Pointer-Range Analysis*

Suan Hsi Yong and Susan Horwitz

Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA
{suan,horwitz}@cs.wisc.edu

Abstract. Array-Range Analysis computes at compile time the range of possible index values for each array-index expression in a program. This information can be used to detect potential out-of-bounds array accesses and to identify non-aliasing array accesses. In a language like C, where arrays can be accessed indirectly via pointers, and where pointer arithmetic is allowed, range analysis must be extended to compute the range of possible values for each pointer dereference.
This paper describes a Pointer-Range Analysis algorithm that computes a safe approximation of the set of memory locations that may be accessed by each pointer dereference. To properly account for non-trivial aspects of C, including pointer arithmetic and type-casting, a range representation is described that separates the identity of a pointer's target location from its type; this separation allows a concise representation of pointers to multiple arrays, and precise handling of mismatched-type pointer arithmetic.

1 Introduction

The goal of Array-Range Analysis is to compute (at compile time) the range of possible index values for each array-index expression in a program. This information can be used for many applications, such as:

- Eliminating unnecessary or redundant bounds-checking operations, for code optimization [13, 26];
- Detecting potential out-of-bounds access errors, for debugging, program verification, or security [25, 17];
- Identifying non-aliasing array accesses, for program understanding, optimization, or parallelization [1, 24, 14].

The importance of Array-Range Analysis is reflected in the extensive body of research conducted over the last three decades. However, most previous work has focused on languages like Fortran and Java.

The C language presents new challenges for array-range analysis. First, arrays can be accessed indirectly via pointers, so pointer arithmetic becomes an alternative way to compute the index into an array. Second, type-casts and unions

* This work was supported in part by the National Science Foundation under grants CCR-9987435 and CCR-0305387.

allow an array of one type to be accessed as an array of a different type, possibly with a different size. Third, even deciding what is an "array" is difficult, especially with heap allocated storage, where the same mechanism (a call to `malloc`) is used whether one is allocating a single object or an array of objects. This also means that a pointer dereference to a single object and a pointer dereference to an array object cannot always be syntactically differentiated.

Given these features, we approach the problem of range analysis for C by treating *all* pointer dereferences as array accesses, and by treating each solitary object as an array with one element. Since an array indexing expression `a[i]` is semantically equivalent to the dereference `*(a+i)`, the analysis can be described purely in terms of pointer dereferences and pointer arithmetic, rather than array accesses and array-index computation; hence the name Pointer-Range Analysis.

This paper describes a Pointer-Range Analysis algorithm to compute, for each dereference in a program, a safe approximation of the set of memory locations that may be accessed by the dereference. An abstract representation of ranges is presented that can safely and portably handle challenging aspects of analyzing C pointers, including:

- Pointer arithmetic.
- Type mismatches, which arise due to unions or casts.
- Imprecise points-to information, where a pointer may point to one of several arrays.

The pointer-range representation has three components: target location, target type, and offset range. The separate tracking of the target's location and type allows a single location to be treated as different types, and allows precise type information to be maintained when location information is lost as a result of analysis imprecision. Maintaining types rather than numeric values of sizes preserves portability, allowing the analysis to be applied when exact sizes of types cannot be assumed. Experimental results are presented that show the potential utility of pointer-range analysis in various contexts, such as eliminating unnecessary bounds checks and identifying non-aliasing accesses.

2 Representing Ranges

We define Pointer-Range Analysis as a forward dataflow-analysis problem, where at each edge of the control-flow graph (CFG), a mapping is maintained from each location x to an abstract representation of the range of values x may hold at runtime. This abstract representation must be a safe approximation (represent a superset) of the actual range of possible values. We follow the convention in dataflow analysis of performing a meet ($\sqcap$) at control-flow merge points, so the elements of the abstract domain must be partially ordered such that $r_1 \sqsubseteq r_2$ implies r_1 is more approximate than r_2; i.e., the range represented by r_1 is a superset of the range represented by r_2.

When dealing only with numeric values, the Integer Interval Domain can be used to represent the ranges:

Integer Interval Domain $\mathcal{I} = \langle I, \sqsubseteq_i \rangle$:

- $I = \{[min, max] \mid min, max \in \mathbb{Z} \cup \{+\infty, -\infty\}, min \leq max\} \cup \{\emptyset\}$
- $[min, max]$ represents the set of integer values in the range $min \ldots max$.
- $[min_1, max_1] \sqsubseteq_i [min_2, max_2]$ iff $[min_1, max_1] \supseteq [min_2, max_2]$, that is, $min_1 \leq min_2$ and $max_1 \geq max_2$.
- Note that $\mathcal{I}$ is a lattice that satisfies our approximation requirement, with top element $\top = \emptyset$, bottom element $\bot = [-\infty, +\infty]$, and meet operator $\sqcap = \cup$.

Figure 1(a) demonstrates the analysis of a simple example using intervals, and shows how the computed range can be used to decide whether an array index is in bounds. When assigning a constant value to `i`, we can map `i` to a precise interval representation (e.g., $[6, 6]$ at line 4). When the two branches merge at line 8, we take the meet (union) of `i`'s range along the two incoming branches to get an approximate (superset) range $[3, 6]$ of possible values for `i`. Since this falls within the legal range of $[0, 9]$ for indexing into array `a`, the array index at line 9 is guaranteed to be in-bounds.

When dealing with pointers, however, our abstract domain must be able to capture information about a pointer's target (the object to which the pointer may point). As a first step, we define the set *Loc* of abstract representatives of locations, or objects, defined in the program, to which a pointer may legally point:

Locations *Loc*:

- $Loc = \{v \mid$ `v` is a variable in the program $\}$
  $\cup \{\text{MALLOC}_i \mid i$ is a program point where `malloc` is called $\}$

Each location $x \in Loc$ is treated as an array object, with an associated element type τ_x and element count σ_x (for solitary objects, the element count is 1). For a heap location MALLOC_i, which represents all heap objects allocated at program point i, it may not be possible to determine a precise type and count; these values are inferred from the argument to `malloc` as follows: if the argument is a constant C, we set the type to `char` and count to C; if it is of the form

1. `int a[10];`	
2. `int i;`	
3. `if(...){`	
4. `   i = 6;`	`i` $\mapsto [6, 6]$
5. `} else {`	
6. `   i = 3;`	`i` $\mapsto [3, 3]$
7. `}`	
8.	`i` $\mapsto [3, 6]$
9. `a[i] = 0;`	(in-bounds)

(a) Arrays, with Integer Intervals

1. `int a[10];`	
2. `int * p;`	
3. `if(...){`	
4. `   p = &a[6];`	`p` $\mapsto \langle$`a`$, [6, 6]\rangle$
5. `} else {`	
6. `   p = &a[3];`	`p` $\mapsto \langle$`a`$, [3, 3]\rangle$
7. `}`	
8.	`p` $\mapsto \langle$`a`$, [3, 6]\rangle$
9. `*p = 0;`	(in-bounds)

(b) Pointers, with Location-Offset

Fig. 1. In-Bounds Access Example

	Location-Offset	Descriptor-Offset
1. `int a[10], b[8];`		
2. `int * p;`		
3. `if(...){`		
4. `p = &a[6];`	$\mathtt{p} \mapsto \langle \mathtt{a}, [6,6] \rangle$	$\mathtt{p} \mapsto \langle \mathtt{a} : \mathtt{int[10]}, [6,6] \rangle$
5. `} else {`		
6. `p = &b[3];`	$\mathtt{p} \mapsto \langle \mathtt{b}, [3,3] \rangle$	$\mathtt{p} \mapsto \langle \mathtt{b} : \mathtt{int[8]}, [3,3] \rangle$
7. `}`		
8.	$\mathtt{p} \mapsto \bot$	$\mathtt{p} \mapsto \langle \text{UNKNOWN} : \mathtt{int[8]}, [3,6] \rangle$
9. `*p = 0;`	(don't know)	(in-bounds)

Fig. 2. Multiple Target Example

C `* sizeof(`τ`)` we set the type to τ and count to C; otherwise, we set the type to `void` and count to 0.

We now define a Location-Offset domain whose elements represent a pointer to a location plus an offset:

Location-Offset Domain $\mathcal{LO} = \langle LO, \sqsubseteq_{lo} \rangle$:

- $LO = (Loc \cup \{\text{NULL}\}) \times I$
- The element $\langle x, [min, max] \rangle$ represents the address of location x plus an offset in the range $[min, max]$; i.e., the range $[\&\mathtt{x} + min \cdot |\tau_x|, \&\mathtt{x} + max \cdot |\tau_x|]$, where τ_x is the static element type of location x, and $|\tau|$ is shorthand for `sizeof(`τ`)`, the size of τ in bytes.
- A NULL-targeted element $\langle \text{NULL}, [min, max] \rangle$ represents the integer range $[min, max]$.
- $\langle l_1, o_1 \rangle \sqsubseteq_{lo} \langle l_2, o_2 \rangle$ iff $l_1 = l_2$ and $o_1 \sqsubseteq_i o_2$.
- $\mathcal{LO}$ can be converted to a lattice $\mathcal{LO}^{\mathrm{L}}$ by adding a "top" element ($\top$) and a "bottom" element ($\bot$).

Figure 1(b) shows a program that has the same behavior as the program in Figure 1(a), but which uses a pointer to indirectly access the array. At line 4, we map `p` to the location-offset range $\langle \mathtt{a}, [6,6] \rangle$ which represents the constant value $\&\mathtt{a} + 6 \cdot |\mathtt{int}|$. At line 8, the meet operation yields the range $[\&\mathtt{a} + 3 \cdot |\mathtt{int}|, \&\mathtt{a} + 6 \cdot |\mathtt{int}|]$ of possible values for `p`. At line 9, when `p` is dereferenced, since `a` has 10 elements, we can verify that the range of `p` falls within the legal range $[\&\mathtt{a} + 0 \cdot |\mathtt{int}|, \&\mathtt{a} + 9 \cdot |\mathtt{int}|]$, thus `*p` will be in-bounds.

The Location-Offset representation has two weaknesses. First, it can only represent a pointer to a single target location. Consider the example in Figure 2, where `p` is assigned to point to two different arrays, `a` and `b`, along the two branches. Using the location-offset representation, the merge point at line 8 would map `p` to $\bot$, since the elements $\langle \mathtt{a}, [6,6] \rangle$ and $\langle \mathtt{b}, [3,3] \rangle$ from the two incoming branches are $\sqsubseteq_{lo}$-incomparable, and thus the dereference at line 9 cannot be determined to be in-bounds.

The second weakness of the Location-Offset representation is that it may lose precision when handling pointer arithmetic with mismatched types. Consider

	Location-Offset	Descriptor-Offset
1. `int a[2];`		
2. `char *p, *q;`		
3. `p = (char *)&a[0];`	$\texttt{p} \mapsto \langle \texttt{a}, [0,0] \rangle$	$\texttt{p} \mapsto \langle \texttt{a} : \texttt{int[2]}, [0,0] \rangle$
4. `q = p + 6;`	$\texttt{q} \mapsto \langle \texttt{a}, [1,2] \rangle$	$\texttt{q} \mapsto \langle \texttt{a} : \texttt{char[8]}, [6,6] \rangle$
5. `*q = 0;`	(not in-bounds)	(in-bounds)

Fig. 3. Mismatched Types Example, assuming $|\texttt{int}| = 4$.

the example in Figure 3: At line 3, `p` is assigned to point to an array of 2 `int`s. However, since the static type of `p` is `char *`, the pointer arithmetic at line 4 is `char`-based, so it must first be translated to `int`-based arithmetic before being applied to the `int`-based range $\langle \texttt{a}, [0,0] \rangle$. Assuming $|\texttt{int}| = 4$ and $|\texttt{char}| = 1$, the `char`-based addition of 6 becomes an `int`-based addition of $6 \cdot \frac{1}{4} = \frac{3}{2}$, which must be approximated as the range $[1,2]$. With the computed fact $\langle \texttt{a}, [1,2] \rangle$, the dereference at line 5 is identified as being potentially out-of-bounds, even though in fact it is in-bounds.

To address these two weaknesses of the Location-Offset domain, we track the type and element count of the pointer's target separately and explicitly. First, we define the domain of Array Descriptors, whose elements describe the identity, element type, and element count of an (array) object:

Array-Descriptor Domain $\mathcal{D} = \langle D, \sqsubseteq_d \rangle$:

- $D = Loc' \times T \times \mathbb{N}$
  - $Loc' = Loc \cup \{\textsc{unknown}\}$, where $\textsc{unknown}$ represents "an unknown location". A flat semi-lattice $\langle Loc', \sqsubseteq_l \rangle$ is defined such that for all $x, y \in Loc$, $\textsc{unknown} \sqsubseteq_l x$, and $x \sqsubseteq_l y$ iff $x = y$.
  - T is the set of unqualified non-void non-array C types, with `typedef`s expanded to their underlying types, and with all pointer types treated as equivalent.
- The descriptor $\langle x, \tau, \sigma \rangle$ represents the location x treated as an array with at least σ elements each of type τ. For readability, we use the notation $x : \tau\texttt{[}\sigma\texttt{]}$ to represent the triple $\langle x, \tau, \sigma \rangle$. Multi-dimensional arrays are flattened; e.g., a 2×3 array of integers `y` is represented as $y : \texttt{int[6]}$.
- $\textsc{unknown} : \tau\texttt{[}\sigma\texttt{]}$ represents a location of unknown identity that is an array of at least σ elements of type τ.
- $(x_1 : \tau_1\texttt{[}\sigma_1\texttt{]}) \sqsubseteq_d (x_2 : \tau_2\texttt{[}\sigma_2\texttt{]})$ iff $x_1 \sqsubseteq_l x_2$ and $\tau_1 = \tau_2$ and $\sigma_1 \leq \sigma_2$.

We now define the Descriptor-Offset Domain:

Descriptor-Offset Domain $\mathcal{DO} = \langle DO, \sqsubseteq_{do} \rangle$:

- $DO = (D \cup \{\textsc{null}\}) \times I$
- The element $\langle x : \tau\texttt{[}\sigma\texttt{]}, [min, max] \rangle$ represents the address of an array x with at least σ elements each of type τ, plus an offset in the range $[min \cdot |\tau|,\ max \cdot |\tau|]$.

- A NULL-targeted element $\langle \text{NULL}, [min, max] \rangle$ represents the integer range $[min, max]$.
- $\langle d_1, o_1 \rangle \sqsubseteq_{do} \langle d_2, o_2 \rangle$ iff $d_1 \sqsubseteq_d d_2$ and $o_1 \sqsubseteq_i o_2$.
(NULL is not $\sqsubseteq_d$-comparable to any member of D).
- $\mathcal{DO}$ can be converted to a lattice $\mathcal{DO}^{\mathrm{L}}$ by adding a "top" element ($\top$) and a "bottom" element ($\bot$).

Notice that $\mathcal{D}$ is partially ordered such that $d_1 \sqsubseteq_d d_2$ only if the size of the array described by d_1 is less than or equal to the size of the array described by d_2. This ensures that $\mathcal{DO}$ satisfies the safe approximation requirement, since if `p` points to an array of 8 elements, it is a safe approximation to say that `p` points to an array of 6 elements.

The rightmost columns of Figures 2 and 3 show the analysis results using Descriptor-Offset ranges. For Figure 2, the meet operation at line 8 sets the location component to UNKNOWN, but the type, count, and offset components are preserved: we are able to approximate the two incoming facts for `p` by taking the smaller type-count descriptor and the superset of the interval components. When dereferencing `p`, if `p` maps to an element $\langle x : \tau\texttt{[}\sigma\texttt{]}, [min, max] \rangle$ such that $min \geq 0$ and $max < \sigma$, then the dereference is guaranteed to be in-bounds, even if $x = \text{UNKNOWN}$ (as is the case for the dereference on line 9 of Figure 2).

For Figure 3, at line 4 we can change the type and count components of the range, so that array `a` is now treated as an array of 8 `char`s. This allows us to recognize that the dereference at line 5 is guaranteed to be in-bounds.

3 Pointer Arithmetic

An important aspect of pointer-range analysis is the handling of pointer arithmetic. The six classes of additive operations in C for integers and pointers are listed in Figure 4, along with their semantics in terms of integer arithmetic (note that *pointer*+*pointer* and *int*−*pointer* are not allowed). Since the two pointer additions $\texttt{+}^{\tau}_{pi}$ and $\texttt{+}^{\tau}_{ip}$ are similar, and the subtractions $\texttt{-}_{ii}$ and $\texttt{-}^{\tau}_{pi}$ can be trivially converted to the corresponding addition with the negative of the second argument, we will only describe the handling of $\texttt{+}_{ii}$, $\texttt{+}^{\tau}_{pi}$, and $\texttt{-}^{\tau}_{pp}$.

Operator : Type	Integer Semantics
$\texttt{+}_{ii}$: $\texttt{int} \times \texttt{int} \to \texttt{int}$	$\texttt{i}_1\texttt{+}_{ii}\texttt{i}_2 \equiv i_1 + i_2$
$\texttt{-}_{ii}$: $\texttt{int} \times \texttt{int} \to \texttt{int}$	$\texttt{i}_1\texttt{-}_{ii}\texttt{i}_2 \equiv i_1 - i_2$
$\texttt{+}^{\tau}_{pi}$: $\tau\texttt{*} \times \texttt{int} \to \tau\texttt{*}$	$\texttt{p+}^{\tau}_{pi}\texttt{i} \equiv p + (i \cdot \lvert\tau\rvert)$
$\texttt{+}^{\tau}_{ip}$: $\texttt{int} \times \tau\texttt{*} \to \tau\texttt{*}$	$\texttt{i+}^{\tau}_{ip}\texttt{p} \equiv (i \cdot \lvert\tau\rvert) + p$
$\texttt{-}^{\tau}_{pi}$: $\tau\texttt{*} \times \texttt{int} \to \tau\texttt{*}$	$\texttt{p-}^{\tau}_{pi}\texttt{i} \equiv p - (i \cdot \lvert\tau\rvert)$
$\texttt{-}^{\tau}_{pp}$: $\tau\texttt{*} \times \tau\texttt{*} \to \texttt{int}^{\dagger}$	$\texttt{p}_1\texttt{-}^{\tau}_{pp}\texttt{p}_2 \equiv (p_1 - p_2)/\lvert\tau\rvert$

†The result of subtracting two pointers actually has an implementation-defined type `ptrdiff_t`.

Fig. 4. C Addition and Subtraction.

3.1 Well-Typed Arithmetic

An arithmetic operation is well typed if the actual types of the arguments match the types expected by the operation. With the descriptor-offset domain, a targeted range $\langle x : \tau\,\texttt{[}\sigma\texttt{]}\,, o\rangle$ represents a value of type τ `*`, while a NULL-targeted range $\langle \text{NULL}, o\rangle$ represents a value of type `int`.

The addition and subtraction of two integer intervals can be safely approximated by the following equations:[1]

- $[min_1, max_1] + [min_2, max_2] = [min_1 + min_2, max_1 + max_2]$
- $[min_1, max_1] - [min_2, max_2] = [min_1 - max_2, max_1 - min_2]$

Well-typed arithmetic on descriptor-offset ranges can be evaluated by applying these equations to the interval components of the ranges:

- Integer addition ($\texttt{+}_{ii}$) of two NULL-targeted ranges:
  $\langle \text{NULL}, [min_1, max_1]\rangle \texttt{+}_{ii} \langle \text{NULL}, [min_2, max_2]\rangle$
  $= \langle \text{NULL}, [min_1 + min_2, max_1 + max_2]\rangle$
- Pointer addition ($\texttt{+}^{\tau}_{pi}$) of a τ-based range and a NULL-targeted range:
  $\langle x : \tau\,\texttt{[}\sigma\texttt{]}\,, [min_1, max_1]\rangle \texttt{+}^{\tau}_{pi} \langle \text{NULL}, [min_2, max_2]\rangle$
  $= \langle x : \tau\,\texttt{[}\sigma\texttt{]}\,, [min_1 + min_2, max_1 + max_2]\rangle$
- Pointer subtraction ($\texttt{-}^{\tau}_{pp}$) of two ranges with the same target location $x \neq$ UNKNOWN and the same element type τ:
  $\langle x : \tau\,\texttt{[}\sigma\texttt{]}\,, [min_1, max_1]\rangle \texttt{-}^{\tau}_{pp} \langle x : \tau\,\texttt{[}\sigma\texttt{]}\,, [min_2, max_2]\rangle$
  $= \langle \text{NULL}, [min_1 - max_2, max_1 - min_2]\rangle$

In C, subtraction of two pointers ($\texttt{-}^{\tau}_{pp}$) is well defined only if the two pointers point to the same array. Therefore, pointer subtraction of two ranges with different or UNKNOWN target locations evaluates to $\perp$.

3.2 Mismatched-Type Arithmetic

An arithmetic operation that is not well typed can arise because C permits casting between pointers to different types, and between integers and pointers; it can also arise from the use of unions. This section addresses the handling of arithmetic operations on ranges with mismatched types. This includes integer addition ($\texttt{+}_{ii}$) with a pointer-typed argument, and pointer addition or subtraction where the type of the operation does not match the argument type.

How this problem is handled depends first on the requirements of the client of the analysis. Specifically, is the client interested only in well-typed accesses? If so, then the result of any pointer arithmetic operation with mismatched types should be $\perp$. However, this is usually too strong a requirement for C programs, because its weak typing discipline means any memory location could be accessed as if it were of any type. With this model of memory locations, we can weaken the definition of the array-descriptor ordering $\sqsubseteq_d$ defined on page 5 so that:

[1] For brevity, we omit details concerning infinite bounds, which are handled by setting respectively the upper/lower bound to plus/minus infinity if either argument needed to compute the bound is infinite.

- $(x_1 : \tau_1\texttt{[}\sigma_1\texttt{]}) \sqsubseteq_d (x_2 : \tau_2\texttt{[}\sigma_2\texttt{]})$ iff $x_1 \sqsubseteq_l x_2$ and $|\tau_1\texttt{[}\sigma_1\texttt{]}| \leq |\tau_2\texttt{[}\sigma_2\texttt{]}|$.

That is, d_1 is a safe approximation of d_2 if the array described by d_1 is smaller than the array described by d_2, regardless of the element types of the descriptors.

This means that if the size of each type is known at analysis time, we can convert a range's type from τ_a to τ_b as follows:

$$\langle x : \tau_a\texttt{[}\sigma\texttt{]}, [min, max]\rangle \Longrightarrow \langle x : \tau_b\texttt{[}\left\lfloor \sigma \cdot \tfrac{|\tau_a|}{|\tau_b|}\right\rfloor\texttt{]}, \left[\left\lfloor min \cdot \tfrac{|\tau_a|}{|\tau_b|}\right\rfloor, \left\lceil max \cdot \tfrac{|\tau_a|}{|\tau_b|}\right\rceil\right]\rangle \quad (1)$$

We can also transform the base type of a pointer addition by adjusting the right-hand-side interval. A τ_b-based pointer addition ($\texttt{+}_{pi}^{\tau_b}$), where the right-hand-side is NULL-targeted, can be converted to a τ_a-based addition as follows:

$$r_1 \texttt{+}_{pi}^{\tau_b} \langle \text{NULL}, [min_2, max_2]\rangle \Longrightarrow r_1 \texttt{+}_{pi}^{\tau_a} \langle \text{NULL}, \left[\left\lfloor min_2 \cdot \tfrac{|\tau_b|}{|\tau_a|}\right\rfloor, \left\lceil max_2 \cdot \tfrac{|\tau_b|}{|\tau_a|}\right\rceil\right]\rangle \quad (2)$$

Transformation (1) or (2) can be used to eliminate any type mismatch, to get a well-typed operation that can be evaluated by the equations in Section 3.1.

Revisiting the Figure 3 example, the addition `p + 6` at line 4 has a type mismatch, because `p` maps to an `int`-based range, while the addition is `char`-based. We can apply either transformation (1) or (2), to get the following results:

$$\begin{aligned} &\langle a : \texttt{int[2]}, [0,0]\rangle \texttt{+}_{pi}^{\texttt{char}} \langle \text{NULL}, [6,6]\rangle \\ &\quad = (1) \Rightarrow \langle a : \texttt{char[8]}, [0,0]\rangle \texttt{+}_{pi}^{\texttt{char}} \langle \text{NULL}, [6,6]\rangle = \langle a : \texttt{char[8]}, [6,6]\rangle \\ &\quad = (2) \Rightarrow \langle a : \texttt{int[2]}, [0,0]\rangle \texttt{+}_{pi}^{\texttt{int}} \langle \text{NULL}, [1,2]\rangle \quad = \langle a : \texttt{int[2]}, [1,2]\rangle \end{aligned}$$

Because of the floor and ceiling operations, there may be some loss in precision as a result of applying either transformation (1) or (2). It is therefore important to choose a transformation that minimizes loss of precision. In practice, the size of one of the types τ_a, τ_b is usually a multiple of the size of the other (making either $\frac{|\tau_a|}{|\tau_b|}$ or $\frac{|\tau_b|}{|\tau_a|}$ a round number), so that at least one of the transformations will result in no loss of precision.

Transformations (1) and (2) can only be applied if the sizes of types are known at analysis time. If an analysis is designed to be portable across all platforms, then specific sizes of types cannot be assumed. In such a case, we can still make some safe approximations to get results that are more precise than $\perp$, by making use of portable information about the sizes of types as defined or implied in the C specifications:

1. $|\texttt{char}| = 1$
2. $|\texttt{char}| \leq |\tau|$ for any non-void C type τ.
3. $|\texttt{char}| \leq |\texttt{short}| \leq |\texttt{int}| \leq |\texttt{long}| \leq |\texttt{long long}|$
4. $|\texttt{float}| \leq |\texttt{double}| \leq |\texttt{long double}|$
5. $|\tau\texttt{[}\sigma\texttt{]}| = |\tau| \cdot \sigma$
6. $|union\ \{\tau_1 \ldots \tau_n\}| \geq \max_{i=1\ldots n}(|\tau_i|)$
7. $|struct\ \{\tau_1 \ldots \tau_n\}| \geq \sum_{i=1}^{n} |\tau_i|$
8. $|struct\ \{\tau_1 \ldots \tau_n\}| \leq |struct\ \{\tau_1 \ldots \tau_n \ldots\}|$
9. $|\tau_1\texttt{*}| = |\tau_2\texttt{*}|$ for any C types τ_1, τ_2.

Item 1 implies that `char`-pointer arithmetic is equivalent to integer arithmetic ($+_{pi}^{\texttt{char}} \equiv +_{ii}, -_{pp}^{\texttt{char}} \equiv -_{ii}$). Item 6 states that a union type is at least as large as its largest member, while item 7 states that a struct type is at least as large as the sum of its constituents' sizes (it may be larger due to padding). Item 8 takes advantage of a subtype relationship between two structures that share a common initial sequence. Item 9, which states that all pointers are of the same size, is strictly speaking an unsafe assumption, but it is all but implied by the requirements that all pointers can be cast to `void *` without loss of information, and that the return value of `malloc` can be safely cast to any pointer type. We therefore assume it to be true.

The first safe approximation, which arises often because of the way we normalize multi-dimensional arrays, is to convert a $\tau\texttt{[}\sigma\texttt{]}$-based pointer addition, where $\tau\texttt{[}\sigma\texttt{]}$ is an array type, to a τ-based pointer addition. This is done by applying transformation (2) with the knowledge that $\frac{|\tau\texttt{[}\sigma\texttt{]}|}{|\tau|} = \sigma$:

$$r_1 +_{pi}^{\tau\texttt{[}\sigma\texttt{]}} \langle \text{NULL}, [min_2, max_2]\rangle \Longrightarrow r_1 +_{pi}^{\tau} \langle \text{NULL}, [min_2 \cdot \sigma, max_2 \cdot \sigma]\rangle$$

Next, if we only know the relative sizes of two types, we can make the following approximations for transformation (2).

$$\text{If } |\tau_b| \leq |\tau_a|,\ r_1 +_{pi}^{\tau_b} \langle \text{NULL}, [min_2, max_2]\rangle \Longrightarrow$$
$$r_1 +_{pi}^{\tau_a} \langle \text{NULL}, \left[\begin{pmatrix} min_2 & \text{if } min_2 \leq 0 \\ 0 & \text{otherwise} \end{pmatrix}, \begin{pmatrix} max_2 & \text{if } max_2 \geq 0 \\ 0 & \text{otherwise} \end{pmatrix}\right]\rangle \quad \text{(2a)}$$
$$\text{If } |\tau_a| \leq |\tau_b|,\ r_1 +_{pi}^{\tau_b} \langle \text{NULL}, [min_2, max_2]\rangle \Longrightarrow$$
$$r_1 +_{pi}^{\tau_a} \langle \text{NULL}, \left[\begin{pmatrix} min_2 & \text{if } min_2 \geq 0 \\ -\infty & \text{otherwise} \end{pmatrix}, \begin{pmatrix} max_2 & \text{if } max_2 \leq 0 \\ +\infty & \text{otherwise} \end{pmatrix}\right]\rangle \quad \text{(2b)}$$

For the pointer addition `p + 6` at line 4 of Figure 3, since we know $|\texttt{char}| \leq |\texttt{int}|$, we can apply transformation (2a) to get:

$$\begin{aligned} \langle a : \texttt{int[2]}, [0,0]\rangle +_{pi}^{\texttt{char}} \langle \text{NULL}, [6,6]\rangle &\Longrightarrow \langle a : \texttt{int[2]}, [0,0]\rangle +_{pi}^{\texttt{int}} \langle \text{NULL}, [0,6]\rangle \\ &= \langle a : \texttt{int[2]}, [0,6]\rangle \end{aligned}$$

Note that the resulting range is a safe approximation (superset) of the more precise range $\langle a : \texttt{int[2]}, [1,2]\rangle$ obtained earlier with exact size information.

A similar approximation can be made for transformation (1), but only in one direction:

$$\text{If } |\tau_b| \leq |\tau_a|, \text{ let } n \text{ be such that } 1 \leq n \leq \frac{|\tau_a|}{|\tau_b|},$$
$$\text{then } \langle x : \tau_a\texttt{[}\sigma\texttt{]}, [min, max]\rangle \Longrightarrow$$
$$\langle x : \tau_b\texttt{[}n \cdot \sigma\texttt{]}, \left[\begin{pmatrix} min & \text{if } min \geq 0 \\ -\infty & \text{otherwise} \end{pmatrix}, \begin{pmatrix} max & \text{if } max \leq 0 \\ +\infty & \text{otherwise} \end{pmatrix}\right]\rangle \qquad \text{(1a)}$$

A key here is that $|\tau_a\texttt{[}\sigma\texttt{]}| \geq |\tau_b\texttt{[}n \cdot \sigma\texttt{]}|$, which ensures that the right-hand-side of the transformation is a safe approximation of the left-hand-side. If τ_1 and τ_2 are scalar types, the exact ratio $\frac{|\tau_a|}{|\tau_b|}$ is not portably defined, so the only safe value for n is 1. But if τ_a is an aggregate type, a safe n can be obtained by counting the number of elements in τ_a that are at least as big as τ_b. For example, $\frac{|struct\{int\,\texttt{[2]},long,char\}|}{|int|} \geq 3$. It is then safe to multiply the σ component of the resultant range by n.

Thus, when evaluating the pointer addition

$$\langle x : \tau_a \texttt{[}\sigma\texttt{]}, o_1\rangle \texttt{+}^{\tau_b}_{pi} \langle \text{NULL}, o_2\rangle$$

if $|\tau_a| \leq |\tau_b|$, only transformation (2b) can be applied. But if $|\tau_b| \leq |\tau_a|$, there is a choice between (1a) and (2a). As was the case for transformations (1) and (2), it is important to choose the transformation that minimizes the loss of precision. In general, transformation (1a) is more precise if the left-hand-side offset o_1 is $[0, 0]$; otherwise (2a) is more precise.

4 Experimental Results

The pointer-range analysis was implemented as a context-insensitive interprocedural dataflow analysis (operating on a supergraph of the program). Since the interval lattice has infinite descending chains, widening [5] is used to ensure convergence, and narrowing is used to obtain more precise results. A points-to analysis [8] pass is first performed to safely account for aliasing, and also to identify targets of indirect procedure calls.

The following numbers were collected to gauge the potential utility of this analysis for various applications.

Bounded and half-open ranges: We count the number of dereferences `*p` for which p maps to a range with a known location and is either

- *bounded*: the offset component is finite, or
- *half-open*: the offset component has at least one finite bound, e.g., $[1, +\infty]$ or $[-\infty, 3]$.

Such ranges are potentially useful for dependence analysis, where one is interested in whether two dereferences may access the same memory location.

In-bounds dereferences: At each dereference `*p`, if $p \mapsto \langle x : \tau \texttt{[}\sigma\texttt{]}, [min, max]\rangle$ such that $min \geq 0$ and $max < \sigma$, then the dereference is guaranteed to be in-bounds. This information can be used to eliminate unnecessary bounds checks, and to detect potential out-of-bounds errors.

Figure 5 presents the results of our analysis on benchmarks from Cyclone[15], olden[4], Spec 95 and Spec 2000. Column (a) gives the number of lines of code and column (b) gives the static number of dereferences in each program.

Using the descriptor-offset (*DO*) representation, column (c) gives the percentage of dereferences that had bounded ranges and (d) gives the percentage that had half-bounded ranges. These may be contrasted roughly with the results of numeric range analysis given in [24], which identified about 30% bounded and 40% half-bounded ranges for non-pointer variables in some small benchmarks (100-400 statements).

		num	*DO* %			arrays only %			in-b diff.		kn.
	LOC	derefs	bnd	half-b	in-bnd	bnd	half-b	in-bnd	multi	pred	preds
	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)
Cyclone											
aes	1,822	152	49.3	77.0	46.1	8.6	23.7	5.9	0	7	3
cacm	340	25	48.0	52.0	48.0	40.0	44.0	40.0	0	8	0
cfrac	4,218	446	2.7	4.7	3.6	0.0	0.0	0.0	6	0	1
finger	158	24	4.2	4.2	33.3	0.0	0.0	0.0	7	0	1
grobner	4,737	1,349	26.6	27.9	34.9	0.7	1.7	0.5	117	5	6
matxmult	1,377	13	30.8	30.8	30.8	0.0	0.0	0.0	0	4	0
ppm	1,421	123	5.7	7.3	5.7	0.8	1.6	0.8	0	3	0
tile	4,880	324	6.5	10.2	5.6	2.5	3.1	1.5	0	5	0
olden											
bh	3,200	219	42.5	42.5	42.5	42.5	42.5	42.5	0	60	1
bisort	690	89	10.1	10.1	10.1	0.0	0.0	0.0	0	0	0
em3d	538	51	15.7	15.7	0.0	0.0	0.0	0.0	0	0	0
health	706	84	19.0	19.0	19.0	0.0	0.0	0.0	0	1	0
mst	610	52	11.5	11.5	11.5	0.0	0.0	0.0	0	0	0
perimeter	472	44	31.8	31.8	31.8	0.0	0.0	0.0	0	0	0
power	867	190	30.5	30.5	33.2	7.4	7.4	7.4	5	22	0
treeadd	375	8	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0
tsp	684	94	11.7	11.7	11.7	0.0	0.0	0.0	0	0	0
Spec95											
compress	3,900	83	19.3	32.5	18.1	12.0	14.5	10.8	0	6	0
gcc	205,106	52,108	7.9	10.6	8.6	3.0	5.2	3.0	378	412	58
go	29,629	8,893	7.1	13.0	7.5	7.0	12.7	6.9	40	369	4
ijpeg	31,215	8,718	13.9	16.2	14.5	3.0	3.3	3.0	52	36	3
li	7,630	914	1.2	2.8	5.0	0.5	0.7	0.5	35	3	1
m88ksim	19,227	2,406	25.4	28.8	38.9	21.4	23.2	21.4	326	115	4
perl	26,872	9,096	2.3	2.9	3.2	0.8	1.0	0.8	88	0	5
vortex	67,219	12,017	20.8	21.0	28.0	1.1	1.2	1.0	872	1	175
Spec2000											
ammp	13,483	5,040	9.8	14.3	10.2	7.6	9.2	7.6	22	57	4
art	1,270	258	11.6	12.8	11.6	11.6	12.8	11.6	0	8	0
bzip2	4,650	522	22.6	37.9	19.7	15.5	28.5	12.6	0	43	12
crafty	20,545	3,764	49.6	56.2	48.4	41.7	45.9	39.5	37	631	10
equake	1,513	777	31.3	34.4	32.3	12.5	14.9	12.5	8	68	0
gap	71,363	26,459	3.5	4.9	3.6	3.4	3.7	3.4	12	166	26
gzip	8,605	720	25.3	33.3	27.8	9.6	16.1	9.6	18	58	14
mcf	2,412	568	28.2	30.3	28.2	0.7	2.8	0.7	0	2	0
mesa	58,724	23,741	5.1	8.0	5.8	3.4	5.8	3.4	178	53	29
parser	11,391	2,867	3.6	9.2	3.6	1.5	6.1	1.4	0	9	3
twolf	20,461	9,573	1.0	1.5	0.9	0.8	1.3	0.8	2	9	6
vpr	17,730	2,874	14.3	14.3	16.7	3.2	3.2	3.1	74	20	1
Total		174,685							2277	2181	367
Average			17.6	20.9	18.9	7.1	9.1	6.8	61.5	58.9	9.9

Fig. 5. Results

Column (e) gives the percentage of dereferences found to be in-bounds. While the average percentage is quite low, there are many cases, including some larger programs, for which over 30% of dereferences were found to be in-bounds.

To contrast these numbers against how well Array-Range Analysis would fare, columns (f)-(h) give the percentages of bounded, half-bounded, and in-bounds dereferences that were *direct* array accesses, i.e., accesses of the form `a[`x`]` where `a` is an array object. These represent the results that could be obtained using an Array-Range Analysis approach that does not handle pointers (e.g., [5, 24]). The difference is large for all three categories, confirming that handling of pointers is important when analyzing C programs.

To motivate the use of the *DO* representation rather than the simpler location-offset (*LO*) representation, we evaluated the two ways in which *DO* can give better results than *LO*:

- *multi-target: DO* can represent a pointer to multiple targets, as in the Figure 2 example.
- *transformation (1): DO* allows the application of Transformation (1) or (1a) when handling mismatched-type operations.

We found that *multi-target* made a bigger difference: column (i) gives the number of in-bounds dereferences that were not found when the *multi-target* ability was disabled – on average about 11% of the in-bounds dereferences per benchmark. Most of these come from procedure calls, where different arrays of the same size are passed as an argument to a procedure that accesses the array. As for *transformation (1)*, only 35 in-bounds dereferences were not found when this feature was disabled (one in `gcc`, 21 in `m88ksim`, and 13 in `crafty`). Overall, the difference between the *DO* and *LO* is significant, and shows that the type-count descriptor is an effective mechanism for handling challenging aspects of C.

To measure the price of portability, we looked at the improvement in results if exact sizes of types are assumed, i.e., if type mismatches are handled with transformations (1) and (2) rather than (1a), (2a), and (2b). Only five more in-bounds dereferences were found using exact sizes (two in `gcc` and three in `gap`), suggesting that in practice, the portable transformations produce results that are almost as good as the non-portable ones.

One aspect of range analysis that was not described in this paper is the treatment of ranges at branch nodes. For example, consider a branch node containing the predicate $v_1 \texttt{<} v_2$. If the before- dataflow fact mappings are:

$$\begin{array}{c} v_1 \mapsto \langle x : \tau\texttt{[}\sigma\texttt{]}, [min_1, max_1]\rangle \\ v_2 \mapsto \langle x : \tau\texttt{[}\sigma\texttt{]}, [min_2, max_2]\rangle \end{array}$$

then the after- fact mappings along the true branch will be:

$$\begin{array}{c} v_1 \mapsto \langle x : \tau\texttt{[}\sigma\texttt{]}, [min_1, \min(max_1, max_2 - 1)]\rangle \\ v_2 \mapsto \langle x : \tau\texttt{[}\sigma\texttt{]}, [\max(min_1 + 1, min_2), max_2]\rangle \end{array}$$

This is an important improvement to make for precision, as confirmed by Column (j), which gives the number of in-bounds dereferences that were missed when the

	LOC	time(s)		LOC	time(s)		LOC	time(s)
li	7,630	15	twolf	20,461	12	mesa	58,724	29371
parser	11,391	5	crafty	20,545	16	vortex	67,219	233
ammp	13,483	6	perl	26,872	39	gap	71,363	656
vpr	17,730	7	go	29,629	7	gcc	205,106	2640
m88ksim	19,227	21	ijpeg	31,215	487			

Fig. 6. Analysis Times

range improvements at branch nodes were not applied – on average about 20% of the in-bounds dereferences per benchmark.

Precise treatment of ranges at branch nodes also lets us discover infeasible branches. For example, at the predicate v_1 `<` v_2, if v_1's range is entirely less than v_2's range, then the value of the predicate is statically known, indicating that the false branch is infeasible. Column (k) gives the number of known predicates found in the programs. The large number of known predicates in `vortex` comes from a programming style where a series of procedure calls are each checked for success by `if` statements, even though some of the procedures always return the same value.

Finally, as a rough indicator of the efficiency of the algorithm, Figure 6 gives the analysis times (wallclock time, in seconds) on a 1GHz Pentium II with 500MB RAM, running Linux, listed in order of increasing size (by lines of code). The benchmarks not listed each took less than a second to analyze.

4.1 Improvements

The current implementation includes several weaknesses that can be addressed with known solutions. Among the possible improvements are adding flow-sensitivity or context-sensitivity to the points-to analysis [16, 10, 27], and adding context sensitivity to the dataflow analysis [18], but these improvements will increase the time complexity of the analysis.

Another aspect that could be improved is the handling of heap-allocated objects. Currently, only `malloc` calls for which the argument is a constant C or an expression C `* sizeof(`τ`)` are mapped to a `malloc` location with a non-`void` type and non-zero count. Such cases account for 46% of the `malloc` calls in the programs, so there is room for improvement. Many programs use a `malloc` wrapper to check for error conditions; this common practice becomes a problem for static analysis because it causes multiple conceptual allocation sites to be folded into a single `malloc` callsite. Limited use of inlining and constant propagation can be used to split the `malloc` callsite into multiple callsites, to increase the likelihood of having a MALLOC location with a meaningful type and count.

4.2 Extensions

The range analysis described in this paper only computes ranges with constant bounds. It relies on the presence of constants in the source code to derive meaningful ranges, and does not record information about the relationships between variables. Approaches that track symbolic ranges [2, 21] and constraints between variables [6, 7, 20, 3, 23] can significantly improve results in applications that are interested in bounds checking or discovering non-aliasing memory accesses. Ideas discussed in this paper could be applied to extend previous approaches to handle pointers in general.

String manipulation is another aspect of C worthy of special consideration. A string is conceptually a separate data type, with its own library to manipulate values, but its implementation on top of arrays makes it susceptible to out-of-bounds array accesses. Tracking the string length as a separate attribute from the array size, and deriving information based on the semantics of C library functions, can lead to more precise results when trying to discover potentially out-of-bounds dereferences [25, 9], which is an important concern for program security.

5 Related Work

Range analysis has been around for decades, and was the motivating example used in the seminal paper on abstract interpretation [5], which introduced the notions of widening and narrowing. Other early work on range analysis relied on the presence of structured loops to infer loop bounds information [13, 26]. Verbrugge *et al* describe range analysis as "generalized constant propagation" [24], and use it for dead-code elimination and array dependence testing in the McCAT optimizing/parallelizing compiler. Stephenson *et al* [22] use range analysis to compute the number of bits needed to store a given value in hardware.

Patterson [19] uses range analysis for static branch prediction: each variable at each program point is mapped to a set of probability-weighted ranges. The weights are used at branch predicates to predict the likelihood of branching in a given direction, and is used for various code-generation optimizations. Gu *et al* [11] use range analysis to discover opportunities for array privatization and parallelization in loops, while Gupta *et al* [12] do the same for recursive divide-and-conquer procedures. They both use a Guarded Array Region representation that associates a predicate with each range. Balakrishnan and Reps [1] use range analysis to infer high-level information from binary code: with a range representation of the form $a \times [b, c] + d$, they compute value sets that are conceptually equivalent to the high-level notion of a variable, to enable high-level analyses like reaching definitions to be applied to binary code.

These four approaches all include the notion of a "stride" in their representation to capture the common access pattern of arrays. Wilson *et al* [27] also use a stride to improve their pointer analysis. Conceptually, the τ component in our descriptor-offset representation encodes the stride in a portable format, allowing our analysis to be used in settings where exact sizes of types cannot be assumed.

Numerous approaches compute symbolic range information, to allow tracking of constraints between variables, but few have dealt with pointers. Rugina and Rinard [21] compute symbolic ranges for variables including pointers, and use linear programming to identify non-intersecting ranges that could be used for automatic parallelization or identifying in-bounds accesses. Approaches that deal with C strings to identify potential buffer overruns [25, 9, 17] must necessarily handle pointers, but only `char` pointers; thus they do not need to address problems related to casting.

6 Conclusion

We have presented a pointer-range analysis that extends traditional array-range analysis to handle pointers as well as non-trivial aspects of C, including pointer arithmetic and type-casting. We described two possible range representations: the intuitive location-offset representation, and the descriptor-offset representation, and showed that the latter yields better results in practice. The ideas we have presented can provide useful insight into extending existing array-based range analysis to handle pointers in C-like languages.

References

1. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *International Conference on Compiler Construction*, Barcelona, Spain, Mar. 2004.
2. W. Blume and R. Eigenmann. Demand-driven, symbolic range propagation. In *8th International workshop on Languages and Compilers for Parallel Computing*, pages 141–160, Columbus OH, Aug. 1995.
3. R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, Vancouver, BC, June 2000.
4. M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. Technical Report TR-483-95, Princeton University, 1995.
5. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *ACM Symposium on Principles of Programming Languages*, pages 106–130, Apr. 1976.
6. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Symposium on Principles of Programming Languages*, pages 84–96, Jan. 1978.
7. B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, Dec. 1996.
8. M. Das. Unification-based pointer analysis with directional assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, Vancouver, BC, June 2000.
9. N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *The 8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, page 194. Springer, July 2001.
10. M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, FL, June 1994.

11. J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *ACM/IEEE Conference on Supercomputing*, San Diego, CA, Dec. 1995.
12. M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 139–148, Newport Beach, CA, Oct. 1999. IEEE Computer Society.
13. W. H. Harrison. Compiler analysis of the value ranges for variables. In *IEEE Transactions on Software Engineering*, volume SE-3, pages 243–250, May 1977.
14. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions of Parallel and Distributed Computing*, 2(3):350–360, July 1991.
15. T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.
16. W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, CA, June 1992.
17. D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington, D.C., Aug. 2001.
18. F. Martin. Experimental comparison of *call string* and *functional* approaches to interprocedural analysis. In *6th Int. Conf. on Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 63–75. Springer, Mar. 1999.
19. J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 67–78, La Jolla, CA, June 1995.
20. W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems*, 20(3):635–678, May 1998.
21. R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, Vancouver, BC, June 2000.
22. M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 108–120, Vancouver, BC, June 2000.
23. Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 280–295, Mar. 2004.
24. C. Verbrugge, P. Co, and L. Hendren. Generalized constant propagation: A study in C. In *6th Int. Conf. on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 74–90. Springer, Apr. 1996.
25. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symposium on Network and Distributed Systems Security*, pages 3–17, San Diego, CA, Feb. 2000.
26. J. Welsh. Economic range checks in Pascal. *Software–Practice and Experience*, 8:85–97, 1978.
27. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, CA, June 1995.