \equiv x_2$. Then:

$$\begin{aligned} s \in \gamma(a) &\Rightarrow s \in \gamma(p \sqcap_\diamond \langle x_1 = x_2, *^\sharp(x_1) = *^\sharp(x_2)\rangle, \mathcal{R}^\sharp, *^\sharp) \\ &\Rightarrow s \in \gamma((p[x_1/x_2])[*^\sharp(x_1)/\, *^\sharp(x_2)], \mathcal{R}^\sharp, *^\sharp) \\ &\Rightarrow s \in \gamma(Merge(a_1, x_1, x_2)) \end{aligned}$$

The soundness of *antialias* follows. □

Lemma 8. *Let P be a* `MEMP` *program. Let $M = Interpret(P)$. Then, for all labels ℓ, ℓ' of P:*

$$(P \vdash (\ell, \mathcal{R}, *) \xrightarrow{c} (\ell', \mathcal{R}', *')) \Longrightarrow ((\mathcal{R}, *) \in \gamma(M[\ell]) \Rightarrow (\mathcal{R}', *') \in \gamma(M[\ell']))$$

Proof. Trivially follows from the formal semantics and from the definition of transfer functions, except for `STORE`. Let $a' = (p', \mathcal{R}^{\sharp'}, *^{\sharp'}) = (\texttt{STORE}\ r_1\ r_2)^\sharp(a)$. The proof follows from noting that: 1) Both in the *Create* and *Replace* cases, we obtain $*^{\sharp'}(\mathcal{R}^{\sharp'}(r_1)) = \mathcal{R}^{\sharp'}(r_2)$, which is coherent with the formal semantics of `STORE`; 2) The soundness of *May* follows from the soundness of $\sqcup$ and *Replace*. □

Lemma 9. *Algorithm 1 terminates.*

Proof. Because $\triangledown$ is applied on loop headers and $\triangledown$ is a valid widening operator. □

6 Related works

Abstract interpretation using polyhedra has been first described in [4]. Static analysis tools such as Astree [5], Frama-C [9] or PAGAI [10] use various abstract domains (including polyhedra) to generate invariants for proving various properties, such as the absence of array out-of-bounds accesses for instance.

While Astree and Frama-C work on the Abstract Syntax Tree, PAGAI processes LLVM Intermediate Representation (IR). Compared to our approach, both the AST and LLVM representations are closer to the source code, and contain information on variables and their types, and also a precise control flow. This makes the analysis easier to design, but less precise as far as WCET is concerned.

Several other abstract domains other that polyhedra, capable of representing linear constraints between variables, have been proposed, such as for instance [11,12,13]. Choosing the most appropriate domain boils down to a trade-off between the execution time and the precision of the analysis. In our work we chose the polyhedra domain and thus favored precision. However, we think that it would be simple to adapt our work to another domain (e.g. to reduce analysis time), because our computation of memory and register mappings does not depend on how constraints between variables are represented and computed.

Several works address static analysis of binary code [14,15,16,17,18], however they do not consider the problem of identifying memory locations of interest. In contrast, we identify these locations during the analyses.

An important problem when dealing with binary code analysis is to figure out the set of interesting data locations used by the program. This is related to pointer analysis (the so-called *aliasing* problem), and has been extensively studied [19,20]. While the majority of pointer analyses have been proposed in the context of compiler optimizations, a certain number of ideas can be borrowed and applied to binary code analysis.

In this paper, our approach is applied to static loop bound estimation, in the context of WCET analysis, so we compare our results with other loop bound estimation tools. The oRange tool [21] is based on an abstract interpretation method defined in [22]. It provides a very fast estimation of loop bounds, but it is restricted to C source code. SWEET [23] features a loop bound estimator, which works on an intermediary representation (ALF format). The approach is based on slicing and abstract interpretation and it generally provides very tight loop bounds even in complex cases, but the running time of the analysis seems to depend on the loop bound values, and in our experience for large loop bounds the analysis did not terminate.

KTA [24] is a static WCET analysis tool based on abstract interpretation and path exploration of binary code. As its purpose is to compute a WCET, it does not directly provide information on loop bounds and we could not find documentation on the method used to compute these bounds. Thus, KAT was not included in our benchmarks. Furthermore, the analysis time seems to depend on the loop bound values.

Compared to these existing works, our approach combines the polyhedral domain with binary code analysis, taking into account memory accesses and supporting analysis of relations between unknown memory addresses; moreover our method is proved to be sound and to always terminate.

7 Experimental results

Our methodology is implemented in a prototype called Polymalys. Our experiments consist of two parts. First, we validate our approach by comparing Polymalys with other existing loop bound analysis tools on classic benchmarks. Then, we provide detailed examples of programs for which Polymalys successfully estimates loops bounds, while the other tools fail to do so.

7.1 Implementation

Polymalys is implemented as a plugin of OTAWA (version 2.0), an open source WCET computation tool [25]. Polymalys relies on OTAWA for control-flow analysis and manipulation, and on PPL [26] for polyhedra operations. Polymalys implements several optimizations to reduce the number of variables and constraints of an abstract state $(p, \mathcal{R}^\sharp, *^\sharp)$, most notably:

– *Unmapped variables*: any variable that is not in $\mathcal{R}^\sharp$ or in $*^\sharp$ can be safely removed from the polyhedron by performing a projection on the remaining (used) variables;
– *Dead registers*: we remove *dead register* variables by perform a preliminary *liveness analysis*, using classic data-flow analysis methods [27];
– *Out-of-scope variables*: whenever modifying the stack pointer register (SP), assuming that the stack grows downwards, for each pair of variables (x_i, x_j) such that $*^\sharp(x_i) = x_j$, if $p \sqsubseteq_\diamond \langle x_i < \mathcal{R}^\sharp[SP] \rangle$ then x_i and x_j can be removed.

7.2 Benchmarks

The analyses have been executed on a PC with an Intel core i5 3470 at 3.2 Ghz, with 8 GB of RAM. Every benchmark has been compiled with ARM crosstool-NG 1.20.0 (gcc version 4.9.1), using the -O1 optimization level.

First, we report the results of our experiments on the Mälardalen benchmarks [28] and on PolyBench [29] in Table 1. The benchmarks *gemver*, *covariance*, *correlation*, *nussinov* and *floyd-warshall* are from PolyBench, while the others are from Mälardalen. We exclude benchmarks that are not supported by OTAWA, mainly due to floating point operations or indirect branching (e.g. `switch`). We compare Polymalys with SWEET [30], PAGAI [10] and oRange [21]. For each benchmark, we report: the number of lines of code (in the C source), the total number of loops, the number of loops that are correctly bounded by each tool, and the computation time. We do not report the computation time for SWEET because we only had access to it through an online applet. For oRange, computation time is below the measurement resolution (10ms), except for *edn*, where it reaches 50ms. We ran PAGAI with the -d pk -t lw+pf options. For the PolyBench benchmarks, we did not succeed in running them with SWEET due to the online applet limitation. For the *correlation* benchmark, we did not succeed in running it with PAGAI, it terminates without giving any result.

The execution time of Polymalys is typically higher than that of PAGAI because we introduce more variables and constraints. We believe that we can reduce the gap with additional optimizations, however Polymalys will probably remain more costly , because it works at a lower level of abstraction.

Cases where tools fail to analyze some loop bounds are depicted in bold. There is only one benchmark for which Polymalys did not find a loop bound: for janne_complex. The difficulty is that it contains complex loop index updates inside a `if-then-else`. On the contrary, there are several cases where Polymalys successfully estimates loops bounds, while the other tools fail to do so. Note that PAGAI does not specifically compute loop bounds, instead it computes loop invariants. We deduced loop bounds from these invariants.

7.3 Loop bounds examples

We further illustrate the differences between tool capabilities on some synthetic program examples.

			Loops Correctly Bounded				Time (ms)	
Benchmark	*LoC*	*Loops*	*Polymalys*	*SWEET*	*PAGAI*	*oRange*	*Polymalys*	*PAGAI*
crc	16	1	1	1	1	1	150	40
fibcall	22	1	1	1	1	1	230	50
janne_complex	26	2	**1**	2	**1**	**1**	870	140
expint	56	3	3	**2**	3	3	732	9140
matmult	84	5	5	5	5	5	3455	1380
fdct	149	2	2	2	2	2	7421	2150
jfdctint	165	3	3	3	3	3	10660	1960
fir	189	2	2	2	2	1	4989	390
edn	198	12	12	12	**9**	12	21356	15660
ns	414	4	4	4	4	4	1700	380
gemver	186	10	10	**N/A**	10	10	12136	6029
covariance	138	11	11	**N/A**	11	11	7248	836
correlation	168	13	13	**N/A**	**N/A**	13	9129	25062
nussinov	143	8	8	**N/A**	8	8	7272	2811
floyd-warshall	112	7	7	**N/A**	**2**	7	2904	468

Table 1. Benchmark results.

Example 11. The following example contains pointer aliasing and pointer arithmetic:

```
#define MAXSIZE 10
foo() {
  int base, end, i;
  if (end - base > MAXSIZE)
    end = base + MAXSIZE;
  for (i = base; i < end; i++);
}
```

PAGAI does not find the loop bound (the loop is considered unbounded), because it does not infer that `ptr = &bound` when executing the instruction `*ptr=15`. Other tools bound the loop correctly (15 iterations).

Example 12. The following example contains an off-by-one array access:

```
#define SIZE 10
foo(int offset) {
  int i, bound = 10;
  int tab[SIZE];
  if ((offset > SIZE) || (offset < 0))
    return -1;
  tab[offset] = 100;
  for (i = 0; i < bound; i++);
}
```

The off-by-one error (lines 5-6) may cause the array cell assignment (line 7) to overwrite the `bound` variable with the value 100. Polymalys correctly detects that

the loop may iterate 100 times, while oRange and SWEET detect a maximum of 10 iterations. PAGAI also bounds to 10 iterations, but warns about a possible undefined behavior and unsafe result. Note that the bound depends on the stack variable allocation layout. In our experiments, the compiler allocates the `bound` variable next to the array. Such an information is much easier to analyze at the binary code level than at the source code level.

Example 13. The following example shows the benefits of a relational domain:

```
#define SIZE 10
foo(int offset) {
  int i, bound = 10;
  int tab[SIZE];
  if ((offset > SIZE) || (offset < 0))
    return -1;
  tab[offset] = 100;
  for (i = 0; i < bound; i++);
}
```

Here, we do not know statically the value of `end` and `base`. However, due to the *if* statement (line 4), Polymalys introduces the constraint $\texttt{end} - \texttt{base} \leq 10$. Thus, Polymalys bounds the loop correctly (10 iterations), while PAGAI, oRange and SWEET do not.

Example 14. Finally, we report analysis results for the motivational example of Figure 1. Polymalys correctly finds that the loop bound is equal to the maximum size of the UDP payload; PAGAI, oRange and SWEET fail to provide any bound.

8 Conclusion

In this paper we propose a novel technique for performing abstract interpretation of binary code using polyhedra. It consists in adding new variables to the polyhedra as the analysis progresses, and maintaining a correspondence with registers and memory addresses. Thanks to the relational properties of polyhedra, our technique naturally provides information on pointer relations when compared to other techniques based on non-relational domains. While the complexity of our method is currently still higher than other existing techniques, we believe that there is room for improvement. In particular, we are planning to extend our work with a modular procedure analysis and a data-structure analysis.

References

1. C. Ballabriga, J. Forget, and G. Lipari, "Symbolic wcet computation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 2, p. 39, 2018.
2. S. Bygde, B. Lisper, and N. Holsti, "Fully bounded polyhedral analysis of integers with wrapping," *Electronic Notes in Theoretical Computer Science*, vol. 288, pp. 3–13, 2012.

3. M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," 1978.
4. P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*. ACM, 1978, pp. 84–96.
5. D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival, "Astrée: Proving the absence of runtime errors," in *Embedded Real Time Software and Systems (ERTS2'10)*, May 2010, p. 9.
6. A. Miné, A. Ouadjaout, and M. Journault, "Design of a Modular Platform for Static Analysis," in *9h Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, ser. Lecture Notes in Computer Science (LNCS), 08 2018, p. 4.
7. R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.
8. S. Gulwani, K. K. Mehra, and T. Chilimbi, "Speed: precise and efficient static estimation of program computational complexity," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, vol. 44, no. 1. ACM, 2009, pp. 127–139.
9. L. Correnson and J. Signoles, "Combining analyses for c program verification," in *Formal Methods for Industrial Critical Systems*, M. Stoelinga and R. Pinger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 108–130.
10. J. Henry, D. Monniaux, and M. Moy, "Pagai: A path sensitive static analyser," *Electronic Notes in Theoretical Computer Science*, vol. 289, pp. 15–25, 2012.
11. M. Karr, "Affine relationships among variables of a program," *Acta Informatica*, vol. 6, no. 2, pp. 133–151, Jun 1976.
12. A. Venet, "The gauge domain: Scalable analysis of linear inequality invariants," in *24th International Conference on Computer Aided Verification (CAV'12)*, 2012, pp. 139–154.
13. A. Miné, "Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics," in *ACM SIGPLAN/ SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, 2006, pp. 54–63.
14. G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Compiler Construction*. Springer, 2004, pp. 2732–2733.
15. A. Djoudi and S. Bardin, "Binsec: Binary code analysis with low-level regions," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, 2015.
16. T. Reps and G. Balakrishnan, "Improved memory-access analysis for x86 executables," in *Compiler Construction*. Springer, 2008, pp. 16–35.
17. A. Sepp, B. Mihaila, and A. Simon, "Precise static analysis of binaries by extracting relational information," in *18th Working Conference on Reverse Engineering (WCRE'11)*. IEEE, 2011.
18. S. Bardin, P. Herrmann, and F. Védrine, "Refinement-based CFG reconstruction from unstructured programs," in *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, 2011.
19. M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering(PASTE'01)*. New York, NY, USA: ACM, 2001, pp. 54–61.
20. B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 290–299, 2007.

21. A. Bonenfant, M. de Michiel, and P. Sainrat, "oRange: A tool for static loop bound analysis," in *Workshop on Resource Analysis, University of Hertfordshire, Hatfield, UK*, 2008.
22. Z. Ammarguellat and W. L. Harrison, III, "Automatic recognition of induction variables and recurrence relations by abstract interpretation," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI '90. New York, NY, USA: ACM, 1990, pp. 283–295.
23. A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper, "Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis," in *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, ser. OpenAccess Series in Informatics (OASIcs), C. Rochange, Ed., vol. 6. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007.
24. D. Broman, "A brief overview of the KTA WCET tool," *arXiv preprint arXiv: 1712.05264*, 2017.
25. C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: An open toolbox for adaptive WCET analysis," in *Software Technologies for Embedded and Ubiquitous Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, vol. 6399, pp. 35–46.
26. R. Bagnara, P. M. Hill, and E. Zaffanella, "The Parma polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems," *Science of Computer Programming*, vol. 72, no. 1, pp. 3–21, 2008.
27. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
28. J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," in *OASIcs-OpenAccess Series in Informatics*, vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.
29. L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," 2012, http://www.cs.ucla.edu/pouchet/software/polybench.
30. B. Lisper, "SWEET–a tool for WCET flow analysis," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2014, pp. 482–485.