Range Analysis of Whole Programs

Victor Hugo Sperle Campos, Igor Rafael de Assis Costa, Douglas do Couto Teixeira, Fernando Magno Quintão Pereira

UFMG – 6627 Antônio Carlos Av, 31.270-010, Belo Horizonte, Brazil
{victorsc,igor.rafael,douglas,fernando}@dcc.ufmg.br

Abstract. This paper describes an inter-procedural range analysis algorithm that scales up to programs with millions of assembly instructions. Contrary to many previous techniques, we handle comparisons between variables without resorting to relational lattices or expensive algorithms. We achieve path sensitiveness by using Bodik's Extended Static Single Assignment form as the intermediate representation. We claim to present in this paper the most extensive empirical evaluation of range analysis in an industrial strength compiler. We have implemented our technique in LLVM and have been able to process programs totaling 2.72 million lines of C in a few seconds. We do global program analysis, and achieve a simple form of context sensitiveness via function inlining.

1 Introduction

The analysis of integer variables on the interval lattice has been the canonical example of abstract interpretation since its introduction in Cousot and Cousot's seminal paper [8]. Compilers use range analysis to infer the possible values that discrete variables may assume during program execution. This analysis has many uses. For instance, it allows the optimizing compiler to remove from the program text redundant overflow tests [29] and unnecessary array bound checks [5]. Additionally, range analysis is essential not only to the bitwidth aware register allocator [2, 33], but also to more traditional allocators that handle registers of different sizes [16, 26, 27]. Finally, range analysis has also seen use in the static prediction of branches [25], to detect buffer overflow vulnerabilities [28, 36], to find the trip count of loops [20] and even in the synthesis of hardware [6, 19, 21].

Given this great importance, it comes as no surprise that the compiler literature is rich in works describing in details algorithmic variations of range analyses [14, 21, 30, 32]. On the other hand, none of these authors provide experimental evidence that their approaches are able to deal with very large programs. There are researchers who have implemented range analyses that scale up to large programs [4, 35]; nevertheless, because the algorithm itself is not the main focus of their works, they neither give details about their design choices nor provide experimental data about it. This scenario was recently changed by Oh *et al.* [24], who introduced an abstract interpretation framework which processes programs with hundreds of thousands of lines of code. Nevertheless, Oh *et al.* have designed a very simple range analysis, which does not handle comparisons

between variables, for instance. They also do not discuss the precision of their implementation, but only its runtime.

In this paper we provide a complete description of a range analysis algorithm, and show extensive experimental data that justifies our engineering choices. Our first algorithmic contribution on top of previous works is a three-phases approach to handle comparisons between variables without resorting to any exponential time technique. The few publicly available implementations of range analyses that we are aware of, such as those in FLEX [1] and gcc [2] only deal with comparisons between variables and constants. Even theoretical works, such as Su and Wagner's [32] or Gawlitza *et al.*'s [14] suffer from this limitation. This deficiency is one of the reasons explaining why none of these works has made their way into industrial-strength compilers. Two other insights allow our implementation to scale up to very large programs. We use Bodik's Extended Static Single Assignment (e-SSA) form [5] to perform range analysis sparsely. This program representation ensures that the interval associated with a variable is constant along its entire live range, and allows us to compute path-sensitive results. Finally, we process the strongly connected components that underline our constraint system in topological ordering. It is well-known that this technique is essential to speedup constraint solving algorithms [23, Sec 6.3]; however, we show that a careful propagation of information along strong components not only gives us speed, but also improves the precision of our results.

This work concludes a two years long effort to produce a solid and scalable implementation of range analysis. Our first endeavor to implement such an algorithm was based on Su and Wagner's constraint system, which can be solved exactly in polynomial time [31, 32]. However, although we could use their formulation to handle a subset of C-like constructs, their description of how to deal with loops was not very explicit. Thus, in order to solve loops we adopted Gawlitza *et al.*'s [14] approach. This technique uses the Bellman-Ford algorithm to detect increasing or decreasing cycles in the constraint system, and then saturates these cycles via a simple widening operator. A detailed description of our implementation has been published by Couto and Pereira [12]. Nevertheless, the inability to handle comparisons between variables, and the cubic complexity of the Bellman-Ford method eventually lead us to seek alternative solutions to range analysis. This quest reached a pinnacle in the present work.

We have implemented our algorithm in the LLVM compiler [18], and have used it to analyze a test suite with 2.72 million lines of C code. In Section 4 we provide empirical evidence showing that our implementation is fast: it globally analyzes the gcc source code in less than 15 seconds, for instance. It is also precise: our results are similar to Stephenson *et al*'s [30], even though out analysis does not require a backward propagation phase. Furthermore, we have been able to find tight bounds to the majority of the examples used by Costan *et al.* [7] and Lakhdar *et al.* [17], who rely on much more costly methods.

[1] The MIT's FLEX/Harpoon compiler provides an implementation of Stephenson's algorithm [30], and is available at `http://flex.cscott.net/Harpoon/`.

[2] Gcc's VRP pass implements a variant of Patterson's algorithm [25].

2 Background

Following Gawlitza *et al.*'s notation, we shall be performing arithmetic operations over the complete lattice $\mathcal{Z} = \mathbb{Z} \cup \{-\infty, +\infty\}$, where the ordering is naturally given by $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots + \infty$. For any $x > -\infty$ we define:

$x + \infty = \infty, x \neq -\infty$ $\quad x - \infty = -\infty, x \neq +\infty$
$x \times \infty = \infty$ if $x > 0$ $\quad x \times \infty = -\infty$ if $x < 0$
$0 \times \infty = 0$ $\quad (-\infty) \times \infty =$ not defined

From the lattice $\mathcal{Z}$ we define the product lattice $\mathcal{Z}^2$, which is partially ordered by the subset relation $\sqsubseteq$. $\mathcal{Z}^2$ is defined as follows:

$$\mathcal{Z}^2 = \emptyset \cup \{[z_1, z_2] |\ z_1, z_2 \in \mathcal{Z},\ z_1 \leq z_2,\ -\infty < z_2\}$$

Given an interval $\iota = [l, u]$, we let $\iota_\downarrow = l$, and $\iota_\uparrow = u$. We let $\mathcal{V}$ be a set of constraint variables, and $I : \mathcal{V} \mapsto \mathcal{Z}^2$ a mapping from these variables to intervals in $\mathcal{Z}^2$. Our objective is to solve a constraint system C, formed by constraints such as those seen in Figure 1(left). Figure 1(right) defines a valuation function e that computes $Y = f(\ldots)$, given I. Armed with these concepts, we define the range analysis problem as follows:

Definition 1. Range Analysis Problem
Input: *a set C of constraints ranging over a set $\mathcal{V}$ of variables.*
Output: *a mapping I such that, for any variable $V \in \mathcal{V}$, $e(V) = I[V]$.*

$$Y = [l, u] \qquad e(Y) = [l, u]$$

$$Y = \phi(X_1, X_2) \qquad \frac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [\min(l_1, l_2), \max(u_1, u_2)]}$$

$$Y = X_1 + X_2 \qquad \frac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [l_1 + l_2, u_1 + u_2]}$$

$$Y = X_1 \times X_2 \qquad \frac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2] \qquad L = \{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}}{e(Y) = [\min(L), \max(L)]}$$

$$Y = aX + b \qquad \frac{I[X] = [l, u] \qquad k_l = al + b \qquad k_u = au + b}{e(Y) = [\min(k_l, k_u), \max(k_l, k_u)]}$$

$$Y = X \sqcap [l', u'] \qquad \frac{I[X] = [l, u]}{e(Y) \leftarrow [\max(l, l'), \min(u, u')]}$$

Fig. 1. A suite of constraints that produce an instance of the range analysis problem.

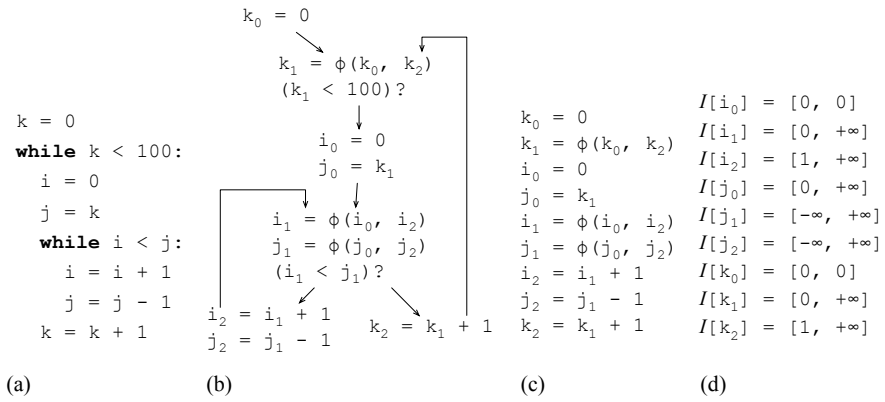


Fig. 2. (a) Example program. (b) Control Flow Graph in SSA form. (c) Constraints that we extract from the program. (d) Possible solution to the range analysis problem.

We will use the program in Figure 2(a) as the running example to illustrate our range analysis. Figure 2(b) shows the same program in SSA form [11], and Figure 2(c) outlines the constraints that we extract from this program. There is a clear correspondence between instructions and constraints. A possible solution to the range analysis problem, as obtained via the techniques that we will introduce in Section 3, is given in Figure 2(d). The SSA form, so conspicuous in modern compilers, leads to a very conservative solution. As we will see shortly, we can improve this solution substantially by using a more sophisticated program representation – the e-SSA form – which gives us path-sensitiveness.

3 Our Design of a Range Analysis Algorithm

In this section we explain the algorithm that we use to solve the range analysis problem. This algorithm involves a number of steps. First, we convert the program to a suitable intermediate representation that makes it easier to extract constraints. From these constraints, we build a dependence graph that allows us to do range analysis sparsely. Finally, we solve the constraints applying different fix-point iterators on this dependence graph. Figure 3 gives a global view of this algorithm. Some of the steps in the algorithm are optional. They improve the precision of the range analysis, at the expense of a longer running time.

Choosing a Program Representation. The solution to the range analysis problem in Figure 2 is imprecise because we did not take conditional tests into considerations. Branches give us information about the ranges that some variables assume, but only at *specific* program points. For instance, given a test such as $(k_1 < 100)$? in Figure 2(b), we know that $I[k_1] \sqsubseteq [-\infty, 99]$ whenever the condition is true. In order to encode this information, we might split the live range

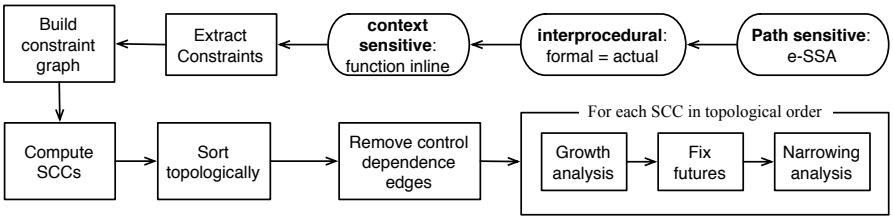


Fig. 3. Our implementation of range analysis. Rounded boxes are optional steps.

of k_1 right after the branching point; thus, creating two new variables, one at the path where the condition is true, and another where it is false. There is a program representation, introduced by Bodik *et al.* [5], that performs this live range splitting: the *Extended Static Single Assignment* form, or e-SSA for short.

Given that the exact rules to convert a program to e-SSA form have never been explicitly stated in the literature, we describe our rules as follows. Let $(v < c)?$ be a conditional test, and let l_t and l_f be labels in the program, such that l_t is the target of the test if the condition is true, and l_f is the target when the condition is false. We split the live range of v at any of these points if at least one of two conditions is true: (i) l_f or l_t dominate any use of v; (ii) there exist a use of v at the dominance frontier of l_f or l_t. For the notions of dominance and dominance-frontier, see Aho *et al.* [1, p.656]. To split the live range of v at l_f we insert at this program point a copy $v_f = v \sqcap [c, +\infty]$, where v_f is a fresh name. We then rename every use of v that is dominated by l_f to v_f. Dually, if we must split at l_t, then we create at this point a copy $v_t = v \sqcap [-\infty, c-1]$, and rename variables accordingly. If the conditional uses two variables, e.g., $(v_1 < v_2)?$, then we create intersections bound to *futures*. We insert, at l_f, $v_{1f} = v_1 \sqcap [\mathbf{ft}(v_2), +\infty]$, and $v_{2f} = v_2 \sqcap [-\infty, \mathbf{ft}(v_1)]$. Similarly, at l_t we insert $v_{1v} = v_1 \sqcap [-\infty, \mathbf{ft}(v_2) - 1]$ and $v_{2f} = v_2 \sqcap [\mathbf{ft}(v_1) + 1, +\infty]$.

We use the notation $\mathbf{ft}(v)$ to denote the *future* bounds of a variable. As we will show in Section 3.1, once the growth pattern of v is known, we can replace $\mathbf{ft}(v)$ by an actual value. Once we are done placing copies, we insert ϕ-functions into the transformed program to convert it to SSA form. This last step avoids that two different names given to the same original variable be simultaneously alive at the program code. Figure 4(a) shows our running example changed into standard e-SSA form. The part (b) of this figure shows the solution that we get to this new program. The e-SSA form allows us to bind interval information directly to the live ranges of variables; thus, giving us the opportunity to solve range analysis sparsely. More traditional approaches, which we call *dense analyses*, bind interval information to pairs formed by variables and program points.

Extracting Constraints. Our implementation handles 18 different assembly instructions. The constraints in Figure 1 show only a few examples. Instructions that we did not show include, for instance, the multiplicative operators `div` and

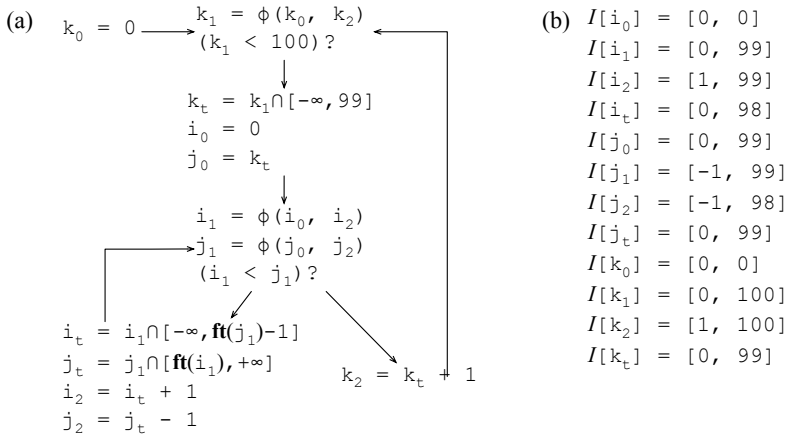


Fig. 4. (a) The control flow graph from Figure 2(b) converted to standard e-SSA form. (b) A solution to the range analysis problem

`modulus`, the bitwise operators `and`, `or`, `xor` and `neg`, the different types of shifts, and the logical operators `andalso`, `orelse` and `not`. Most of these instructions are sign-agnostic; that is, given that numbers are internally represented in 2's complement, the same implementation of a constraint handles positive and negative numbers. However, there are instructions that require different constraints, depending on the input being signed or not. Examples include `modulus` and `div`. We also handle different kinds of type conversion, e.g., converting 8-bit characters to 32-bit integers and vice-versa. In addition to constraints that represent actual assembly instructions, we have constraints to represent ϕ-functions, and intersections, as seen in Figure 1. The growth analysis that we will introduce in Section 3.1 require monotonic transfer functions. Many assembly operations, such as modulus or division, do not afford us this monotonicity. However, these non-monotonic instructions have conservative approximations [37].

The Constraint Graph. The main data structure that we use to solve the range analysis problem is a variation of Ferrante *et al.*'s *program dependence graph* [13]. For each constraint variable V we create a variable node N_v. For each constraint C we create a constraint node N_c. We add an edge from N_v to N_c if the name V is used in C. We add an edge from N_c to N_v if the constraint C defines the name V. Figure 5 shows the dependence graph that we build for the e-SSA form program given in Figure 4(a). If V is used by C as the input of a future, then the edge from N_v to N_c represents what Ferrante *et al.* call a *control dependence* [13, p.323]. We use dashed lines to represent these edges. All the other edges denote *data dependences* [13, p.322].

Strongly Connected Components. To solve range analysis we find all the strongly connected components (SCCs) of the dependence graph and collapse

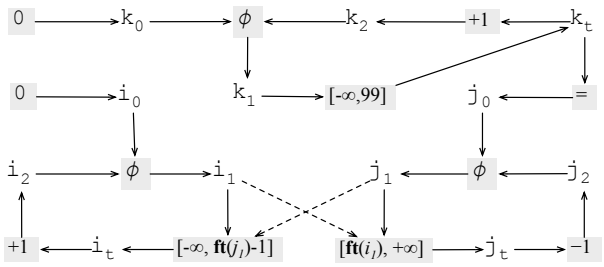


Fig. 5. The dependence graph that we build to the program in Figure 4.

them into single nodes; thus, obtaining a directed acyclic graph. We then sort the resulting DAG topologically, and apply the analyses from Section 3.1 on every SCC in topological order. Once we solve the range analysis problem for a SCC, we propagate the intervals that we found to the variable nodes at the *frontier* of this SCC. A variable node N_v is said to be in the frontier of a strongly connected component S if: (i) $N_v \notin S$; and (ii) there exists a variable node $N'_v \in S$, and a constraint node N_c, such that $N_v \leftarrow N_c$, and $N_c \leftarrow N'_v$. This propagation ensures that when analyzing a strongly connected component S any influence that S might suffer from nodes outside it has already been considered.

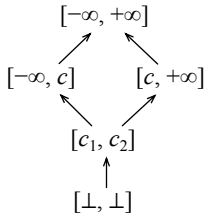
When finding strongly connected components, we take control dependence edges into consideration. For instance, in Figure 5 the nodes that correspond to the variables i_1, i_2, i_t, j_1, j_2 and j_t form a single component. The dashed edges, which represent control dependences, keep all these variables connected. In this way, we ensure that, upon stumbling upon an interval associated with future bounds, e.g., $\mathbf{ft}(v)$, either variable v has been solved in a previous component, or it belongs to the current component. In the latter case, as we will see soon, we can still take v's interval into consideration. As we show in Section 4, most of the strong components in actual programs are singletons. Furthermore, the composite components tend to be small. These two facts ensure that the more costly parts of our algorithm only have to handle small inputs.

3.1 Finding Ranges in Strongly Connected Components

Given a strongly connected component of the dependence graph with N nodes, we solve the range analysis problem in three-steps:

1. Run growth analysis: $O(N)$.
2. Fix intersections: $O(N)$.
3. Apply the narrowing operator: $O(N^2)$.

However, before we start, we remove control dependence edges from the strongly connected component, as they have no semantics to our transfer functions.



$$\frac{I[Y] = [\bot, \bot]}{I[Y] \leftarrow e(Y)} \qquad \frac{e(Y)_\downarrow < I[Y]_\downarrow \qquad e(Y)_\uparrow > I[Y]_\uparrow}{I[Y] \leftarrow [-\infty, +\infty]}$$

$$\frac{e(Y)_\downarrow < I[Y]_\downarrow}{I[Y] \leftarrow [-\infty, I[Y]_\uparrow]} \qquad \frac{e(Y)_\uparrow > I[Y]_\uparrow}{I[Y] \leftarrow [I[Y]_\downarrow, +\infty]}$$

Fig. 6. (Left) The lattice of the growth analysis. (Right) Cousot and Cousot's widening operator. We evaluate the rules from left-to-right, top-to-bottom, and stop upon finding a pattern matching. Again: given an interval $\iota = [l, u]$, we let $\iota_\downarrow = l$, and $\iota_\uparrow = u$

Growth Analysis. The first step of our algorithm consists in determining how the interval bound to each variable grows. We ensure termination of this phase via Cousot and Cousot's widening operator [8, p.247]. The possible behaviors of an interval are: (i) does not change; (ii) grows towards $+\infty$; (iii) grows towards $-\infty$; and (iv) grows in both directions. The lattice of abstract states, plus a constraint system representing our growth analysis is given in Figure 6. Because the lattice has height three, the intervals bound to each variable can change at most three times. However, as we show in Figure 13(Bottom), we can be more precise if we evaluate the constraints a few times before applying widening.

Fixing futures. The ranges found by the growth analysis tells us which variables have fixed bounds, independent on the intersections in the constraint system. Thus, we can use actual limits to replace intersections bounded by futures. Figure 7 shows the rules to perform these substitutions. In order to correctly replace a future $\mathbf{ft}(V)$ that limits a variable V', we need to have already applied the growth analysis onto V. Had we considered only data dependence edges, then it would be possible that V' be analyzed before V. However, because of control dependence edges, this case cannot happen. The control dependence edges ensure that any topological ordering of the constraint graph either places N_v before $N_{v'}$, or places these nodes in the same strongly connected component. For instance, in Figure 5, variables J_1 and I_t are in the same SCC only because of the control dependence edges.

Narrowing Analysis. The growth analysis associates very conservative bounds to each variable. Thus, the last step of our algorithm consists in narrowing these intervals. We accomplish this step via Cousot and Cousot's classic narrowing operator [8, 248], which we show in Figure 8.

Example: Continuing with our example, Figure 9 shows the application of our algorithm on the last strong component of Figure 5. Upon meeting this SCC, we have already determined that the interval $[0, 0]$ is bound to I_0 and that the interval $[100, 100]$ is bound to J_0, as we show in Figure 5(a). We are not guaranteed to find the least fix point of a constraint system. However, in this example we did it. We emphasize that finding this tight solution was only possible

$$\frac{Y = X \sqcap [l, \mathbf{ft}(V) + c] \qquad I[V]_\uparrow = u}{Y = X \sqcap [l, u + c]} \quad u, c \in \mathbb{Z} \cup \{-\infty, +\infty\}$$

$$\frac{Y = X \sqcap [\mathbf{ft}(V) + c, u] \qquad I[V]_\downarrow = l}{Y = X \sqcap [l + c, u]} \quad l, c \in \mathbb{Z} \cup \{-\infty, +\infty\}$$

Fig. 7. Rules to replace futures by actual bounds. S is the interval bound to each variable after the widening analysis.

$$\frac{I[Y]_\downarrow = -\infty \qquad e(Y)_\downarrow > -\infty}{I[Y] \leftarrow [e(Y)_\downarrow, I[Y]_\uparrow]} \qquad \frac{I[Y]_\downarrow > e(Y)_\downarrow}{I[Y] \leftarrow [e(Y)_\downarrow, I[Y]_\uparrow]}$$

$$\frac{I[Y]_\uparrow = +\infty \qquad e(Y)_\uparrow < +\infty}{I[Y] \leftarrow [I[Y]_\downarrow, e(Y)_\uparrow]} \qquad \frac{I[Y]_\uparrow < e(Y)_\uparrow}{I[Y] \leftarrow [I[Y]_\downarrow, e(Y)_\uparrow]}$$

Fig. 8. Cousot and Cousot's narrowing operator.

because of the topological ordering of the constraint graph in Figure 5. Had we applied the widening operator onto the whole graph, then we would have found out that variable j_0 is bound to $[-\infty, +\infty]$, because (i) it receives its interval directly from variable k_t, which is upper bounded by $+\infty$, and (ii) it is part of a negative cycle. On the other hand, because we only analyze j's SCC after we have analyzed k's, k only contribute the constant range $[0, 99]$ to j_0.

Figure 4(b) shows our final solution for this example. This solution is very precise, in the sense that it is the maximum fixed point of the constraint system given in Figure 1. However, the solution is still an over approximation of the dynamic behavior of the program in Figure 4. For instance, we have found that variable i could reach the upper value of 99. In any actual run of the program, i could be at most 50. Analysis on relational lattices, such as the polyhedron [9] or the Octagon [22] domains, can infer such tighter bounds, as shown by Lakhdar-Chaouch *et al.* [17]. However, analyses on these higher dimensional domains are much more computationally expensive than analyses on the interval domain [24].

4 Experiments

We have implemented our range analysis algorithm in LLVM 3.0, and have run experiments on a Intel quad core CPU with a 2.40GHz clock, and 3.6GB of RAM. Each core has a 4,096KB L1 cache. We use Linux Ubuntu 10.04.4. Our implementation of range analysis has 3,958 lines of commented C++ code, and our e-SSA conversion module has 672 lines. We have analyzed 428 C programs that constitute the LLVM test suite plus the integer benchmarks in SPEC CPU 2006. Together, these programs contain 4.76 million assembly instructions.

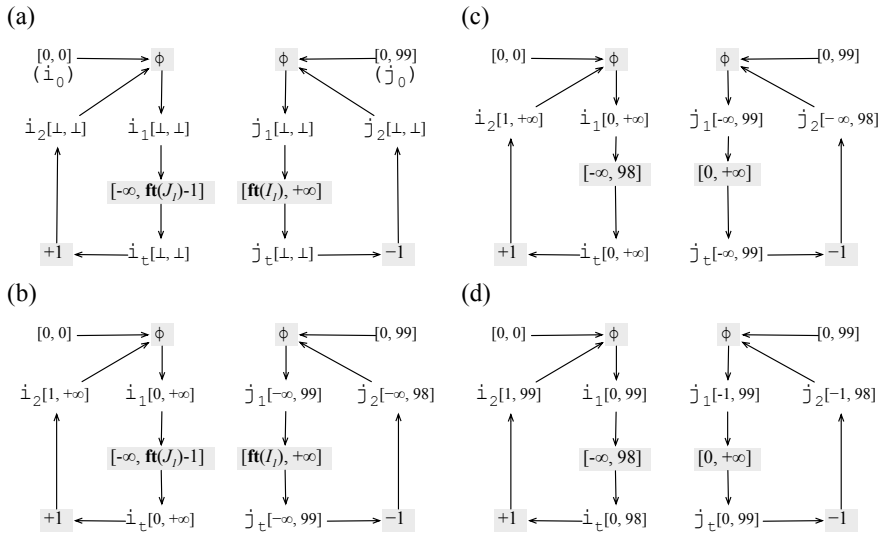


Fig. 9. Four snapshots of the last SCC of Figure 4. (a) After removing control dependence edges. (b) After running the growth analysis. (c) After fixing the intersections bound to futures. (d) After running the narrowing analysis.

Structural Properties. Table 1 shows data taken from the SPEC 2006 integer benchmark suite for the inter-procedural version of our analysis. Three observations provide evidence that our algorithm is linear in practice, as we will show in Figure 10. First, most of the strongly connected components found in constraint graphs are singletons. For instance, 99.11% of the SCCs in gcc (`403.gcc`) have only one node. Hence, the most complex phases of our algorithm are confined to a small part of the constraint graph. Second, the strongly connected components are not large – our largest component, found in `403.gcc`, has 2,131 nodes. It exists due to a long chain of mutually recursive function calls. We do not count control dependency edges when measuring the size of SCCs, as they have no impact on the algorithm explained in Section 3.1. Third, as we see in the table, any node was visited at most three times by the narrowing operator.

Asymptotic complexity. Figure 10 provides a visual comparison between the time to run our algorithm and the size of the input programs. We show data for the 100 largest benchmarks in our test suite, in number of variable nodes in the constraint graph. We perform function inlining before running our analysis, to increase program sizes. Each point in the X line corresponds to a benchmark. We analyze the smallest benchmark in this set, `Prolangs-C`, which has 1,131 variable nodes in the constraint graph, in 20ms. We take 15,91secs to analyze our largest benchmark, `403.gcc`, which, after function inlining, has 1,266,273 assembly instructions, and a constraint graph with 679,652 variable nodes. For

Name	464	473	483	458	429	471	403	445	462	401	456
#I	139626	8900	593765	26947	2738	94036	887191	141330	6237	17663	61736
$\#N_v$	97494	5490	352423	14407	1853	50095	449442	84846	3293	12517	38409
$\#N_c$	69530	3799	213535	11164	1156	28943	309310	63506	2336	8731	24530
#S	91731	5108	346451	13309	1751	48784	435372	79909	3003	11539	35619
L	38	49	120	116	16	258	2131	311	9	18	41
#U	89461	4983	344132	12919	1726	48400	431703	78407	2885	11267	34631
#V	3	2	2	2	2	2	2	3	2	2	2

Table 1. Structural Properties of the programs in the SPEC CPU 2006 integer benchmark suite. #I: number of assembly instructions in the original program. $\#N_v$: number of variable nodes in the constraint graph. $\#N_c$: number of constraint nodes in the constraint graph. #S: number of strongly connected components (SCC). L: size of the largest SCC. #U: number of SCCs with only one node. #V: maximum number of times any variable node was visited by narrowing.

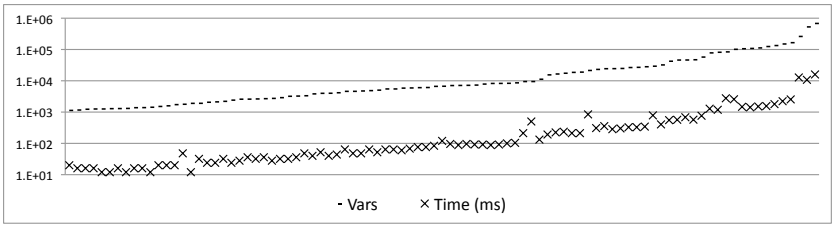


Fig. 10. Correlation between program size (measured in number of nodes in constraint graphs after inlining) and analysis runtime (ms). Coefficient of determination = 0.967.

this data set, the coefficient of determination (R^2) is 0.967, which provides very strong evidence about the linear asymptotic complexity of our implementation.

Time distribution. Figure 11(Left) shows the distribution of processing time among the main phases of the inter-procedural version of our algorithm. Functions are inlined to increase program sizes. We analyze the SPEC 2006 integer benchmarks in 41.42secs. The time to build the e-SSA representation is 7.03secs; the time to extract constraints from the program and build the constraint graphs is 7.87secs; the time to find SCCs via Nuutila's algorithm and sort them topologically is 10.55secs, and the time to run the analysis itself is 15.96secs.

Intra vs Inter-procedural runtimes. As we saw in Figure 3, our implementation supports intra- and inter-procedural execution modes. On top of these modes, we can optionally perform function inline to obtain limited context-sensitiveness. Our baseline compiler, LLVM, is able to inline non-recursive function calls. Figure 11(Right) compares three different execution modes. Times are normalized to the time to run the intra-procedural analysis without inlining. On the average, the intra-procedural mode is 28.92% faster than the inter-procedural mode. If we perform function inlining, then this difference is larger: 45.87%. These numbers are close because we run our analysis on strong components.

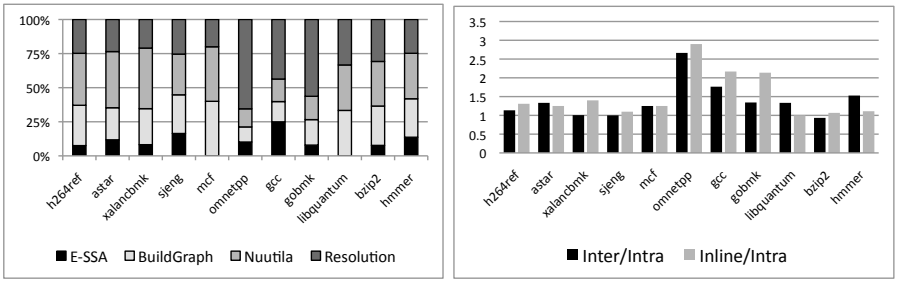


Fig. 11. (Left) Time distribution across different phases of our inter+inline algorithm. (Right) Runtime comparison between the intra, inter e inter+inline versions of our algorithm. The bars are normalized to the time to run the intra-procedural version.

The impact of strong components in time and precision. Figure 12(a) shows that strong components dramatically decrease the runtime of our analysis. For instance, this design improves the analysis time of `483.xalancbmk` 452 times! Moreover, as the example in Section 3.1 shows, our algorithm benefits from the partitioning of constraint graphs into SCCs to be more precise. Figure 12(b) measures this precision. Range propagation along strong components allows us, for example, to reduce 41.5% more bits in `401.bzip2`.

Impact of the e-SSA on runtime. Figure 12(c) shows that the e-SSA conversion increases the size of the SPEC 2006 integer benchmarks by 6.52%. This growth is due to the σ-functions used to split live ranges after conditionals, and to the extra ϕ-functions necessary to preserve the single static assignment property. As a consequence of the program growth, the e-SSA form increases the runtime of our analysis by 7.11%, as seen in Figure 12(d). We are comparing only the time to perform range analysis. The bars do not include the time to perform the e-SSA conversion, which we showed in Figure 11(Left).

The impact of whole program analysis on precision. Figure 13(Top) shows that the whole program analysis increases moderately the precision of our analysis. We show results for the five largest programs in three different categories of benchmarks: SPEC CPU 2006, the Stanford Suite [3] and Bitwise [30]. Our initial goal when developing this analysis was to support a bitwidth-aware register allocator. Thus, we measure precision by the average number of bits that our analysis allows us to save per program variables. It is very important to notice that we do not consider constants in our statistics of precision. In other words, we only measure bitwidth reduction in variables that a constant propagation step could not remove. Our results for the SPEC programs were disappointingly: on the average, the intra-procedural version of our analysis saves 5.23% of the total program bitwidth. By using the inter-procedural version, with function inlining to simulate context-sensitiveness, we can increase this number to 8.89%. A manual inspection of the SPEC programs reveal that this result is expected:

[3] `http://classes.engineering.wustl.edu/cse465/docs/BCCExamples/stanford.c`

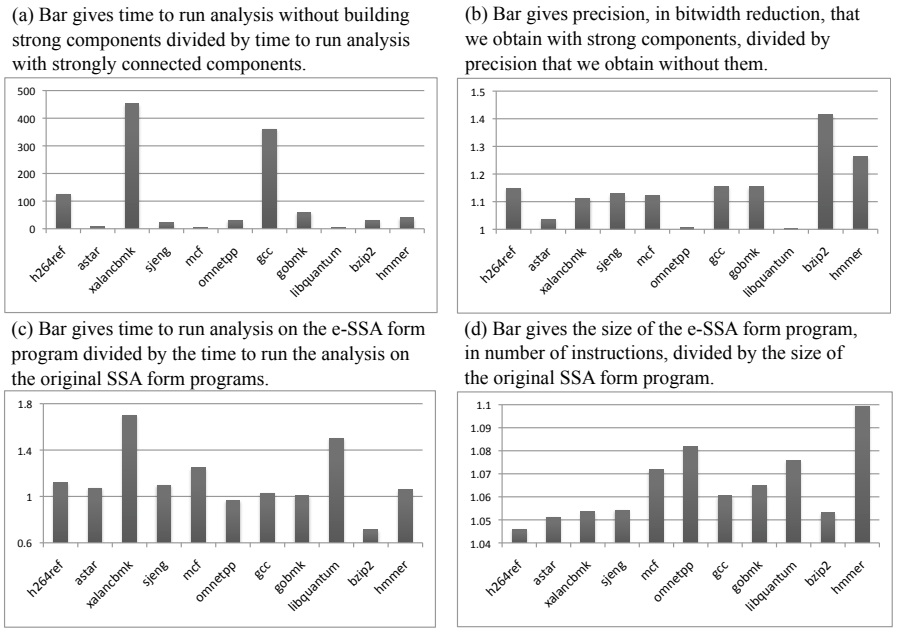


Fig. 12. How strong components and the e-SSA representation improve our algorithm. This data refer to the inter-procedural analysis with function inlining.

they manipulate files, and their code do no provide enough explicit constants to power our analysis up. However, with numerical benchmarks we fare much better. On the average our inter-procedural algorithm reduces the bitwidth of the Stanford benchmarks by 36.24%. Notice that this number is better than the one we get with inlining (31.97%), because functions in those benchmarks are called only at one location. Finally, for Bitwise we obtain a bitwidth reduction of 12.27%. However, this average is brought down by two outliers: `edge_detect` and `sha`, which cannot be reduced. The bitwise benchmarks were implemented by Stephenson *et al.* [30] to validate their bitwidth analysis. Our results are on par with those found by the original authors. The bitwise programs contain only the `main` function; thus, different versions of our algorithm find the same results.

The impact of e-SSA on precision. Figure 13(Middle) shows that the e-SSA transformation improves the precision of our analysis dramatically. If we convert the input programs from the original SSA representation used by LLVM to the e-SSA format, then, in some cases, e.g., `Stanford.queens`, we can increase bitwidth reduction by 154x. This impressive difference happens because the standard SSA representation does not give our analysis any subsidy to acquire information from the outcome of conditionals.

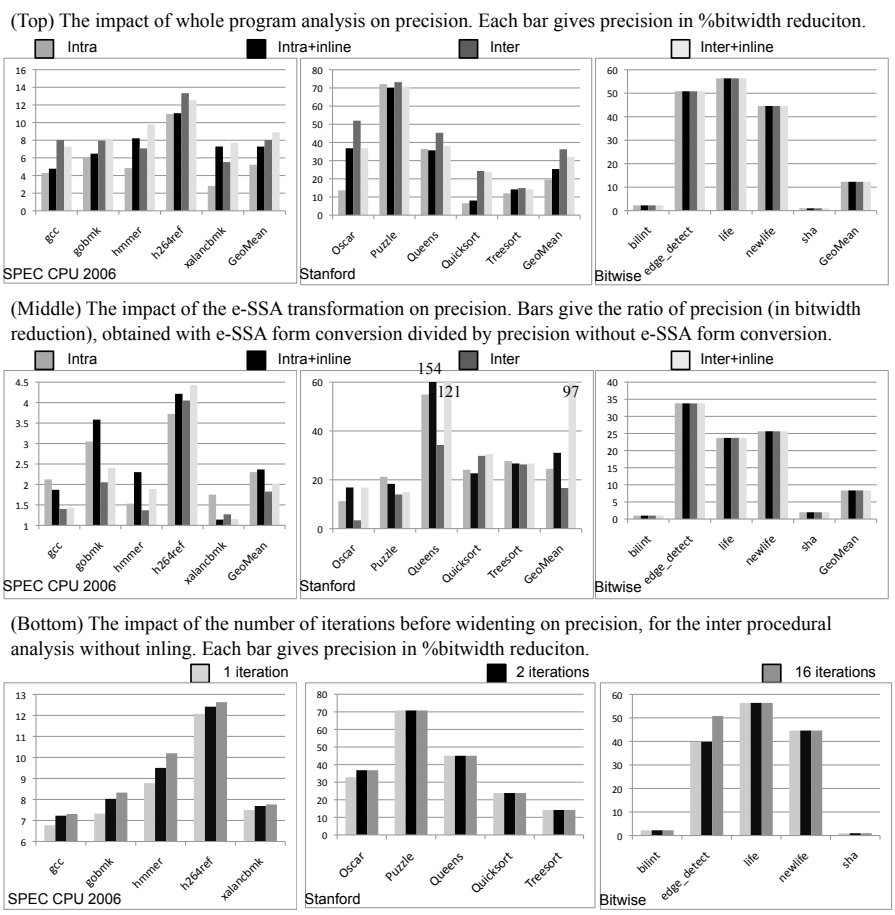


Fig. 13. The impact of different design trade-offs on precision of range analysis.

The impact of the number of iterations before widening on precision. It is well-known that premature widening might cause loss of information [10, Sec5.2]. Figure 13 shows how the precision of our analysis varies with the number of iterations used before resorting to widening. We have applied the widening operator in Figure 6 after each node was visited 1, 2 and 16 times.

5 Related works

Range analysis is an old ally of compiler writers. The notions of widening and narrowing were introduced by Cousot and Cousot in one of the most cited papers in computer science [8]. Different algorithms for range analysis have been later

proposed by Patterson [25], Stephenson *et al.* [30], Mahlke *et al.* [21] and many other researchers. Recently there have been many independent efforts to find exact, polynomial time algorithms to solve constraints on the interval lattice [7, 14, 17, 31, 32]. However, these works are still very theoretical, and have not yet been used to analyze large programs. Lakhdar *et al.*'s relational analysis [17], for instance, takes about 25 minutes to go over a program with almost 900 basic blocks. We analyze programs of similar size in seconds. We do not claim our approach is as precise as such algorithms, even though we are able to exactly analyze 4/5 of the examples presented in [17]. On the contrary, this paper presents a compromise between precision and speed that scales to very large programs.

There have been many practical approaches to abstract interpretation, with special emphasis on range analysis [3, 4, 10, 15]. Cousot's group, for instance, has been able to globally analyze programs with thousands of lines of code, albeit they use domain specific tools. Astré, for example, analyzes programs that do not contain recursive calls. The work that is the closest to us is Oh *et al.*'s very recent abstract interpretation framework [24]. Oh *et al.* discuss an implementation of range analysis on the interval lattice that scales up to a program with 1,363K LoC (ghostscript-9.00), but they do not provide results about its precision. We could not find the benchmarks used in those experiments for a direct comparison – the distribution of ghostscript-9.00 available in the LLVM test suite has 27K LoC. On the other hand, we globally analyzed our largest benchmark, SPEC CPU 2006's `gcc`, enabling function inlining, in less than 15secs. Oh *et al.*'s implementation took orders of magnitude more time to go over programs of similar size. However, whereas they provide a framework to develop general sparse analyses, we only solve range analysis on the interval lattice.

6 Final Considerations

In this presentation we chose to omit correctness proofs for our algorithms. For a proof that the widening and the narrowing operators give origin to approximating sequences, we recommend Cousot and Cousot's work [8]. For a proof that the e-SSA form transformation is semantics preserving, we invite the reader to check a recent report of ours [34]. In this work we also show that the results that we obtain via the sparse analysis are equivalent to the results provided by a dense analysis. In other words, if the dense analysis tells us that variable v is associated with the interval range $[l, u]$ at program point i, and v' is the new name of v, alive at i in the e-SSA form program, then v' is associated with the interval $[l, u]$ along its entire live range. Additionally, we have used a dynamic profiler, available in our webpage, to empirically validate our results.

Our implementation of the range analysis algorithm described in this paper is publicly available at `http://code.google.com/p/range-analysis/`. This repository contains instructions about how to deploy and use our implementation. We provide a gallery of examples, including source codes, CFGs and constraint graphs that we produce for meaningful programs at `http://code.google.com/p/range-analysis/wiki/gallery`.

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.
2. Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. Optimal bitwise register allocation using integer linear programming. In *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2006.
3. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *I@A*, pages 1–38. AIAA, 2010.
4. 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*, pages 196–207. ACM, 2003.
5. Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.
6. Jason Cong, Yiping Fan, Guoling Han, Yizhou Lin, Junjuan Xu, Zhiru Zhang, and Xu Cheng. Bitwidth-aware scheduling and binding in high-level synthesis. *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, 2:856–861, 18-21 Jan. 2005.
7. Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *CAV*, pages 462–475, 2005.
8. 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.
9. P. Cousot and N.. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.
10. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Form. Methods Syst. Des.*, 35(3):229–264, 2009.
11. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.
12. Douglas do Couto Teixeira and Fernando Magno Quintao Pereira. The design and implementation of a non-iterative range analysis algorithm on a production compiler. In *SBLP*, pages 45–59. SBC, 2011.
13. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.
14. Thomas Gawlitza, Jerome Leroux, Jan Reineke, Helmut Seidl, Gregoire Sutre, and Reinhard Wilhelm. Polynomial precise interval analysis revisited. *Efficient Algorithms*, 1:422 – 437, 2009.
15. Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. In *SAS*, pages 203–217, 2005.
16. Timothy Kong and Kent D Wilken. Precise register allocation for irregular architectures. In *MICRO*, pages 297–307. IEEE, 1998.
17. Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. Widening with thresholds for programs with complex control graphs. In *ATVA*, pages 492–502. Springer-Verlag, 2011.

18. Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.
19. G Lhairech-Lebreton, P Coussy, D. Heller, and E. Martin. Bitwidth-aware high-level synthesis for designing low-power dsp applications. In *ICECS*, pages 531–534. IEEE, 2010.
20. Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO*, pages 136–146, 2009.
21. S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1355–1371, 2001.
22. Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100, 2006.
23. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.
24. Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for c-like languages. In *PLDI*, page to appear. ACM, 2012.
25. Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78. ACM, 1995.
26. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI*, pages 216–226. ACM, 2008.
27. Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *LCTES/SCOPES*, pages 139–148. ACM, 2002.
28. Axel Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer, 1th edition, 2008.
29. Marcos Rodrigo Sol Souza, Christophe Guillon, Fernando Magno Quintao Pereira, and Mariza Andrade da Silva Bigonha. Dynamic elimination of overflow tests in a trace compiler. In *CC*, pages 2–21, 2011.
30. Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM, 2000.
31. Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In *TACAS*, pages 280–295, 2004.
32. Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science*, 345(1):122–138, 2005.
33. Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. In *POPL*, pages 85–96, New York, NY, USA, 2003. ACM.
34. Andre L. C. Tavares, Benoit Boissinot, Mariza A. S. Bigonha, Roberto Bigonha, Fernando M. Q. Pereira, and Fabrice Rastello. A program representation for sparse dataflow analyses. *Science of Computer Programming*, X(X):2–25, 201X. Invited paper with publication expected for 2012.
35. Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded c programs. *SIGPLAN Not.*, 39:231–242, 2004.
36. David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, pages 3–17. ACM, 2000.
37. Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., 2002.