Static Analysis Of Binary Code With Memory Indirections Using Polyhedra
Clément Ballabriga, Julien Forget, Laure Gonnord, Giuseppe Lipari, Jordy Ruiz

To cite this version:

HAL Id: hal-01939659
https://hal.archives-ouvertes.fr/hal-01939659
Submitted on 29 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Static Analysis Of Binary Code With Memory Indirections Using Polyhedra*.

Clément Ballabriga, Julien Forget, Laure Gonnord, Giuseppe Lipari, and Jordy Ruiz

1 CRIStAL (Univ. Lille, CNRS, Centrale Lille, UMR 9189), Lille, France
clement.ballabriga@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 [6].

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

* Partially funded by the French National Research Agency, Corteva project (ANR-17-CE25-0003), and CODAS project (ANR-17-CE23-0004-01)
Fig. 1. Network-inspired benchmark

actually runs on the machine, hence no need for additional assumptions on how
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, ad-

dresses and values are well identified. In binary code, the notion of program
variable is lost, so we can only analyze processor registers and memory loca-
tions. We propose to identify the subset of registers and memory locations to
be represented in the abstract state as the analysis progresses. This representa-
tion 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. We remind however that our method-
ology addresses (disassembled) binary code.

The send_request function sends a request in some application-layer proto-
col 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

3 The original bench listing is available here: https://pastebin.com/CSUFYRx3
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 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 \textit{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 \cite{10,28}). We also reduce the set of instructions to a minimum (Polymalys actually supports the ARM A32 instruction set). The syntax of \textit{MEMP} is defined in Figure \ref{fig:MEMP-lang}. 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\textsuperscript{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).

\begin{verbatim}
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 OP\textsuperscript{c}(r_2, r_3) | OP r_1 r_2 r_3
r \leftarrow c | SET r c
Emulate undefined r | RAND r
r_1 \leftarrow *(r_2) | LOAD r_1 r_2
*_(r_1) \leftarrow r_2 | STORE r_1 r_2
Branch to l if r = 0 | BR r l
Halt | END
\end{verbatim}

\textbf{Fig. 2.} Syntax of MEMP

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 \( \xrightarrow{\cdot} \), which operates in a context \((R, \ast)\) consisting of two mappings where:

- \( R : R \rightarrow Z \) is the \textit{registers content}, which maps registers to their values. We assume that it is initially empty;
- \( \ast : Z \rightarrow Z \) is the \textit{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 \( N \) instead of \( Z \) \[10]\.

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 \( m \).

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

3 Abstract domain

The abstract domain we propose is based on the \textit{polyhedral abstract domain} \[12\], to which we add information to track relations between polyhedra variables and registers or memory addresses.
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 \( \text{POLYMAP} \) is defined as \( \mathcal{A} = \mathcal{P} \times (\mathcal{R} \rightarrow \mathcal{V}) \times (\mathcal{V} \rightarrow \mathcal{V}) \). An abstract state \( a \in \mathcal{A} \), with \( a = (p, \mathcal{R}, s) \), consists of a polyhedron
p, a register mapping \( R^♯ \) and an address mapping \( ∗^♯ \). We have \( R^♯(r) = v \) iff variable \( v \) represents the value of register \( r \) in \( p \). We have \( ∗^♯(x_1) = x_2 \) iff variable \( x_2 \) represents the value at the memory address represented by variable \( x_1 \). We denote \( \text{vars}_R(p) \) the codomain of \( R^♯ \) (i.e. register content variables), \( \text{vars}_A(p) \) the domain of \( ∗^♯ \) (i.e. address variables) and \( \text{vars}_C(p) \) the codomain of \( ∗^♯ \) (i.e. address content variables). Sets \( \text{vars}_R(p) \), \( \text{vars}_A(p) \) and \( \text{vars}_C(p) \) are disjoint and are all subsets of \( \text{vars}(p) \).

Example 1. In the following abstract state, register \( r_0 \) contains value 2, and address 2 contains value 1:
\[
\left\{ x_1 = 2, x_2 = x_1, x_3 = 1 \right\}, \left\{ r_0 : x_1 \right\}, \left\{ x_2 : x_3 \right\}
\]
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 \( p \sqsubseteq ▷ \langle x_1 = x_2 \rangle \).

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

To avoid ambiguities with notations on constraints, let \( \text{same}(x_1, x_2) \) 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 (\( R^♯ \) is a function). The absence of aliases can thus be stated as follows.

**Definition 1.** Let \( s = (p, R^♯, ∗^♯) \) be an abstract state. We say that \( s \) is alias free iff:
\[
\forall x_1, x_2 \in \text{vars}_A(p), x_1 \equiv x_2 \Rightarrow \text{same}(x_1, x_2)
\]

### 4 Computing abstract states

Our analysis follows the abstract interpretation framework proposed in [12], adapted to our setting with non-local control-flow, following the technique proposed in Astrée [21] and MOPSA [23]. An important singularity of our analysis is that polyhedral variables are progressively created or removed during the analysis. Whenever a new polyhedral 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 \(\lambda(p, R^\sharp, s^\sharp, (p \cap p', R^\sharp[r_i : x_i], s^\sharp[x_j : x_k]))\), and denote — when a state component remains unchanged. Procedures to compute the join (\(\sqcup\)), widening (\(\delta\)) and antialias of abstract states, and the transfer function \((I)\) 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 \(\textsc{memp}\). 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 [29]). 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(\(\ell, a, L\)) \(\triangleright\) Auxiliary procedure
2: \(a \leftarrow \text{antialias}(a)\)
3: if \(\ell \in L_W\) then \(\triangleright\) Check if \(l\) is a loop header
4: \(new \leftarrow M[\ell] \sqcup (M[\ell] \cup a)\)
5: else
6: \(new \leftarrow M[\ell] \cup a\)
7: end if
8: if \(new \not\sqsubseteq M[\ell]\) then \(\triangleright\) Abstract value for \(\ell\) changed, propagate
9: \(M[\ell] \leftarrow new; L \leftarrow L \cup \ell\)
10: end if
11: end procedure
12: for all \((\ell, I) \in P\) do \(\triangleright\) Start of main procedure
13: \(M[\ell] \leftarrow \bot\) \(\triangleright\) Begin with empty abstract states
14: end for
15: \(M[\ell_1] \leftarrow \top; L \leftarrow \{\ell_1\}\) \(\triangleright\) Program starting label
16: while \(L \neq \emptyset\) do \(\triangleright\) Fixpoint iteration
17: Pick and remove \(\ell\) from \(L\)
18: match \(P[\ell]\)
19: with BR \(x \ell'\)
20: update(\(\ell', ((r = 0), -, -)(M[\ell]), L\)) \(\triangleright\) Branching case
21: update(\(\ell + 1, (-, -, -)(M[\ell]), L\)) \(\triangleright\) Not branching case
22: with END
23: skip
24: with \(
25: update(\(\ell + 1, ((P[\ell])^\sharp)(M[\ell]), L\)) \(\triangleright\) 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, \text{\lor} and \text{\land}). In practice, aliases are introduced when encountering a conditional branching (see Section 4.3). We remove an alias using procedure \textit{antialias}, which relies on the procedure \textit{Merge} defined below. It is based on the following observation: if two addresses are equal, then the values stored at these addresses must be equal too.

Let \( x_1, x_2 \) be two variables of \( \text{vars}_A(p) \) such that: \( \neg \text{same}(x_1, x_2) \land x_1 \equiv x_2 \).

\[
\text{Merge}((p, \mathcal{R}_x^p, \ast^p), x_1, x_2) = (p', \mathcal{R}_x^{p'}, \ast^{p'}) \quad \text{with} \quad p' = (p[x_1/x_2])[\ast^p(x_1)/\ast^p(x_2)] \\
\text{and} \quad \ast^{p'} = \ast^p \setminus \{x_2 : \ast^p(x_2)\}
\]

Function \textit{antialias} : \( A \rightarrow A \) applies \textit{Merge} for each pair of distinct equivalent address variables of an abstract state.

\textit{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 \( \text{Merge}(a, x_1, x_2) \) 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.

\[
a = (\{x_1 = x_2, x_3 \geq 4, x_4 \leq 5\}, \neg, \ast^p = \{x_1 : x_3, x_2 : x_4\}) \\
\text{Merge}(a, x_1, x_2) = ((4 \leq x_3 \leq 5), \neg, \ast^{p'} = \{x_1 : x_3\})
\]
4.3 Transfer functions

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

**Binary operation** If the relation \(r_1 = \text{OP}_c(r_2, r_3)\) 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.

\[
\text{OP} \ r_1 \ r_2 \ r_3 \stackrel{\sharp}{=} \begin{cases} 
  (\langle x = \text{OP}_c(\mathcal{R}^\sharp(r_2), \mathcal{R}^\sharp(r_3)), [r_1 : x], -\rangle(\cdot) & \text{if linear(\text{OP}_c)} \\
  (-, [r_1 : x], -\rangle(\cdot) & \text{otherwise}
\end{cases}
\]

*Example 3.* In Figure 5 at label 6 (i.e. the label immediately following the \texttt{ADD} operation) we introduce the constraint \(x_3 = x_0 + x_1\) and the register mapping \(\mathcal{R}^\sharp_1(r_3) = x_3\).

**Set** The impact of the immediate load instruction is straightforward:

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

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

\[
(\text{RAND}\ r_1)^\sharp = (-, [r_1 : x], -\rangle(\cdot)
\]

**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:

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

*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 \(\mathcal{R}^\sharp_3[r_6] = x_{10}\).

**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 \( \sqcap \) on both cases to manage this uncertainty, and add the constraints for each may-alias address (see May below).

\[
(\text{STORE } r_1 \ r_2)^{\sharp} = \begin{cases} 
\lambda s.\text{Replace}(a)(\text{May}(s)) & \text{if } \exists a \in \text{vars}_A(p), a \equiv r_1 \\
\lambda s.\text{Create}(\text{May}(s)) & \text{otherwise}
\end{cases}
\]

With (\( \circ \) denotes function composition):

\[
\begin{align*}
\text{Replace}(a) &= ((x = R^{\sharp}(r_2)), \neg, [a : x])(\cdot) \\
\text{Create} &= ((x_i = R^{\sharp}(r_1), x_j = R^{\sharp}(r_2)), \neg, [x_i : x_j])(\cdot) \\
\text{May} &= \bigcirc_{a \in A(a \text{ may-alias } r_1)} \lambda s.\text{Replace}(a)(s) \sqcap s
\end{align*}
\]

Example 5. In Figure 5 at label 7, we create a new memory mapping \( s^1_1(x_4) = x_5 \) 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 when branching to a target label \( (\ell') \) 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 \( V_c = \text{vars}(p_1) \cap \text{vars}(p_2) \) and \( p' = \text{proj}(p_1, V_c) \cup_0 \text{proj}(p_2, V_c) \). We denote \( \text{npiv}(p') \) the set of non-pivot variables discovered by Gauss-Jordan elimination performed on the system of equality constraints of \( p' \) (we exclude inequalities). Then, \( \text{npiv}(p') \) is such that, in \( p' \):

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

Let \( \text{linexpr}(x, p_1, \text{npiv}(p')) \) denote the linear expression representation of variable \( x \in \text{vars}(p_1) \) in terms of variables in \( \text{npiv}(p') \), represented as the vector of the linear expression coefficients. Let \( C' \) be the constraint system of \( \text{proj}(p_1, x \cup \text{npiv}(p')) \). If \( C' \) contains an equality constraint involving \( x \), then computing \( \text{linexpr}(x, p_1, \text{npiv}(p')) \) 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 1). Then, we check for the equivalence of address variables according to their linear expression representation, and we perform variable substitutions in \( p_2', R_2^{x_2}, \) and \( s_2' \) in case of equivalence (line 8). Register unification is simpler, we just replace the bindings in \( R_2^{x_2} \) by those of \( R_1^{x_1} \) (line 12).

**Algorithm 2** unif\(y((p_1, R_1^x, s_1^x), (p_2, R_2^x, s_2^x))\)

1: \((p_2', R_2^x', s_2') \leftarrow (p_2, R_2^x, s_2')\) \quad \triangleright \text{common variables}
2: \(V_c \leftarrow \text{vars}(p_1) \cap \text{vars}(p_2)\)
3: \(p' \leftarrow \text{proj}(p_1, V_c) \cup_0 \text{proj}(p_2, V_c)\)
4: \(B \leftarrow \text{npiv}(p')\)
5: \textbf{for all} \((x_i, x_j) \in \text{vars}_A(p_1) \times \text{vars}_A(p_2)\) \textbf{do}
6: \(v_i = \text{linexpr}(x_i, p_1, B); v_j = \text{linexpr}(x_j, p_2, B)\)
7: \textbf{if} \(v_i \neq []\) and \(v_j \neq []\) and \(v_i = v_j\) \textbf{then} \quad \triangleright \text{variables are equivalent}
8: \text{Replace } x_j \text{ by } x_i \text{ and } s_j(x_j) \text{ by } s_i(x_i) \text{ in } p_2', R_2^x', \text{ and } s_2'
9: \textbf{end if}
10: \textbf{end for}
11: \textbf{for all} \(r \in \text{Dom}(R_1^x) \cap \text{Dom}(R_2^x)\) \textbf{do} \quad \triangleright \text{variables are trivially equivalent}
12: \text{Replace } R_2^x(r) \text{ by } R_1^x(r) \text{ in } p_2', R_2^x', \text{ and } s_2'
13: \textbf{end for}
14: \textbf{return} \((p_2', R_2^x', s_2')\)

**Example 8.** In Figure 3 when computing \( \text{unif}(10, 10') \), \( s_1 \) 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 \(\text{linexpr}(x_4) = [1; -1; 0; -4]\) (corresponding, respectively, to the coefficients of \(x_4\), \(x_0\), \(x_7\), and the constant). Since \(\ast_s^2(x_4) = x_9\) (in \(s_1\)) and \(\ast_s^1(x_4) = x_5\) (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 \(f \sqsubseteq g\) when \(\text{Dom}(f) \subseteq \text{Dom}(g)\) and \(\forall x \in \text{Dom}(f) : f(x) = g(x)\). Introducing new mappings in \(\mathcal{R}^f\) or \(\ast^g\) (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 \(a_1 = (p_1, \mathcal{R}_1^f, \ast^f_1)\) and \(a_2 = (p_2, \mathcal{R}_2^f, \ast^f_2)\). The ordering operator \(\sqsubseteq\) is defined as follows:

\[
a_1 \sqsubseteq a_2 \iff p_1^f \sqsubseteq p_2 \land \mathcal{R}_2^f \subseteq \mathcal{R}_1^f \land \ast^f_2 \subseteq \ast^f_1
\]

with \((p_1', \mathcal{R}_1', \ast^f_1') = \text{unify}(a_2, a_1)\)

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 \(\bot = (p_\bot, \mathcal{R}_1^\bot, \ast^f_1)\), where \(p_\bot\) is the empty polyhedron and \(\mathcal{R}_1^\bot, \ast^f_1\) 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** \((p_1, \mathcal{R}_1^f, \ast^f_1) \sqcup (p_2, \mathcal{R}_2^f, \ast^f_2)\)

1: \((p_2', \mathcal{R}_2^f, \ast^f_2') = \text{unify}((p_1, \mathcal{R}_1^f, \ast^f_1), (p_2, \mathcal{R}_2^f, \ast^f_2))\)
2: \(p \leftarrow p_1 \sqcup p_2'\)
3: \(\mathcal{R}^f \leftarrow \emptyset; \ast^f \leftarrow \emptyset\)
4: for all \(r \in \text{Dom}(\mathcal{R}_1^f)\) do
5: if \(\mathcal{R}_2^f(r) = \mathcal{R}_1^f(r)\) then \(\mathcal{R}_2^f(r) \leftarrow \mathcal{R}_1^f(r)\) end if
6: end for
7: for all \(a \in \text{Dom}(\ast^f_1)\) do
8: if \(\ast^f_2(a) = \ast^f_1(a)\) then \(\ast^f_2(a) \leftarrow \ast^f_1(a)\) end if
9: end for
10: return \((p, \mathcal{R}^f, \ast^f)\)
Example 9. In Figure 5, when computing $10 \sqcup 10'$, we obtain identical register and memory mappings for $10$ and $\text{unify}(10, 10')$. The convex hull $p_5 \sqcup p_7$ 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 $\nabla$ is used to ensure that our analysis reaches a fixpoint. $\nabla$ is defined just like $\sqcup$, except that we use a polyhedra widening operator $\nabla_\diamond$ in place of $\sqcup_\diamond$.

### 4.5 Loop bounds

To compute loop bounds, for each loop header label $\ell$ we create a “virtual” register $r_\ell$, to count the number of iterations of $\ell$. We instrument the program so that the register $r_\ell$ is set to 0 when entering loop $\ell$, and incremented at each iteration of $\ell$ (which is fairly classic, see e.g. [15]).

Finally, let $P$ a program of MEMP and $M = \text{interpret}(P)$. Let $\ell_e$ be the label of instruction END in $P$. Let $(p_f, R^f_\ell, *^f_\ell) = M[\ell_e]$. Then the loop bound for a loop header $\ell$ is computed as $\max(p_f, R^\ell_\ell[r_\ell])$ (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 $a_1 = (p_1, R^1_1, *^1_1), a_2 \in A$, $a_3 = (p_3, R^1_3, *^1_3) = a_1 \sqcup a_2$. We have:

$$(p_1 \sqsubseteq o p_3) \land (R^1_3 \sqsubseteq R^1_1) \land (*^1_3 \sqsubseteq *^1_1)$$

Proof. Considering Algorithm 3, $(p_1 \sqsubseteq o p_3)$ follows from line 2, $(R^1_3 \sqsubseteq R^1_1)$ from line 5, and $(*^1_3 \sqsubseteq *^1_1)$ from line 8.

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

$$(R^2_2 \sqsubseteq R^1_2) \land (*^2_2 \sqsubseteq *^1_2) \Rightarrow (R^2_2 \sqsubseteq R^{*'}_1) \land (*^2_2 \sqsubseteq *^{*'}_1)$$

Proof. Obvious from Algorithm 2

**Lemma 1.** Let $a_1, a_2 \in A$. We have: $(a_1 \sqsubseteq a_1 \sqcup a_2) \land (a_2 \sqsubseteq a_1 \sqcup a_2)$. 
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 $\text{vars}(p_1)$, it remains in $a_3$; 2) $v$ is equivalent to a variable $v_1$ of $\text{vars}(p_1)$, it is replaced by $v_1$ in $a_3$ (Algorithm 2 line 8); 3) otherwise, it is removed (Algorithm 3). Then, let $a'_2 = \text{unify}(a_3, a_2)$. 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 $R^2_3 \subseteq R^2_2$, $s^2_3 \subseteq s^2_2$, 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: $(a_1 \sqcup a_2) \subseteq (a_1 \triangledown a_2)$.

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

Property 4. Let $a_1 = (p_1, R^i_1, s^i_1)$, $a_2 \in \mathcal{A}$, $a_3 = (p_3, R^i_3, s^i_3) = a_1 \triangledown_\circ a_2$. We have: $(p_1 \subseteq_\circ p_3) \land (R^i_3 \subseteq R^i_1) \land (s^i_3 \subseteq s^i_1)$

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 $i > N$, $R^i_{i+1} = R^i_i$ and $s^i_{i+1} = s^i_i$. Thus, $a_{i+1} = (p_i \triangledown_\circ q_{i+1}, R^i_i, s^i_i)$, where $q_{i+1}$ is the polyhedron of $b_{i+1}$ and $p_i$ that of $a_i$.

Assuming that $\triangledown_\circ$ is a valid polyhedra widening operator, there exists $m > N$ such that $p_{m+1} = p_m$. Since $m > N$ we also have $R^i_{m+1} = R^i_m$ and $s^i_{m+1} = s^i_m$, 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 $\mathcal{C} = ((\mathbb{R} \rightarrow \mathbb{Z}) \times (\mathbb{Z} \rightarrow \mathbb{Z}))$ 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 $\gamma$ relates data location values to data location variables as follows:
Definition 5. Let \( a = (p, R^\sharp, *^\sharp) \) be an abstract state. The concretization function \( \gamma \) is defined as follows:

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

\[
(p, R^\sharp, *^\sharp) \mapsto \left\{ (*, R) \mid \exists f : \text{Dom}(*^\sharp) \rightarrow \text{Dom}(*), \right. \\
\left. \bigcap_{r \in \text{Dom}(R^\sharp)} (R^\sharp(r) = R(r)) \cap_{x \in \text{Dom}(*^\sharp)} (x = f(x), *^\sharp(x) = *f(x))) \subseteq p \right\}
\]

More intuitively, we build a polyhedron \( p' \) with the following constraints: 1) register values of the concrete state \((R^\sharp(r))\) must be equal to the corresponding variable in the abstract state \((R^\sharp(r))\); 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 \( p' \subseteq p \) then the concrete state satisfies the constraints of \( p \) and belongs to the concretization.

Example 10.

\[
a = \{(1 \leq x_1 \leq 2, x_2 = x_1, x_3 = 1), \{r_0 : x_1\}, \{x_2 : x_3\}\}
\]

\[
\gamma(a) = \left\{(\{r_0 = 1\}, \{*(1) = 1\}), \left\{(r_0 = 2), \{*(2) = 1\}\right) \right\}
\]

\[
(f(x_2) = 1, f(x_2) = 2)
\]

Let \( \rightarrow^* \) denote the transitive closure of \( \rightarrow \). The soundness of our abstract interpretation is established as follows:

Theorem 1. Let \( P \) be a \textsc{Memp} program. Let \( M = \text{Interpret}(P) \). Then, for any concrete state \( s_{\text{init}} \): \( (P \vdash (1, s_{\text{init}}) \rightarrow^* (f, s)) \implies (s \in \gamma(M[f])) \)

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 \( a_1, a_2 \in A \). We have: \( \gamma(a_1) = \gamma(\text{unify}(a_2, a_1)) \).

Proof. Let \( a'_1 = \text{unify}(a_2, a_1) \). 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. \( \square \)

Lemma 4. Let \( a_1, a_2 \in A \). We have: \( (a_1 \subseteq a_2) \implies \gamma(a_1) \subseteq \gamma(a_2) \)

Proof. Let \( s \in \gamma(a_1) \). Let \( a'_1 = (p'_1, R'^\sharp_1, *^\sharp_1) = \text{unify}(a_2, a_1) \). 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 \( p'_1 \subseteq p_2 \land R^\sharp_2 \subseteq R'^\sharp_1 \land *^\sharp_2 \subseteq *^\sharp_1 \) (i.e. \( a_1 \subseteq a_2 \)). 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 \( \text{Dom}(*^\sharp_2) \). So \( s \in \gamma(a_2) \). \( \square \)
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: \( \gamma(a_1) \cup \gamma(a_2) \subseteq \gamma(a_1 \land a_2) \).

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

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

Proof. Let \((p, \mathcal{R}^\dagger, \star^\dagger) = a\). Let \( x_1, x_2 \in \text{vars}_A(p) \) be such that \( \neg \text{same}(x_1, x_2) \land x_1 \equiv x_2 \). Then:

\[
s \in \gamma(a) \Rightarrow s \in \gamma(\langle x_1 = x_2, \star^\dagger(\langle x_1 \rangle) \cup \mathcal{R}^\dagger, \star^\dagger \rangle) \Rightarrow s \in \gamma(\langle p[\langle x_1 \rangle], \mathcal{R}^\dagger, \star^\dagger \rangle) \Rightarrow s \in \gamma(\text{Merge}(a_1, x_1, x_2))
\]

The soundness of \text{antialias} follows.

Lemma 8. Let \( P \) be a \text{MEMP} program. Let \( M = \text{Interpret}(P) \). Then, for all labels \( \ell, \ell' \) of \( P \):

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

Proof. Trivially follows from the formal semantics and from the definition of transfer functions, except for \text{STORE}. Let \( a' = (p', \mathcal{R}'^\dagger, \star') = (\text{STORE} r_1 r_2)^\dagger(a) \).

The proof follows from noting that: 1) Both in the \text{Create} and \text{Replace} cases, we obtain \( \star^\dagger(\mathcal{R}'^\dagger(r_1)) = \mathcal{R}'^\dagger(r_2) \), which is coherent with the formal semantics of \text{STORE}; 2) The soundness of \text{May} follows from the soundness of \( \sqcup \) and \text{Replace}.

Lemma 9. Algorithm 1 terminates.

Proof. Because \( \sqcup \) is applied on loop headers and \( \sqcup \) is a valid widening operator.

6 Related works

Abstract interpretation using polyhedra has been first described in [12]. Static analysis tools such as Astree [21], Frama-C [11] or PAGAI [15] 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 than polyhedra, capable of representing linear constraints between variables, have been proposed, such as for instance [20,30,24]. 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 [4,13,26,27,7], 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 [19,17]. 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 [8] is based on an abstract interpretation method defined in [2]. It provides a very fast estimation of loop bounds, but it is restricted to C source code. SWEET [14] 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 [9] 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 \[5\]. Polymalys relies on OTAWA for control-flow analysis and manipulation, and on PPL \[3\] for polyhedra operations. Polymalys implements several optimizations to reduce the number of variables and constraints of an abstract state \((p, R^\sharp, *^\sharp)\), most notably:

- Unmapped variables: any variable that is not in \(R^\sharp\) 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 \[1\];
- 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 \(*^\sharp(x_i) = x_j, if p \subseteq_0 \langle x_i < R^\sharp[SP] \rangle\) 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 \[16\] and on PolyBench \[25\] 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 \[22\], PAGAI \[18\] and oRange \[8\]. 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.
that PAGAI does not specifically compute loop bounds, instead it computes loop invariants. We deduced loop bounds from these invariants.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LoC</th>
<th>Loops</th>
<th>Polymalys</th>
<th>SWEET</th>
<th>PAGAI</th>
<th>oRange</th>
<th>Polymalys</th>
<th>PAGAI</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>150</td>
</tr>
<tr>
<td>fibcall</td>
<td>22</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>230</td>
</tr>
<tr>
<td>janne_complex</td>
<td>26</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>870</td>
</tr>
<tr>
<td>expint</td>
<td>56</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>732</td>
</tr>
<tr>
<td>matmult</td>
<td>84</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3455</td>
</tr>
<tr>
<td>f dct</td>
<td>149</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>7421</td>
</tr>
<tr>
<td>jfdctint</td>
<td>165</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>10660</td>
</tr>
<tr>
<td>fir</td>
<td>189</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4989</td>
</tr>
<tr>
<td>edn</td>
<td>198</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>9</td>
<td>12</td>
<td>9</td>
<td>12</td>
<td>21356</td>
</tr>
<tr>
<td>ns</td>
<td>414</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1700</td>
</tr>
<tr>
<td>genver</td>
<td>186</td>
<td>10</td>
<td>10</td>
<td>N/A</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>12136</td>
</tr>
<tr>
<td>covariance</td>
<td>138</td>
<td>11</td>
<td>11</td>
<td>N/A</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>7248</td>
</tr>
<tr>
<td>correlation</td>
<td>168</td>
<td>13</td>
<td>13</td>
<td>N/A</td>
<td>N/A</td>
<td>13</td>
<td>N/A</td>
<td>N/A</td>
<td>9129</td>
</tr>
<tr>
<td>nussinov</td>
<td>143</td>
<td>8</td>
<td>8</td>
<td>N/A</td>
<td>8</td>
<td>8</td>
<td>N/A</td>
<td>8</td>
<td>7272</td>
</tr>
<tr>
<td>floyd-warshall</td>
<td>112</td>
<td>7</td>
<td>7</td>
<td>N/A</td>
<td>2</td>
<td>7</td>
<td>N/A</td>
<td>2</td>
<td>2904</td>
</tr>
</tbody>
</table>

Table 1. Benchmark results.

7.3 Loop bounds examples

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

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

```c
foo() {
    int i, bound = 10;
    int *ptr = &bound;
    ptr++; ptr--; *ptr = 15; k = 0;
    for (i = 0; i < bound; 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:

```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++);
}

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 MAXSIZE 10
foo() {
    int base, end, i;
    if (end - base > MAXSIZE)
        end = base + MAXSIZE;
    for (i = base; i < end; 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 end - base ≤ 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


