Range Analysis of Binaries with Minimal Effort

Edd Barrett and Andy King

School of Computing, University of Kent, CT2 7NF, UK

Abstract. COTS components are ubiquitous in military, industrial and governmental systems. However, the benefits of reduced development and maintainance costs are compromised by security concerns. Since source code is unavailable, security audits necessarily occur at the binary level. Push-button formal method techniques, such as model checking and abstract interpretation, can support this process by, among other things, inferring ranges of values for registers. Ranges aid the security engineer in checking for vulnerabilities that relate, for example, to integer wrapping, uninitialised variables and buffer overflows. Yet the lack of structure in binaries limits the effectiveness of classical range analyses based on widening. This paper thus contributes a simple but novel range analysis, formulated in terms of linear programming, which calculates ranges without manual intervention.

1 Introduction

Where once reverse engineering was the preserve of the black-hat, now binaries are routinely inspected members of the intelligence community, military organisations and employees of security firms. For these parties, an area of concern is the security of commercial off-the-shelf software (COTS) such as linkable code libraries [30]. COTS is increasingly deployed since it reduces development times, but such code is written by third-parties, typically with an eye towards functionality rather than security and reliability [11]. COTS could corrupt the system on which it is running or, more insidious still, introduce a trojan horse. The threat posed by COTS is significant motivating security audits which, since the source code is unavailable, are necessarily conducted at the binary level.

Surprisingly, buffer overflow deficiencies are still very popular targets for cyber-criminals [1]. Programs with buffer overflow deficiencies may fall victim to (amongst others) privilege escalation or code injection attacks. Often range information can help in identifying such deficiencies by, for example, asserting that an array index may be out of bounds. Whilst it is recognised that range information can aid the security engineer in the auditing process [12], industrial decompilers do not currently infer ranges for the values stored in basic data-types (though one commercial tool vendor recently mentioned this on its wish list). Range analysis is the pedagogical example that is used to illustrate the need for the widening and narrowing in program comprehension [10]. Even for finite-precision integers, the domain of ranges (also known as intervals) $D = \{\emptyset\} \cup \{[l, u] \mid -2^{31} \leq l \leq u \leq 2^{31} - 1\}$ admits long ascending chains such as

$d_0 = \emptyset$ and $d_{n+1} = [0, n]$ where $n \in [0, 2^{32} - 1]$. The force of this is that fixpoint acceleration aka. widening need be applied to compute an over-approximation of the ranges for registers that arise in loops. The idea behind widening is to accelerate convergence by leap frogging over intermediate points in the chain. To illustrate, observe that the lower bound in the chain d_0, d_1, d_2 is stable after d_1 whilst the upper bound is strictly increasing. Widening would typically enlarge, literally widen, d_3 to $[0, 2^{31} - 1]$ to preserve the lower bound of 0 whilst relaxing the upper bound to the maximum representable signed integer. This side-steps the generation of the intermediates $d_4, \ldots, d_{2^{31}-2}$.

One does not need to relax an unstable bound to the largest, or conversely the smallest, representable number in a single step. Instead, one can prescribe a set of increasing thresholds which are widened to in a series of steps. If relaxing a bound to one threshold is not sufficient for stability then, at the next step, the bound is related the next threshold, and so on. This is called widening with thresholds [6] yet it requires the thresholds to be specified a priori. With a view towards automation, widenings [14, 25] have been suggested that infer the thresholds based on the structure of a program, in particular, where a transition in a chain from one interval to the next flips an inequality from unsatisfiable to satisfiable. The inequalities in question are those that occur in a control structure such as a conditional branch or a loop condition, the intuition being that the larger interval enables a new path through the program to be reached. However, quite apart from reasoning about the satisfiability of systems of inequalities, such widenings rely on extracting inequalities from the program, a problem that is straightforward for a source program, but difficult for its binary counterpart.

Iterative methods based on widening are sound in that they infer ranges which enclose any value that can reside in a register. Such analyses actually compute a post-fixpoint, though the most desirable solution is the least fixpoint which presents the best over-approximation of the intervals. This raises the question of whether least fixpoint can be found directly, dispensing with the need for iteration and widening. Different responses to this question are represented in the works of [15, 22, 26] that compute ranges by, respectively, mixed integer programming, parametric linear programming, and a mixture of transformation and chaotic iteration. The latter approach, in effect, proposes a constraint solver for a class of range constraints that can be solved in polynomial time. The former approaches, attempt to exploit existing mathematical or linear programming packages, though this presents the problem of how to express the fixpoint as an optimisation problem. This is not straightforward and indeed the way branching conditions are encoded in [22] is unsound for some classes of loop. (In the spirit of the call for papers, this shortcoming is discussed in Sect. 5 as well as its relationship to follow-on work [29]). Whether by design or by accident, the mathematical programming formulation, which is subsequently linearised, seems to avoid this problem but the encoding is not straightforward and it is not easy to validate the method due to its conceptual complexity. In this paper we extend existing techniques by, paradoxically, stripping them down. In doing so, we make the following contributions:

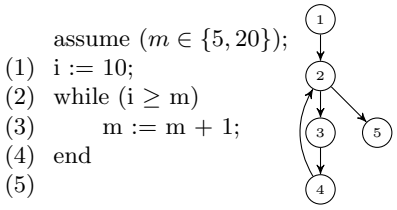


Fig. 1: (a) Program code (b) CFG and (c) collecting semantics

- We show how range analysis can be formulated, in the words of the title "with minimal effort", using systems of min and max constraints;
- We show how systems of such constraints can be solved by repeatedly calling a linear programming package;
- We show how the number of calls to the package can be significantly reduced by solving the linear programs in a propitious order.

The structure of the remainder of this paper is as follows. Sect. 2 shows a worked example of our analysis over a program, then in Sect. 3 we detail how we solve systems of inequalities containing disjunctions using repeated linear programming (LP). Experimental results of our analysis applied to several small programs are presented in Sect. 4 and in Sect. 5 we discuss some shortcomings in existing work that influenced the design of our analysis. Related work is discussed in Sect. 6 before we conclude the main body of the paper in Sect. 7.

2 Worked Example

In this section we explain how ranges can be derived without resorting to widening. Although our work is targeted at the binary level, we will introduce the ideas in terms of a generic high-level program so as to aid comprehension.

2.1 Collecting Semantics

Fig. 1a shows a small program with the program points annotated (1) through to (5). Our problem is how to summarise the program state at each of these points without actually running the program. We start by considering a natural set representation of all possible values of i and m at each program point. We aim to compute the smallest hyper-rectangle (a tuple of intervals) summarising all possible i and m combinations for each program point. To this end, the state at a single program point is expressed as a 2-dimensional vector $\langle i, m\rangle$, thus the states that can possibly arise at these 5 program points is described by 5 sets of vectors, namely $S_k \subseteq [-2^{31}, 2^{31} - 1]^2$. Each set S_k is finite, though possibly large, since we suppose that i and m are represented by 32-bit signed integers.

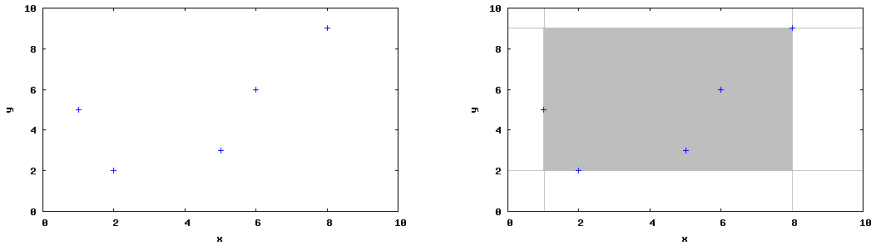


Fig. 2: (a) $S = \{\langle 2,2\rangle, \langle 5,3\rangle, \langle 1,5\rangle, \langle 6,6\rangle, \langle 8,9\rangle\}$ (b) $\alpha(S) = [1,8] \times [2,9]$

Fig. 1c presents a system of recursive equations that define and relate the sets $S_1, \ldots, S_5$. The dependencies between the program points, hence the sets, are illustrated in Fig. 1b. Note S_2 is defined in terms of S_4 and S_2^* which, in turn, is defined in terms of S_1. This is because control passes from (1) and (4) to program point (2). The set S_2^* is merely introduced as a calculational device (an intermediate set) that is used to decompose S_2 into an update operation and a merge operation, that define S_2^* and S_2 respectively. Note too that the increment operation at line (3) can potentially overflow, though it does not in this example. Instead of separately modelling the two modes of the increment: the exact mode when the increment does not wrap, and the overflow mode, and then distinguishing between these two modes with a guard, we simplify the presentation by modelling the overflow with a min operation. Together these equations can be considered as defining a collecting semantics [9] for the program; a semantics over sets that provides a basis for abstraction.

2.2 Abstract Semantics

Every set $S \subseteq [-2^{31}, 2^{31}-1]$ can be described by an interval drawn from the abstract domain $D = \{\emptyset\} \cup \{[l,u] \mid -2^{31} \leq l \leq u \leq 2^{31}-1\}$. Moreover, an n-ary tuple of intervals can describe a set of n-ary vectors, an idea that is formalised with an abstraction α mapping and a concretisation γ mapping [9]. The latter map explains how to interpret an n-tuple of intervals and the former specifies how best to describe a set $S \subseteq [-2^{31}, 2^{31}-1]^n$. These maps are defined thus:

$$\begin{array}{ll} \gamma : D^n \to \wp([-2^{31}, 2^{31}-1]^n) & \alpha : \wp([-2^{31}, 2^{31}-1]^n) \to D^n \\ \quad \gamma(\emptyset) = \emptyset & \quad \alpha(\emptyset) = \emptyset \\ \quad \gamma(S') = S' & \quad \alpha(S) = \cap\{S' \in D^n \mid S \subseteq S'\} \end{array}$$

Note how an n-tuple of intervals $\langle d_1, \ldots, d_n\rangle \in D^n$ is interpreted as its cartesian product $d_1 \times \ldots \times d_n$ which defines an hyper-rectangle in n-dimensional space. Thus the subset ordering on D naturally lifts to D^n by $\langle d_1 \ldots d_n\rangle \subseteq \langle e_1 \ldots e_n\rangle$ iff $d_i \subseteq e_i$ for all $i \in \{1, \ldots, n\}$. Observe too how $\alpha(S)$ is defined as the least hyper-rectangle that encloses S. Fig. 2 illustrates $\alpha(S)$ for a set S that is planar.

Abstraction and concretisation relate sets of vectors to hyper-rectangles. With this relationship in place, we can relax the collecting semantics given previously, to a system of recursive equations that operate over hyper-rectangles

rather then arbitrary sets. Each S'_k is designed to faithfully characterise S_k in that $S_k \subseteq \gamma(S'_k)$. This relationship can be shown to hold inductively when:

$$\begin{aligned}
S'_1 &= \{\langle i, m\rangle \mid -2^{31} \leq i \leq 2^{31}-1 \wedge 5 \leq m \leq 20\} \\
S'^{*}_2 &= \{\langle 10, m\rangle \mid \langle i, m\rangle \in S'_1\} \\
S'_2 &= \alpha(\ S'^{*}_2 \cup S'_4\ ) \\
S'_3 &= \alpha(\ \{\langle i, m\rangle \mid \langle i, m\rangle \in S'_2 \wedge i \geq m\}\ ) \\
S'_4 &= \alpha(\ \{\langle i, \min(m+1, 2^{31}-1)\rangle \mid \langle i, m\rangle \in S'_3\}\ ) \\
S'_5 &= \alpha(\ \{\langle i, m\rangle \mid \langle i, m\rangle \in S'_2 \wedge i < m\}\ )
\end{aligned}$$

When incrementing m (S'_4) the resulting upper bound of may saturate. It is possible to obtain a more faithful model of integer overflow using integer linear programming, but in the interest of brevity we refrain from presenting this idea here. Since the above semantics is derived as an abstraction of the collecting semantics, henceforth it will be referred to as the abstract semantics.

2.3 Direct Calculation of the Abstract Semantics

The abstract semantics can be evaluated by iteratively applying the above equations, with widening, until stability is achieved. This does not necessarily give the least (best) solution due to the approximation introduced by widening. However, the hyper-rectangles can be found directly by solving systems of equations. Let $S'_1 = [l_{i,1}, u_{i,1}] \times [l_{m,1}, u_{m,1}], \ldots, S'_5 = [l_{i,5}, u_{i,5}] \times [l_{m,5}, u_{m,5}]$. The solution to the following reformulation (as an optimisation problem) is the least fixed point of the abstract semantics:

$$\text{Minimise} : \sum_{j=1}^{5}(u_{i,j} - l_{i,j}) + \sum_{j=1}^{5}(u_{m,j} - l_{m,j})$$

subject to the (non-linear) constraints:

$$\begin{array}{rcrcr}
l_{i,1} = -2^{31} & \wedge & u_{i,1} = 2^{31}-1 & \wedge \\
l_{i,2^*} = 10 & \wedge & u_{i,2^*} = 10 & \wedge \\
l_{i,2} = \min(l_{i,2^*}, l_{i,4}) & \wedge & u_{i,2} = \max(u_{i,2^*}, u_{i,4}) & \wedge \\
l_{i,3} = \max(l_{i,2}, l_{m,2}) & \wedge & u_{i,3} = u_{i,2} & \wedge \\
l_{i,4} = l_{i,3} & \wedge & u_{i,4} = u_{i,3} & \wedge \\
l_{i,5} = l_{i,2} & \wedge & u_{i,5} = \min(u_{i,2}, u_{m,2}-1) & \wedge \\
l_{m,1} = 5 & \wedge & u_{m,1} = 20 & \wedge \\
l_{m,2^*} = l_{m,1} & \wedge & u_{m,2^*} = u_{m,1} & \wedge \\
l_{m,2} = \min(l_{m,2^*}, l_{m,4}) & \wedge & u_{m,2} = \max(u_{m,2^*}, u_{m,4}) & \wedge \\
l_{m,3} = l_{m,2} & \wedge & u_{m,3} = \min(u_{i,2}, u_{m,2}) & \wedge \\
l_{m,4} = \min(l_{m,3}+1, 2^{31}-1) & \wedge & u_{m,4} = \min(u_{m,3}+1, 2^{31}-1) & \wedge \\
l_{m,5} = \max(l_{m,2}, l_{i,2}+1) & \wedge & u_{m,5} = u_{m,2} &
\end{array}$$

The cost function asserts that the desired solution is the least (best) hyper-rectangle that satisfies all of the constraints. Of particular note are the constraints $l_{i,2} = \min(l_{i,2^*}, l_{i,4})$ and $u_{i,2} = \max(u_{i,2^*}, u_{i,4})$ which assert that $[l_{i,2}, u_{i,2}]$

is the smallest interval that encloses both $[l_{i,2^*}, u_{i,2^*}]$ and $[l_{i,4}, u_{i,4}]$. Likewise $l_{m,2} = \min(l_{m,2^*}, l_{m,4})$ and $u_{m,2} = \max(u_{m,2^*}, u_{m,4})$ assert tight bounds on m. In combination, these four constraints symbolically define S'_2 as the merge of the hyper-rectangles S'_1 and S'^*_2. Modelling the loop condition $i \geq m$ is a particular subtlety. Note how $l_{i,3} = \max(l_{i,2}, l_{m,2})$ and $u_{i,3} = u_{i,3}$ strengthen (not weaken) the lower bound of i but preserve its upper bound. Conversely $l_{m,3} = l_{m,2}$ and $u_{m,3} = \min(u_{i,2}, u_{m,2})$ refine the upper bound of m but preserve its lower bound. An analogous construction is used to model the loop exit condition.

Solving the above (with the technique outlined in the following section) we find the following ranges:

$$\begin{array}{ll} S'_1 = [-2^{31}, 2^{31} - 1] \times [5, 20] & S'_4 = [10, 10] \times [6, 11] \\ S'_2 = [10, 10] \times [5, 20] & S'_5 = [10, 10] \times [11, 20] \\ S'_3 = [10, 10] \times [5, 10] & \end{array}$$

3 Solving Minimum and Maximum Constraints

The min and max terms in our system of inequalities are non-convex, yet convexity is a prerequisite of classical linear programming. We overcome this through repeated linear programming, which we overlay with heuristics.

3.1 Constraint Decomposition

First we decompose our system of constraints into a set of linear constraints L and a vector of non-convex complementary constraints $\boldsymbol{C}$. Note that the constraints in $\boldsymbol{C}$ must be disjunctions of linear terms and not arbitrary non-convex terms. Constraints containing min or max terms are rewritten using the following equivalence:

$$\begin{aligned} x = \min(y, z) &\equiv (x \leq y) \wedge (x \leq z) \wedge (x = y \vee x = z) \\ x = \max(y, z) &\equiv (x \geq y) \wedge (x \geq z) \wedge (x = y \vee x = z) \end{aligned}$$

For example, the constraint $u_{m,3} = \min(u_{i,2}, u_{m,2})$ is decomposed into the linear system $L = \{u_{m,3} \leq u_{i,2}, u_{m,3} \leq u_{m,2}\}$ which is complemented with the system $\boldsymbol{C} = \langle (u_{m,3} = u_{i,2} \vee u_{m,3} = u_{m,2}) \rangle$. The decomposed constraints for the worked example are show in figure 3.

3.2 Constraint Solving

Although the disjuncts of $\boldsymbol{C}$ preclude LP from being directly applied, the complementary constraints can be supported by repeatedly solving LPs. To see this, observe that the complementary constraint $l_{m,6} = l_{m,5} \vee l_{m,6} = l_{i,5} + 1$ has one of two states, according to whether the first or the second equality holds. The disjuncts of $\boldsymbol{C}$ thus prescribe a search space of $2^{|\boldsymbol{C}|}$ combinations. In principle each of these combinations could be enumerated and combined with the linear component L to form an LP. Each LP could then be independently solved and

$$
\begin{aligned}
L = \{\ & l_{i,1} = -2^{32}, && u_{i,1} = 2^{31}-1, && && \\
& l_{i,2*} = 10, && u_{i,2*} = 10, && && \\
& l_{i,2} \le l_{i,2*}, && l_{i,2} \le l_{i,4}, && u_{i,2} \ge u_{i,2*}, && u_{i,2} \ge u_{i,4} \\
& l_{i,3} \ge l_{i,2}, && l_{i,3} \ge l_{m,2}, && u_{i,3} = u_{i,2}, && \\
& l_{i,4} = l_{i,3}, && u_{i,4} = u_{i,3} && && \\
& l_{i,5} = l_{i,2}, && u_{i,5} \le u_{i,2}, && u_{i,5} \le u_{m,2}-1, && \\
& l_{m,1} = 5, && u_{m,1} = 20, && && \\
& l_{m,2*} = l_{m,1}, && u_{m,2*} = u_{m,1}, && && \\
& l_{m,2} \le l_{m,2*}, && l_{m,2} \le l_{m,4}, && u_{m,2} \ge u_{m,2*}, && u_{m,2} \ge u_{m,4}, \\
& l_{m,3} = l_{m,2}, && u_{m,3} \le u_{i,2}, && u_{m,3} \le u_{m,2} && \\
& l_{m,4} \le l_{m,3}+1, && l_{m,4} \le 2^{31}-1 && , u_{m,4} \le u_{m,3}+1 && , u_{m,4} \le 2^{31}-1 \\
& l_{m,5} \ge l_{m,2}, && l_{m,5} \ge l_{i,2}+1, && u_{m,5} = u_{m,2} && \}
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{C} = \langle\ & (l_{i,2} = l_{i,2*} \vee l_{i,2} = l_{i,4}), && (u_{i,2} = u_{i,2*} \vee u_{i,2} = u_{i,4}) \\
& (l_{i,3} = l_{i,2} \vee l_{i,3} = l_{m,2}), && (u_{i,5} = u_{i,2} \vee u_{i,5} = u_{m,2}-1) \\
& (l_{m,2} = l_{m,2*} \vee l_{m,2} = l_{m,4}), && (u_{m,2} = u_{m,2*} \vee u_{m,2} = u_{m,4}) \\
& (u_{m,3} = u_{i,2} \vee u_{m,3} = u_{m,2}), && (l_{m,4} = l_{m,3}+1 \vee l_{m,4} = 2^{31}-1) \\
& (u_{m,4} = u_{m,3}+1 \vee u_{m,4} = 2^{31}-1), && (l_{m,5} = l_{m,2} \vee l_{m,5} = l_{i,2}+1 \quad \rangle
\end{aligned}
$$

Fig. 3: Worked example constraints decomposed.

then compared to find the least value of the objective function overall. However, we suggest an alternative strategy.

The boilerplate of our algorithm is shown in Algorithm 1. Before the algorithm commences, we augment L with:

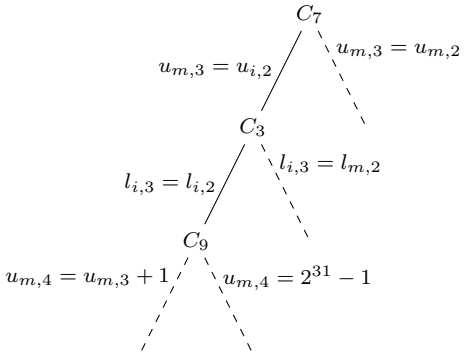
$$\bigwedge_{1 \le k \le 5} (-2^{31} \le l_{i,k} \wedge u_{i,k} \le 2^{31}-1) \wedge (-2^{31} \le l_{m,k} \wedge u_{m,k} \le 2^{31}-1)$$

so as to ensure that all the LPs are bounded. This augmented system will henceforth be denoted $\bar{L}$. The search starts at the root node of the search space with $\tau = \mathit{true}$. At each stage in the search $\bar{L} \wedge \tau$ is tested for satisfiability with a solver, where τ is the conjunction of equalities selected thus far from $\boldsymbol{C}$ (as illustrated in Fig. 4). If $\bar{L} \wedge \tau$ is unsatisfiable, then there is no solution for this choice of τ. Furthermore, augmenting τ with additional equalities from $\boldsymbol{C}$ would further constrain the LP rather than relax it. However, if $\bar{L} \wedge \tau$ is satisfiable, then another equality is selected from $\boldsymbol{C}$ from a disjunct that has not already been considered. This is the role of ChooseNextDecision. If exactly one equality has been selected from each disjunct of $\boldsymbol{C}$ and $\bar{L} \wedge \tau$ is still feasible, then a solution is recorded. The search terminates when the search space is exhausted, at which point the solution with the least objective function value is reported as the overall minimum.

The benefit of this strategy is that if inconsistency is detected when τ contains relatively few equalities from $\boldsymbol{C}$ then many branches through the search space can be discarded simultaneously. The effectiveness of this pruning strategy is dependent upon the ordering of decisions, and in particular the equalities that

Algorithm 1 Binary search algorithm.

```
1: function BinSearch(L̄, F, C, τ)
2:     r ← MinimizeLp(F, L̄ ∧ τ)
3:     if ¬ Sat(r) then
4:         return []            // No solutions here, prune.
5:     else if AllDecisionsMade(C, τ) then
6:         return [(r, τ)]      // Found a leaf with a solution
7:     end if
8:     (e1 ∨ e2) ←ChooseNextDecision(C, r, τ)
9:     s_l ← BinSearch(L̄, F, C, τ ∧ e1)
10:    s_r ← BinSearch(L̄, F, C, τ ∧ e2)
11:    return Append(s_l, s_r)
12: end function
```



$\bar{L}_x$	Relaxation ($\bar{L} \wedge \tau$)
1	$\bar{L}$
2	$\bar{L} \wedge (u_{m,3} = u_{i,2})$
3	$\bar{L} \wedge (u_{m,3} = u_{i,2}) \wedge (l_{i,3} = l_{i,2})$
⋮	⋮

Fig. 4: First three linear relaxations of the worked example program.

are selected from $\boldsymbol{C}$. For the search to be effective, inconsistencies need to be found early in the search, at a shallow depth in the tree, in order to maximise the effect of pruning. If an inconsistency is found later, then it is likely to be duplicated down alternative paths, nullifying the effect of pruning. Like many combinatoric search problems, the worst case complexity is high (worst case number of linear programs is $2^{|\boldsymbol{C}|+1} - 1$), but in practice performance can be significantly improved with the use of heuristics.

3.3 Heuristics

In order to improve upon the worst case complexity of our search space, we implement the following heuristics:

H1: Prune Inconsistencies Early. This heuristic suggests which disjunct $C_n \in \boldsymbol{C}$ is a good candidate from which an equality should be selected. Suppose solving $\bar{L} \wedge \tau$ returns a solution for which $(u_{m,3} = -2^{31}) \wedge (u_{i,2} = 10) \wedge (u_{m,2} = 20)$. Observe that the disjunct $C_7 = (u_{m,3} = u_{i,2} \vee u_{m,3} = u_{m,2})$ is unsatisfiable under

Algorithm 2 Heuristic 1

```
1: function ChooseNextDecision(C, r, τ)
2:     v ← GetViolatedCCs(C, τ)
3:     if |v| > 0 then
4:         let n ∈ v
5:     else
6:         n ← ChooseArbitraryNextDecision(C, r)
7:     end if
8:     return n
9: end function
```

this assignment. Then the heuristic suggests that τ should next be extended with an equality from C_7. If all complementary constraints are satisfied, then an arbitrary $C_n \in \boldsymbol{C}$ is chosen for selecting an equality; the selected C_n is literally chosen at random, thus introducing non-determinism into the algorithm. The intuition behind this selection strategy is that if C_7 is unsatisfiable for one solution to the LP, then extending τ with one of its equalities is likely to detect an inconsistency thereby pruning the search space. Algorithm 2 provides an implementation of ChooseNextDecision using this heuristic.

H2: Block Weak and Duplicate Solutions. A solution is found once an equality is selected from each disjunct of $\boldsymbol{C}$ such that $\bar{L} \wedge \tau$ remains satisfiable. There is only one minimal solution, but it is possible for other solutions to exist which, whilst they satisfy the min/max constraints, yield less tight intervals. It is also possible for both sides of the disjunct of a complimentary constraint to evaluate true (eg. $(l_{i,3} = l_{i,2} \vee l_{i,3} = l_{m,2})$ where $l_{i,2} = l_{i,3} = l_{m,2} = 1$), thereby introducing ineffectual decisions in C and ultimately duplicate solutions. Because there is no value in finding a solution if it does not improve the objective, we propose adding an extra linear constraint to the system that ensures that any solution that is subsequently found improves on the least value of the objective. Suppose that we analyse the worked example program and a solution is found whose objective function value we call o_{min}. Subsequent linear programs are solved in conjunction with an additional blocking constraint: $\sum_{j=1}^{5}(u_{i,j} - l_{i,j}) + \sum_{j=1}^{5}(u_{m,j} - l_{m,j}) < o_{min}$. Through this construction only solutions yielding a strictly smaller objective are feasible, thus further pruning the search space and in turn the number of LPs the analysis must perform.

4 Experimental Results

Our tooling, given a control flow graph and a description of CFG edge operations, generates $\langle \bar{L}, \boldsymbol{C} \rangle$ and proceeds to perform the binary search as described in Sect. 3. The binary search uses the `lpsolve` solver which we interface using Python language bindings. Individual search heuristics (H1 and H2) may be switched on and off, allowing performance comparisons to be drawn under different heuristics configurations. Evaluation of the complimentary constraints

Interval	$m_1 \in$		
	$[2,2]$	$[63,71]$	$[5,20]$
m_{2*}	$[2,2]$	$[63,71]$	$[5,20]$
m_2	$[2,11]$	$[63,71]$	$[5,20]$
m_3	$[2,10]$	$[63,10]$	$[5,10]$
m_4	$[3,11]$	$[64,11]$	$[6,11]$
m_5	$[11,11]$	$[63,71]$	$[11,20]$
i_1	$[0,255]$	$[0,255]$	$[0,255]$
i_{2*}	$[10,10]$	$[10,10]$	$[10,10]$
i_2	$[10,10]$	$[10,10]$	$[10,10]$
i_3	$[10,10]$	$[63,10]$	$[10,10]$
i_4	$[10,10]$	$[63,10]$	$[10,10]$
i_5	$[10,10]$	$[10,10]$	$[10,10]$

(a) Intervals determined by the analysis.

$m_1 \in$	H1	H2	Mean #LPs	Mean Time (s)
$[2,2]$	✗	✗	208	0.2
	✓	✗	183	0.9
	✗	✓	152	0.1
	✓	✓	38	0.2
$[63,71]$	✗	✗	200	0.2
	✓	✗	125	0.6
	✗	✓	105	0.1
	✓	✓	45	0.2
$[5,20]$	✗	✗	207	0.2
	✓	✗	211	1.0
	✗	✓	143	0.1
	✓	✓	44	0.2

(b) Mean number of LPs and time required to find the best solution (sample size of 10, H1/H2 show heuristics enabled).

Fig. 5: Experimental results for the worked example program shown in Fig. 1.

(for heuristic 1) is performed by SymPy, a computer algebra library for Python. Experiments were run on a 3GHz 64-bit Intel machine running OpenBSD.

The tables in Fig. 5 show some experimental results for the worked example (Fig. 1) with varying initial values of m_1 and heuristics configurations. Because the algorithm is non-deterministic, each experiment configuration was run 10 times and averages were taken. We show the intervals inferred by our analysis, the mean number of linear programs required (out of a possible worst case number of $2^{10+1} - 1 = 2047$) to find the best solution and the average amount of time spent finding the solution (in seconds). The intervals of the best solution are precise and in all cases, our heuristics reduced the number of LPs required to find the best solution. Further, when $m_1 \in [63, 71]$, the loop body is not entered and this is reflected in our results by the empty intervals at program points 3 and 4. Interestingly run-times appear to be longer when heuristic 1 is enabled. Profiling revealed that the evaluation of complimentary constraints (GetViolatedCCs) accounts for a large portion of solving time for this small example.

A second program was analysed by our analysis, this time at the binary level. Fig. 6 shows the disassembly of a defective implementation of `memcpy(3)` for the x64 architecture. The function takes a pointer to a buffer to write to (`rdi`), a buffer to read from (`rsi`) and a length argument (`rdx`). The `r15` register is used as both a loop counter and as an index into the source and destination buffers. Let $r15_1 \in [l_{r15,1}, u_{r15,1}]$ be the interval representing `r15` at the program point marked `p1`, where bytes are written into the destination buffer. In order to apply our conditional semantics to binary programs, high-level predicates are extracted from pairs of assembler instructions which define and use boolean flags within the status register [5]. For example, `cmp r15, rdx; jg return` causes a control flow despatch if `r15` $>$ `rdx`.

```
memcpy: xor r15, r15                    # loop counter
loop:   cmp r15, rdx
        jg return
        mov byte ptr cl, [rsi+r15]      # read out src
p1:     mov byte ptr [rdi+r15], cl      # write in dest
        inc r15
        jmp loop
return: mov rax, rdi                    # return ptr to dest
        ret
```

Fig. 6: A function to copy buffers.

`rdx` ∈	$r15_1$	H1	H2	MLP	MT
$[8, 8]$	$[0, 8]$	✗	✗	25143	75
		✓	✗	18596	178
		✗	✓	11940	45
		✓	✓	69	1
$[8, 4096]$	$[0, 4096]$	✗	✗	31045	116
		✓	✗	18963	198
		✗	✓	8989	45
		✓	✓	62	1
$[31, 66]$	$[0, 66]$	✗	✗	28639	107
		✓	✗	18963	194
		✗	✓	13885	55
		✓	✓	68	1

(a) `memcpy(3)`

`rdi` ∈	rax_2	H1	H2	MLP	MT
$[8, 8]$	$[1, 8]$	✗	✗	36621	219
		✓	✗	20342	302
		✗	✓	7891	34
		✓	✓	85	1
$[7, 13]$	$[1, 13]$	✗	✗	35856	151
		✓	✗	19977	258
		✗	✓	8701	37
		✓	✓	99	1
$[4, 128]$	$[1, 128]$	✗	✗	40352	166
		✗	✓	19696	252
		✓	✗	7948	34
		✓	✓	105	1

(b) Endian swap

Fig. 7: Results for the second and third experiments (MLP is the mean number of linear programs required and MT is the mean time in seconds). Means calculated from a sample of 10 runs.

Fig. 7a shows the results of our analysis upon the `memcpy(3)` implementation. If a buffer size of between 8 and 4096 is passed to this function, then our analysis indeed infers $r15_1 \in [0, 4096]$, thereby indicating that one byte is potentially written outside of the allocated buffer. Again, the number of LPs the analysis is required to solve is improved through the use of heuristics. Evaluation of complimentary constraints is especially expensive when heuristic 1 alone is enabled, but the overall time spent searching is vastly improved through the use of heuristics 1 and 2 combined. This experiment utilises 18 complimentary constraints, so the theoretical worst case number of LPs required is $2^{18+1} - 1 = 524287$.

Fig. 8 shows an algorithm to byte-swap 16-bit words in a memory buffer. The function takes a buffer length (`rdi`) and a pointer to a buffer to swap (`rsi`). The register `rax` is being used as an index into the buffer pointed to by `rsi`. Let $rax_1 \in [l_{rax,1}, u_{rax,1}]$ and $rax_2 \in [l_{rax,2}, u_{rax,2}]$ be the intervals of `rax` at marked points `p1` and `p2` respectively. The results of the analysis of this program (Fig. 7b) highlight an interesting deficiency in our analysis. If the function is called with an odd buffer size argument, then the function indeed writes one byte outside its allocated buffer. Yet if we pass our analysis a a buffer size argument of 8, then

```
endswap: xor r15, r15   # loop counter
loop:    cmp r15, rdi
         jge return
         mov rax, r15   # rax is used as a write index
         mov byte ptr dl, [rsi+r15]
         inc r15
         mov byte ptr cl, [rsi+r15]
         inc r15
p1:      mov byte ptr [rsi+rax], cl
         inc rax
p2:      mov byte ptr [rsi+rax], dl
         jmp loop
return:  ret
```

Fig. 8: A 16-bit byte swap.

we infer $rax_2 \in [1, 8]$. This would suggest that a byte was written outside of the allocated buffer, however, in reality this is untrue. Our analysis is unable to take into account the strided nature of the loop count and thus over-approximates the upper bound of `rax` upon entry to the loop. Nevertheless, the solution safely over-approximates all possible register values. The solution is found quickly and in a fraction of the worst case number of linear programs ($2^{22+1} - 1 = 8388607$).

5 Discussion

The analysis presented in this paper was mostly inspired by the pioneering work by Rugina et al. [22]. Our extension to Rugina's work diverges in some aspects with response to some shortcomings that are not mentioned in the literature. In this section we will discuss these aforementioned shortcomings thus providing an insight into some of the design decisions of our analysis.

5.1 Conditional Semantics

It would appear that Rugina's branching semantics are unable to model a class of loop constructs correctly. One such example is the program:

`assume(m < 10);` B_1`: int i = 10;` B_2`: while (i >= m){`B_3`: m = m + 1;}`

Following Rugina's constraint generation scheme we reduce this program to the following constraints, which are infeasible:

$$\begin{array}{llllllll} l_{i,2} \leq 10 & \wedge & 10 \leq u_{i,2} & \wedge & l_{m,2} \leq l_{m,1} & \wedge & u_{m,1} \leq u_{m,2} & \wedge \\ l_{i,3} \leq l_{m,2} & \wedge & u_{i,2} \leq u_{i,3} & \wedge & l_{m,3} \leq l_{m,2} & \wedge & \mathbf{u_{m,2} \leq u_{m,3}} & \wedge \\ l_{i,2} \leq l_{i,3} & \wedge & u_{i,3} \leq u_{i,2} & \wedge & l_{m,2} \leq l_{m,3} + 1 & \wedge & \mathbf{u_{m,3} + 1 \leq u_{m,2}} & \end{array}$$

5.2 Junk propagation

In both our and Rugina's analysis unreachable code causes the existence of empty intervals (ie. an interval, $[l, u]$, where $l > u$). Consider the following program snippet, in which B_3 is unreachable:

B_1: `int i = 12;` B_2: `while (i <= 10)` {B_3: `i = i + 1`}

If we analyse this program via Rugina's method, we infer the following intervals: $i_2 \in [12, 15], i_3 \in [12, 10]$. The interval at i_3 is empty, correctly indicating that this program point is unreachable. Unfortunately, the loop propagates bounding information from B_3 back to B_2, thereby compromising the precision of the upper bound of i_2. We call this phenomenon "junk propagation".

We overcome imprecision incurred through junk propagation by treating the false branch of the loop check (or any conditional for that matter) as a conditional whose predicate is a negation of the predicate of the true branch. For the counter-example we have just presented, we insert a loop exit block B_4 which is a conditional edge asserting that $i > 11$, thereby retaining the precision of the upper bound of i_2.

6 Related Work

Range analysis has a long history in compilation and verification, dating back to the seminal work of Harrison [16]. This work resonates with ideas in the widening and narrowing approach to abstract interpretation [9], for instance, "this bound may be fed back into the range analysis to revise the ranges" is reminiscent of narrowing which classically following widening so as to tighten ranges. In this work, "each range description describes an arithmetic sequence with a lower bound, upper bound and an increment", and thus the descriptions are actually strided intervals [20], abstractions that are considered to be a recent invention. Widening and narrowing is a research topic within its own right [6, 14, 25]; a topic that is not confined to abstract interpretation either. Indeed, widening has been applied in conjunction with SMT solving [18], to pose successively weaker candidate loop invariants to the solver until an invariant is found that holds across every iteration of the loop. Our paper, together with [15, 26], offers an alternative way of handling loops that aspire to directly compute a fixpoint.

As already discussed, range analysis can be expressed as mathematical integer programming [15], which, in turn, can be reduced to integer linear programming. In this approach binary decision variables are used to indicate the reachability of, among other things, guarded statements. This idea could be developed by making use of the finite nature of machine arithmetic, and encoding a branch condition $x \leq y$ as two inequalities $x \leq y + M(1 - \delta)$ and $x > y + M\delta$ where δ is the binary decision variable and M is a sufficiently large number [29].

Ideally ranges need be combined with the relational domain of congruences [7] since then a range on one variable can be used to trim the range on another and vice versa. Congruence relations, that is, linear constraints that respect the modulo nature of computer arithmetic, can be computed prior to range analysis which leaves the problem of how to amalgamate them into a system of linear constraints. However the congruence $x = y \mod 2^k$ holds iff there exists an integer variable n such that $x = y + n2^k$. This suggests that integer linear programming could be the right medium for marrying congruences with ranges.

Iterative value set analyses have been proposed for binary code [5]. The work, like ours, is predicated on extracting high level predicates from low level conditional jumps. For example the instruction `cmp eax, edx` followed by a `ja` instruction causes a jump if `eax` > `edx`. These authors argue that the predicates can be extracted by pattern matching, a topic that is discussed elsewhere [7].

Interpolation has recently come to the fore in model checking [19] and techniques have now emerged for constructing interpolants in linear arithmetic [23]. Such techniques could be applied with range analysis to find combinations of range constraints that are inconsistent and hence diagnose unreachable code.

7 Conclusions

With an eye towards simplicity, we have shown how range analysis can be computed, not as the solution to a system of recursive equations, but as the solution of a system of constraints over min and max expressions. We have demonstrated how such constraints can be reduced to linear constraints, augmented with complementary constraints, and thus solved by repeated linear programming. The method can be implemented with an off-the-shelf linear programming package which can be used as a black-box. Furthermore, we have shown how the number of calls to the black-box can be reduced by using search heuristics. The result is an analysis that does not depend on classical fixpoint acceleration methods such as widening since it is designed to compute the fixpoint directly.

References

1. National Vulnerability Database. http://nvd.nist.gov.
2. H. R. Andersen. An Introduction to Binary Decision Diagrams. http://www.itu.dk/courses/AVA/E2005/bdd-eap.pdf, 1997.
3. A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge, 2002.
4. G. Balakrishnan and T. Reps. DIVINE: Discovering Variables in Executables. In *VMCAI*, volume 4349 of *LNCS*, pages 1–28. Springer, 2007.
5. G. Balakrishnan and T. W. Reps. WYSINWYX: What You See Is Not What You eXecute. *TOPLAS*, 32(6), 2010.
6. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *PLDI*, volume 38, pages 196–207. ACM, 2003.
7. J. Brauer, A .King, and S. Kowalewski. Range analysis of microcontroller code using bit-level congruences. In *FMICS*, volume 6371 of *LNCS*, pages 82–98. Springer, 2010.
8. L. Chen, A. Miné, J. Wang, and P. Cousot. Linear Absolute Value Relation Analysis. In *ESOP*, volume 6602 of *LNCS*, pages 156–175. Springer, 2011.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
10. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.

11. D. Doan. Commercial Off the Shelf (COTS) Security Issues and Approaches. Master's thesis, Naval Postgraduate School, Monterey, California, 2006. www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA456996.
12. T. Durden. Automated Vulnerability Auditing in Machine Code. *Phrack Magazine*, #64, 2007.
13. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*. The Internet Society, 2008.
14. D. Gopan and T. W. Reps. Lookahead Widening. In *CAV*, volume 4144 of *LNCS*, pages 452–466. Springer, 2006.
15. E. Goubault, S. Le Roux, J. Leconte, L. Liberti, and F. Marinelli. Static Analysis by Abstract Interpretation: A Mathematical Programming Approach. *ENTCS*, 267(1):73–87, 2010.
16. W. H. Harrison. Compiler Analysis for the Value Ranges of Varibles. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, 1977.
17. D. Kapur. Automatically Generating Loop Invariants using Quantifier Elimination. In *International Conference on Applications of Computer Algebra*, 2004.
18. K. R. M. Leino and F. Logozzo. Using Widenings to Infer Loop Invariants Inside an SMT Solver, Or: A Theorem Prover as Abstract Domain. In *WING*, pages 70–84, 2007.
19. K. L. McMillan. Applications of Craig Interpolants in Model Checking. In *TACAS*, volume 3440 of *LNCS*, pages 1–12. Springer, 2005.
20. T. W. Reps, G. Balakrishnan, and J. Lim. Intermediate-Representation Recovery from Low-Level Code. In *PEPM*, pages 100–111. ACM, 2006.
21. E. Rodriguez-Carbonell and D. Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. In *SAS*, volume 3148 of *LNCS*, pages 280–295. Springer, 2004.
22. R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *TOPLAS*, 27:185–235, 2005.
23. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. *Journal of Symbolic Computation*, 45:1212–1233, 2010.
24. B. Schlich. Model Checking of Software for Microcontrollers. *ACM Transactions in Embedded Computing Systems*, 9:1–27, 2010.
25. A. Simon and A. King. Widening Polyhedra with Landmarks. In *APLAS*, volume 4279 of *LNCS*, pages 166–182. Springer, 2006.
26. Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *TCS*, 345(1):122–138, 2005.
27. G. Weissenbacher. *Program Analysis with Interpolants*. PhD thesis, Magdalen College, 2010. http://ora.ouls.ox.ac.uk/objects/uuid:6987de8b-92c2-4309-b762-f0b0b9a165e6.
28. R. Wille, G. Fey, and R. Drechsler. Building Free Binary Decision Diagrams Using Sat Solvers. *Facta universitatis-series: Electronics and Energetics*, 20(3):381–394, 2007.
29. A. Zaks, Z. Yang, I. Shlyakhter, F. Ivancic, S. Cadambi, M. K. Ganai, A. Gupta, and P. Ashar. Bitwidth Reduction via Symbolic Interval Analysis for Software Model Checking. *IEEE TACAD*, 27(8):1513–1517, 2008.
30. Q. Zhong and N. Edward. Security Control COTS Components. *IEEE Computer Society*, 31:67–73, 1998.