Validation of Memory Accesses Through Symbolic Analyses

Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Oliveira, Fernando Magno Quintão Pereira, Laure Gonnord

Accepted in OOPSLA'14 – Author Version

SPLASH

Validation of Memory Accesses Through Symbolic Analyses

Henrique Nazaré Izabela Maffra
Willer Santos Leonardo B. Oliveira
Fernando Magno Quintão Pereira
Universidade Federal de Minas Gerais, Brazil
{hnsantos,izabela,willer,leob.fernando}@dcc.ufmg.br

Laure Gonnord
Université Lyon1, France & LIP
(UMR CNRS / ENS Lyon / UCB Lyon 1 / INRIA)
Laure.Gonnord@ens-lyon.fr

Abstract

The C programming language does not prevent out-of-bounds memory accesses. There exist several techniques to secure C programs; however, these methods tend to slow down these programs substantially, because they populate the binary code with runtime checks. To deal with this problem, we have designed and tested two static analyses - symbolic region and range analysis - which we combine to remove the majority of these guards. In addition to the analyses themselves, we bring two other contributions. First, we describe live range splitting strategies that improve the efficiency and the precision of our analyses. Secondly, we show how to deal with integer overflows, a phenomenon that can compromise the correctness of static algorithms that validate memory accesses. We validate our claims by incorporating our findings into AddressSanitizer. We generate SPEC CINT 2006 code that is 17% faster and 9% more energy efficient than the code produced originally by this tool. Furthermore, our approach is 50% more effective than Pentagons, a state-of-the-art analysis to sanitize memory accesses.

Categories and Subject Descriptors D - Software [*D.3 Programming Languages*]: D.3.4 Processors - Compilers

General Terms Languages, Security, Experimentation

Keywords Security, static analysis, buffer overflow

1. Introduction

C is one of the most popular languages among programmers. It has been used in the development of operating systems, browsers, servers, and a plethora of other essential applications. In spite of its popularity, the development of robust software in C is difficult, due to the weak type system present in this language. The language's semantics does not prevent, for instance, out-of-bounds memory accesses. Much work has been done to mitigate this problem. Today we have tools like SAFECode [13] and AddressSanitizer [29] that extend the C compiler to generate memory safe assembly code. The main drawback of these tools is the overhead that they impose on compiled programs. In this paper, we present a suite of static analyses that removes part of this overhead.

An array access in C or C++, such as $a[i]$, is safe if variable i is greater than, or equal zero, and its value is less than the maximum addressable offset starting from base pointer a. A bound check is a dynamic test that ensures that a particular array access is safe. This definition of a safe array access, although informal, makes it clear that the elimination of bound checks is a problem that involves the comparisons between ranges of variables. There exists a number of instances of Cousot and Cousot's abstract interpretation framework [9] that perform such comparisons. One of the most successful analysis in this domain is due to Logozzo and Fähndrich [19]: the so called *Pentagon Analysis*. The success of this static analysis is, in part, due to its efficiency. Pentagons are much cheaper than previous variations of abstract interpretation which are also able to determine a "less-than" relation between variables, such as octagons [20] or the more general polyhedra [10]. However, in this paper we show that it is still possible to enhance the precision of Pentagons, without increasing its asymptotic complexity.

We have designed, implemented and successfully tested a *Symbolic Range Analysis* that gives us more information than Pentagons, at a small cost in speed. Our static analysis is built on top of a lattice of symbolic constraints described by Blume and Eigenmann in 1994 [5]. As a second contribution, we describe *Forward Symbolic Region Analysis* [1], a form of abstract interpretation that associates pointers to a conservative approximation of their maximum addressable offsets. This problem has been first discussed by Rugina and Rinard [27] in 2005. Nevertheless, we approach region analysis through a completely different algorithm, whose differences we emphasize in Section 6.

[1] We use the term *forward* to distinguish our analysis from Rugina's [27], which is backward. Henceforth we shall drop the word *forward*.

Contrary to many related works, we are aware of the danger that integer overflows pose to the soundness of our analyses. We deal with this problem using the instrumentation recently proposed by Dietz *et al.* [14] to guard arithmetic operations against integer overflows. However, instead of instrumenting the entire program, we restrict ourselves to the program slice that is related to memory allocation or access. We discover this slice via a linear time backward analysis that tracks data and control dependences along the program's intermediate representation. We only instrument operations that are part of this slice. We pay a fee of less than 2.5% of performance overhead to ensure the safety of all our analyses. However, this safety lets us perform much more aggressive symbolic comparisons, outperforming substantially an analysis that is oblivious to integer overflows.

Section 5 contains an extensive evaluation of our ideas. We have tested them in AddressSanitizer [29], an industrial-quality tool built on top of the LLVM compiler [18]. This tool produces instrumented binaries out of C source code, to either log or prevent any out-of-bounds memory access. AddressSanitizer has a fairly large community of users, having been employed to instrument browsers such as Firefox and Chromium. This instrumentation has a cost: in general it slows down computationally intensive programs by approximately 70%, and increases their energy consumption by two times. We can remove about half this overhead, keeping all the guarantees that AddressSanitizer provides. Our results are 45% better than those obtained via Pentagons. We measure this effectiveness in terms of speed and energy consumption. For the latter, we resort to the methodology developed by Singh *et al.* [31], which measures current in embedded boards using an actual power meter. We summarize our main contributions as follows:

- We provide two new abstract-interpretation based static analyses for computing ranges of variables (Section 4.3) and offset of pointers (Section 4.4) that are *parametrized* by the input/unknown values of the program.
- We ensured the correctness of the analyses with respect to *variable overflows* (Section 4.2).
- We enhanced the precision of the analyses by performing *semantic preserving* transformations on the program input (range splitting - Section 4.1).

2. The Static Analysis Zoo

In this section, we discuss two different ways to classify static analyses, so that we can better explain how our work stands among the myriad of ideas that exist in this field.

Relational and Semi-Relational Analyses. The result of a static analysis is a function $\mathcal{F} : \mathcal{S} \mapsto \mathcal{I}$, which maps a universe of *syntactic entities*, $\mathcal{S}$ to elements in a set of *facts* $\mathcal{I}$. These syntactic entities can be any category of constructs present in the syntax of a program's code, such as labels, regions, variables, etc. Facts are elements in a algebraic body called a *semi-lattice*. A semi-lattice is formed by a set, augmented with a partial order between its elements, plus the additional property that every two elements in this set have a least upper bound [23, Apx.A].

If $\mathcal{S}$ is the power set of the variables in a program, than we say that the static analysis is *relational*. Examples of relational analyses include the polyhedra of Cousot and Halbwachs [10], and the octagons of Miné [20] where we can infer properties such that $s - t \leq 1$. If $\mathcal{S}$ is just the set of program variables, but $\mathcal{I}$ can contain relations between program variables, then the analysis is called *semi-relational*. Examples of semi-relational analyses include the "less-than" inference rules used by Logozzo and Fändrich [19] or by Bodik *et al.* [6]. Pentagons are a semi-relational lattice, that associates each program variable v to a pair (L, I). I is v's range on the interval lattice. L is a set of variables proven to be less than v. Finally, if $\mathcal{S}$ is the set of variables, but $\mathcal{I}$ does not refer to other program variables, then the analysis is called *non-relational*. The vast majority of the static analyses used in compilers, from constant propagation to classic range analysis [9], are non-relational.

Example 1 *Figure 1 shows examples of these three types of analyses, including the Symbolic Range Analysis that we describe in Section 4.3. The information associated with a variable depends on which part of the program we are; hence, the figure shows the results of each analysis at three different regions of the code. In this example, classic range analysis can only infer positiveness of variables. Pentagons can infer also that j is always strictly less than N inside the loop. Octagons are more precise since they are able to find that $m = i$ at control points b and c, for instance* [2].

Relational analyses tend to be more precise than their semi-relational and non-relational counterparts. In our context, *precision* is measured by the amount of the information that the function $\mathcal{F}$ can encode. In a relational analysis, this function operates on a much larger set $\mathcal{S}$ than in a semi-relational approach; thus, the difference in precision. On the other hand, semi-relational analyses are likely to be more efficient than relational algorithms, exactly because they deal with a smaller set $\mathcal{S}$. To illustrate this gap, Oh *et al.* [24] have compared the scalability of octagons, one of the most efficient relational domains, against intervals, the domain used by traditional range analysis, as defined by Cousot and Cousot [9]. In their experiments, range analysis, a non-relational analysis, was two orders of magnitude faster than octagons. This difference increases with the size of the programs that must be analyzed. Our symbolic range analysis, which is semi-relational, is as fast as a state-of-the-art implementation of range analysis due to Rodrigues *et al.* [26].

Sparse and Dense Analyses. If the set $\mathcal{S}$ contains only variables, then we say that the static analysis that generates it is *sparse*; otherwise, we say that the analysis is *dense*.

[2] These results can be reproduced at `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`

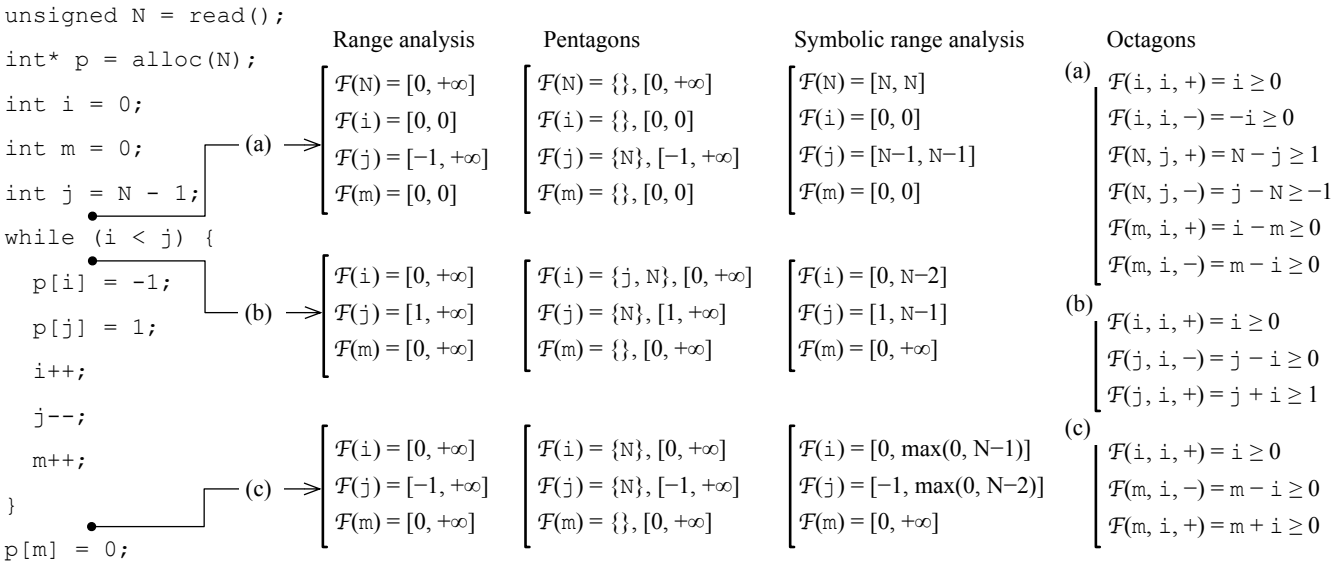


Figure 1. A comparison between four different types of static analyses. Only a few relations are shown for Octagons.

Usually, in a dense analysis, the set $\mathcal{S}$ contains relations between variables and program regions. In other words, the facts associated with a variable depend on which part of the program we consider. All the analyses in Figure 1 are dense. For instance, the range of variable `i` is $[0, 0]$ at program point (a) and $[0, \infty]$ at (b).

In the early nineties, Choi *et al.* [8] have shown that sparse implementations of static analyses tend to outperform dense versions of them, in terms of both, space and time. This observation has been further corroborated by many different works, and more recently, by an investigation due to Oh *et al.* [24]. Thus, in order to capitalize on two decades of advances in the field of compiler theory, all the formalizations that we show in this paper are made on top of a sparse analysis framework.

3. The Program Model

All the analyses that we present in this paper run on programs in *Static Single Assignment* (SSA) form [12]. To formalize our analyses, we define them over a core language, which emulates the imperative features of C that interests us. This section gives the syntax and semantics of this programming model. We conclude our formalization with Definition 3.1, which clarifies the meaning of a *safe* program.

3.1 A Core Language

We define a core language, whose syntax is given in Figure 2, to explain our analyses. The constructions are of three types: variable manipulation (assignments and computation of expressions), memory accesses (allocation, storage at a given address, ...), and control flow (tests, branch, ...). The notation $\overline{v}$, used in the description of ϕ-functions, typical in the SSA representation, represent a vector of variables.

Programs (P)	::=	$\ell_1 : I_1, \ell_2 : I_2, \ldots, \ell_n : \texttt{end}$
Labels (L)	::=	$\{\ell_1, \ell_2, \ldots\}$
Variables (V)	::=	$\{v_1, v_2, \ldots\}$
Constants (C)	::=	$\{c_1, c_2, \ldots\}$
Operands (O)	::=	$V \cup C$
Instructions (I)	::=	
– Assignment	|	$v = o$
– Input	|	$v = \bullet$
– Binary operation	|	$v_1 = v_2 \oplus v_3$
– ϕ-function	|	$\overline{v} = \phi(\overline{v_1}, \ldots, \overline{v_n})$
– Store into memory	|	$*v_1 = v_3$
– Load from memory	|	$v_1 = *v_2$
– Allocate memory	|	$v_1 = \texttt{alloc}(v_2)$
– Liberate memory	|	$\texttt{free}(v)$
– Branch if zero	|	$\texttt{br}(v, \ell)$
– Unconditional jump	|	$\texttt{jmp}(\ell)$
– Halt execution	|	`end`

Figure 2. The syntax of our core language.

Formal semantics The (small step) semantics of our core language is defined by the interpreter shown in Figures 3, 4 and 5. We have validated this interpreter with a Prolog implementation, which is available in our repository. Figure 3 contains the definition of data and arithmetic operations. We use the relation $\xrightarrow{i}$ to describe the computation performed by an arithmetic or data-transfer operation in a a given context (S, H, L, Q), which is composed of :

- A map $S : V \mapsto \mathbb{Z}$ is the *memory stack*, which binds variable, e.g., $v_1, v_2, \ldots$, to integers. We assume that initially S is the empty stack. We represent S as a stack, instead of an associative array, as it is more standard, because this representation makes it easier to emulate the semantics of ϕ-functions.

$$\frac{n = \textbf{if } o \in V \textbf{ then } S(v) \textbf{ else } o}{\langle v = o, S, H, L, Q\rangle \xrightarrow{i} \langle (v, n) : S, H, L, Q\rangle}$$

$$\langle v = \bullet, S, H, L, n : Q\rangle \xrightarrow{i} \langle (v, n) : S, H, L, Q\rangle$$

$$\frac{S(v_2) = n_2 \qquad S(v_3) = n_3 \qquad n_1 = n_2 \oplus n_3}{\langle v_1 = v_2 \oplus v_3, S, H, L, Q\rangle \xrightarrow{i} \langle (v_1, n_1) : S, H, L, Q\rangle}$$

$$\frac{\textbf{searchPar}(S, \overline{v_1}, \ldots \overline{v_n}) = \overline{n} \qquad \textbf{pushPar}(S, \overline{v}, \overline{n}) = S'}{\langle \overline{v} = \phi(\overline{v_1}, \ldots, \overline{v_n}), S, H, L, Q\rangle \xrightarrow{i} \langle S', H, L, Q\rangle}$$

$$\frac{S(v_1) = n_1 \qquad \textbf{inBlock}(L, n_1) \qquad S(v_2) = n_2}{\langle *v_1 = v_2, S, H, L, Q\rangle \xrightarrow{i} \langle S, H[n_1 \mapsto n_2], L, Q\rangle}$$

$$\frac{S(v_1) = n_1 \qquad \textbf{inBlock}(L, n_1) \qquad H(n_1) = n}{\langle v_2 = *v_1, S, H, L, Q\rangle \xrightarrow{i} \langle (v_2, n) : S, H, L, Q\rangle}$$

$$\frac{S(v_2) = n_2 \qquad \textbf{allocBlock}(L, n_2) = (L', a)}{\langle v_1 = \texttt{alloc}(v_2), S, H, L, Q\rangle \xrightarrow{i} \langle S[v_1 \mapsto a], H, L', Q\rangle}$$

$$\frac{S(v) = a \qquad \textbf{freeBlock}(L, a) = L'}{\langle \texttt{free}(v), S, H, L, Q\rangle \xrightarrow{i} \langle S, H, L', Q\rangle}$$

Figure 3. Semantics of data and arithmetic operations. Following Haskell's syntax, we let the colon (:) denote list concatenation.

- $H : \mathbb{N} \mapsto \mathbb{Z}$ is the *memory heap*, which map adresses to values. We define a special value $\perp$ to fill initial cells of the heap, e.g., initally $H = \lambda x.\perp$ We let $n \oplus \perp = \perp \oplus n = \perp$ for any value n and for any operation $\oplus$. We let $H[v \mapsto a]$ denote function updating, e.g., $H[v \mapsto a] \equiv \lambda x.$if $x = v$ then a else $H(x)$
- $L : \mathbb{N} \mapsto \mathbb{N}$ is a set of *allocated blocks*, which maps addresses in H to contiguous blocks of memory. Addresses insided these blocks are given in ascending order.
- An input chanel $Q : \mathbb{N}$ `list`. This structure represents the input data of the program.

The evaluation of an instruction has thus an effect on this context, as shows Figure 3:

- Assignments (including the read operation) insert new binds between variables and values on the top of S, computing an expression with variables requires to find the actual values of the variables in the expression.
- An instruction such as $\overline{v} = \phi(\overline{v_1}, \ldots, \overline{v_n})$ assigns every variable in the vector $\overline{v}$ in parallel. The auxiliary function **searchPar**$(S, \overline{v_1}, \ldots \overline{v_n})$ will search the stack S, from top towards bottom, for the first occurrences of variables in the set formed by $\overline{v_1} \cup \ldots \cup \overline{v_n}$. [3]
- The semantics of loads, stores, `free` and `alloc` use the memory management library in Figure 4. The list L controls which blocks are valid regions inside H. Function **allocBlock**(L, n) creates a block of size n inside L, and returns a tuple (L', a), with the new list, and the address a of the newly created block. Function **freeBlock**(L, a) removes the block pointed by a from L and returns the new list L without that block. Notice that we are assuming the existence of infinite memory space and do not worry about typical packing problems such as fragmentation. In other words, we never recycle holes inside H. Finally, function **inBlock**(L, a) returns true if address a is within a block tracked by L. By definition, **inBlock**$(L, \perp)$ is always false.

allocBlock$([], n) \Rightarrow ([(0, n)], 0)$
allocBlock$([(a, n') : L], n) \Rightarrow ([(a + n', n) : (a, n') : L], a + n')$

freeBlock$([(a, \_) : L], a) \Rightarrow L$
freeBlock$([(x, n_x) : L], a)$, **if** $x < a \Rightarrow (x, n_x) :$ **freeBlock**(L, a)

inBlock$([(a, n) : L], a')$, **if** $a \leq a' < (a + n)$
inBlock$([(a, n) : L], a')$, **if** $a' \geq (a + n) \Rightarrow$ **inBlock**(L, a')

Figure 4. Memory management library.

Now that we are done with data and variables, it remains to add rules for control flow. These rules are described in Figure 5. We denote changes in control flow via a relation $\xrightarrow{c}$, which operates on four-elements tuples $(\texttt{pc}, S, H, L)$ formed by a (i) a program counter, `pc`; (ii) a stack S; (iii) a heap H; (iv) and a list of allocated memory blocks L. We also use a relation $\xrightarrow{e}$ to denote the last transition of a program, which happens once the program counter points to the `end` instruction. We parameterize the relations $\xrightarrow{c}$ and $\xrightarrow{e}$ with the program onto which they apply.

Memory safety. The state M of a program is given by the quadruple $(\texttt{pc}, S, H, L)$. We say that a program P can *take a step* if from a state M it can make a transition to state M' using the relation $\xrightarrow{c}$. We say that the machine is *stuck* at M if it cannot perform any transition from M, and $\texttt{pc} \neq \texttt{end}$. The evaluation of stores and loads, in Figure 3 are the only rules that can cause our machine to be stuck. This event will happen in case the **inBlock** check fails. Armed with the semantics of our core language, we state, in Definition 3.1, the notion of memory safety.

[3] Recently, Zhao *et al.* [37] have demonstrated, mechanically, that this behavior correctly implements the semantics of SSA-form programs, as long as the programs are well-formed. A well-formed SSA-form program has the property that every use of a variable is dominated by its definition.

$$\frac{P[\texttt{pc}] = \texttt{end}}{P \vdash \langle \texttt{pc}, S, H, L, Q\rangle \xrightarrow{e} \langle S, H, L, Q\rangle}$$

$$\frac{P[\texttt{pc}] = \texttt{br}(v, \ell) \qquad S[v] \neq 0}{P \vdash \langle \texttt{pc}, S, H, L, Q\rangle \xrightarrow{c} \langle \texttt{pc}+1, S', H', L', Q'\rangle}$$

$$\frac{P[\texttt{pc}] = \texttt{br}(v, \ell) \qquad S[v] = 0}{P \vdash \langle \texttt{pc}, S, H, L, Q\rangle \xrightarrow{c} \langle \ell, S', H', L', Q'\rangle}$$

$$\frac{P[\texttt{pc}] = \texttt{jmp}(\ell)}{P \vdash \langle \texttt{pc}, S, H, L, Q\rangle \xrightarrow{c} \langle \ell, S', H', L', Q'\rangle}$$

$$\frac{P[\texttt{pc}] = I \qquad I \notin \{\texttt{end}, \texttt{br}, \texttt{jmp}\} \qquad \langle I, S, H, L, Q\rangle \xrightarrow{i} \langle S', H', L', Q'\rangle}{P \vdash \langle \texttt{pc}, S, H, L, Q\rangle \xrightarrow{c} \langle \texttt{pc}+1, S', H', L', Q'\rangle}$$

Figure 5. The small-step operational semantics of instructions that change the program's flow of control.

Definition 3.1 *A program P at state $\langle \texttt{pc}, S, H, L\rangle$ is safe if there exists no sequence of applications of $\xrightarrow{c}$ that cause it to be stuck.*

4. Symbolic Analyses

In this section we present the analyses that we have used to secure memory accesses in the C programming language. Before diving into the static analyses, in Section 4.1 we discuss the notion of *live range splitting*, as this technique is key to ensure sparseness of our algorithms.

4.1 Live Range Splitting

The typical way to "sparsify" a static analysis is through *live range splitting*. We split the live range of a variable v, at program label l, by inserting a copy $v' = v$ at l, and renaming every use of v to v' in points dominated by l. According to Tavares *et al.* [32], it is enough to split live ranges at places where information originates. These places depend on the type of static analysis that we consider. As we show in Section 4, our two core analyses, symbolic ranges and symbolic regions, require different splitting strategies. The key property that live range splitting must ensure is that the abstract state of any variable be invariant in every program point where this variable is alive. A variable v is alive at a label l if there is a path in the program's control flow graph from l to another label l' where (i) v is used, and (ii) v is not redefined along this path.

Splitting Required by Symbolic Range Analysis. The symbolic range analysis of Section 4.3 draws information from the definition of variables and from conditional tests that use these variables. Thus, to make this analysis sparse, we must split live ranges at these places. Splitting at definitions creates the Static Single Assignment representation.

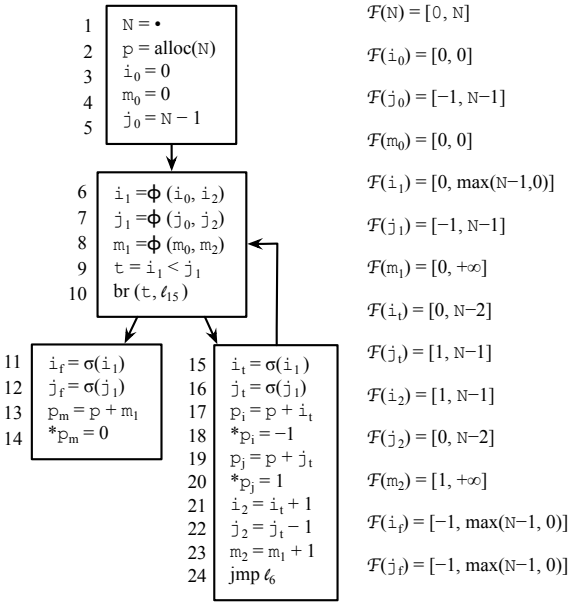


Figure 6. The program in Figure 1 converted into extended static single assignment form, plus the results of symbolic range analysis.

Splitting at conditionals create the representation that Bodik *et al.* have called the *Extended Static Single Assignment form* [6]. However, contrary to Bodik *et al.*, we take transitive dependences between variables into consideration. Conditional tests, such as $cond = a < b; \texttt{br}(cond, l)$, lead us to split the live ranges of a and b at both sides of the branch. We use copies to split live ranger after conditionals. We name the variables created at the "true" side of the branch a_t and b_t, and the variables created at the "false" side of it a_f and b_f. As a convenience, we shall mark these copies with a σ, indicating that they have been introduced due to live range splitting at conditionals. We emphasize that these σ's are just a notation to help the reader to understand our way to split live ranges, and have no semantics other than being ordinary copies [4]. We borrow this notation from Ananian's work [2], who would also indicate live range splitting at branches with σ-functions.

Example 1 (continuing from p. 2) *Figure 6 shows the program in Figure 1 after live range splitting. This program is written in our core language. We preview the results that our symbolic range analysis produces, to show that the abstract state associated with each variable is invariant. By invariant we mean that the symbolic range of each variable v is the same in each point where v is alive.*

Previous implementations of sparse analysis [6, 25] that draw information from conditionals such as $cond = a < b; \texttt{br}(cond, l)$ only split the live ranges of variables used in these conditionals, e.g., a and b. We go beyond, and consider

[4] Bodik *et al.* [6] would name similar instructions π-functions

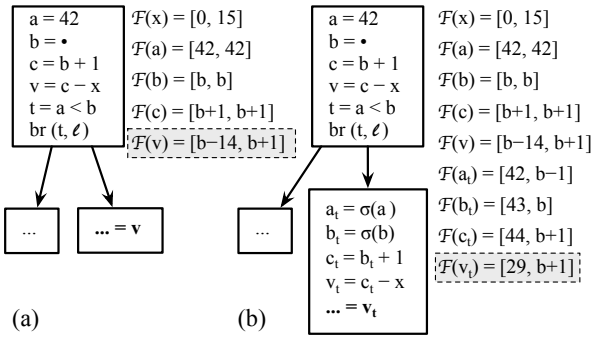


Figure 7. (a-b) Original and transformed program due to live range splitting, plus results of symbolic range analysis (Section 4.3). In this example, we assume that the range of x is [0, 15].

transitive dependences. Let v be a variable different than a and b, and let l' be the label associated with $\texttt{br}(cond, l)$. We split the live range of v along the edge $l' \rightarrow l$, if: (i) a or b depend transitively on v; and (ii) v is used in a label dominated by l. We say that a variable v depends on a variable v' *transitively* if either (1) v appears on the left side of an instruction that uses v'; or (2) v depends transitively on a variable v'', and v'' depends transitively on v'.

Once we split the live range of v, thus creating a new variable, say v_t, we must reconstruct v_t as a function of the transitive chain of dependences that start at either a or b. Figure 7 illustrates this reconstruction. this reconstruction is just a copy of the original chain of dependences. Before moving on, we emphasize that these new instructions have no impact in the runtime of the final program that the compiler produces, because they only exist during our analyses. This more extensive way to split live range improves the precision of our analyses, as Figure 7 demonstrates. As we can see in the figure, we can find a more constrained range for variable v, after the branch, for we have learned new information from the conditional test.

Splitting Required by Symbolic Region Analysis Our symbolic region analysis takes information from instructions that define pointers, and instructions that free memory. We deal with the first source of information via the standard static single assignment form, as we do for the symbolic range analysis. Splitting after `free` is also simple, although this operation requires guidance from alias analysis. If we free the region bound to a pointer p at a program label ℓ, we know that after ℓ every alias of p will point to empty memory space. To make this information clear to our region analysis, we rename every alias p'_k of p to a fresh name p_k". We then initialize each of these new names with the constant zero. In this way, our region analysis will bind these variables to empty array sizes, as we will see in Section 4.4.

4.2 Dealing with Integer Overflows

The integer primitive type has upper and lower bounds in many programming languages, including C, C++ and Java. Thus, there exist numbers that cannot be represented by these types. For instance, considering unsigned integers in C, if a number n is too large to fit into a primitive data type T, then n's value wraps around, and n module T_{max} ends up represented instead. In this case, T_{max} is the largest element in T. This phenomenon might invalidate our analyses. For instance, we can only assume that $i < i + 1$ if we know that i is not the maximum element in the integer type. To circumvent this problem, we instrument every arithmetic operation that has an influence on memory allocation or indexing. We find this set of variables, which we shall call T_b, via the constraints seen in Figure 8. These constraints use a points-to set $\Pi : V \mapsto 2^V$. If $\Pi(v) = A$, then A is the set of aliases of v, e.g., variables that contain addresses that might overlap any region pointed by v. We find Π via an instance of Andersen's style points-to analysis [3], augmented with Lazy Cycle Detection to improve scalability [17]. The constraints of Figure 8 determine a *program slice* [35]. If P is a program, then we say that P' is a slice of P, with regard to the value of a certain variable v at point $\texttt{pc} \in P$, if P' correctly computes the value of v at pc. In our case, we are computing the union of all the slices containing variables used to either index, allocate or free memory. For instance, the constraint for $v_1 = \texttt{alloc}(v_2)$ puts v_2 into T_b, because any instruction used to compute v_2, the size of a memory block, must be guarded against overflows.

We use the *Sparse Evaluation Graph* [12] to generate the constraints seen in Figure 8. The sparse evaluation graph of a SSA-form program contains one vertex for each of its variables, and an edge from v to u if u is used in an instruction that defines v. These edges represent *data dependences*. We must also account for *control dependences*, as defined by Ferrante *et al.* [15, Def.1]. We say that a variable v controls a variable v' if v is used on a branch, e.g., $\texttt{br}(v, l)$, that determines if the instruction that defines v' executes or not. To handle control dependences, we do *predication*. An instruction such as $v = v', \overline{p}$, means that $v = v'$ has been predicated with all the variables in the set $\overline{p}$. The set $\overline{p}$ is formed by all the variables that control the execution of $v = v'$.

We explore transitivity among predicates to avoid marking an instruction with more than one predicate whenever possible. If we operate on *SESE* graph, e.g., control flow graphs that have the single-entry-single-exit property [15], then each instruction can be marked by only one predicate, due to transitivity. The CFGs of go-to free programs naturally produce SESE graphs. The backward slice defined in Figure 8 has a simple geometric interpretation. It determines the set of nodes, in the sparse evaluation graph, that can be reached from a backward traversal, starting from any instruction that either defines or indexes a pointer.

$$
\begin{array}{rcl}
\texttt{free}(v), \overline{p} & \Rightarrow & T_b \supseteq \{v\} \cup \overline{p} \\
\\
v_1 = \texttt{alloc}(v_2), \overline{p} & \Rightarrow & T_b \supseteq \{v_2\} \cup \overline{p} \\
\\
\begin{array}{r} v = v_1, \overline{p} \\ v = v_1 \oplus v_2, \overline{p} \\ v = \phi(v_1, \ldots, v_n), \overline{p} \end{array} & \Rightarrow & \dfrac{v \in T_b}{T_b \supseteq \{v_i\} \cup \overline{p}} \\
\\
*v_1 = v_2, \overline{p} & \Rightarrow & \begin{cases} \text{if } T_b \cap \Pi(v_1) = \emptyset \text{ then} \\ \quad T_b \supseteq \{v_1\} \cup \overline{p} \\ \text{else} \\ \quad T_b \supseteq \{v_1, v_2\} \cup \overline{p} \end{cases} \\
\\
v_2 = *v_1, \overline{p} & \Rightarrow & \begin{cases} \text{if } v_2 \notin T_b \text{ then} \\ \quad T_b \supseteq \{v_1\} \cup \overline{p} \\ \text{else} \\ \quad T_b \supseteq \{v_1\} \cup \overline{p} \cup \Pi(v_1) \end{cases} \\
\\
\texttt{br}(v, l), \overline{p} & \Rightarrow & \dfrac{v \in T_b}{T_f \supseteq \{\overline{p}\}} \\
\\
\text{other instructions} & \Rightarrow & T_b \supseteq \emptyset
\end{array}
$$

Figure 8. Constraints for the backward slice which finds the set T_b of variables to be sanitized against integer overflows. The analysis is parameterized by a points-to set Π.

Example 2 *Figure 9 illustrates the graph-based view of the slice. The graph in Figure 9 (c) represents the dependences in the program of Figure 9 (a). All statements except line ℓ_8 are evaluated under a predication set $\overline{p}$ which is empty. In line ℓ_8, $\overline{p} = \{b_3\}$ because this instruction is controlled by the value of b_3. Figure 9 (b) shows the constraints that we produce for this program, following the rules in Figure 8. Dark-grey boxes in Figure 9 (c) mark the origins of the backward slice; light-grey boxes mark the variables that are part of it. We must guard all the arithmetic operations that define these variables. The only operation that does not require instrumentation is the increment that defines variable v_4 (it has no influence on memory allocation nor indexing).*

Once we have the backward slice of variables T_b, we guard every arithmetic instruction that defines variables in T_b against integer overflows. To achieve this goal, we use the instrumentation proposed by Dietz *et al.* [14], which is available for the LLVM compiler. We have configured this instrumentation to stop the program if an unwanted overflow happens. As we will show in Section 5, this instrumentation adds less than 2% of overhead to the transformed program.

Valid Uses of Integer Overflows. As shown by Dietz *et al.* [14], overflows might be intentional in real-world programs, and at least for unsigned integers this behavior is

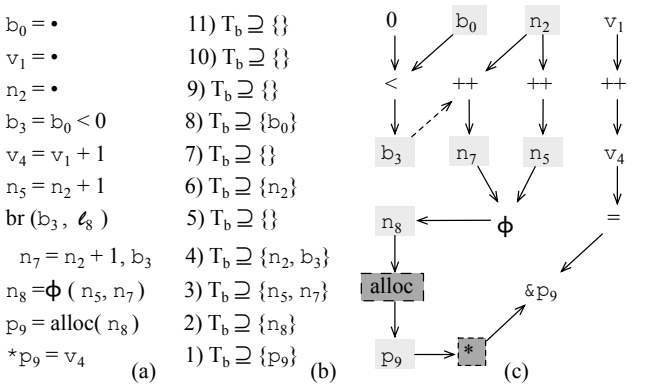


Figure 9. (a) Program that stores variable v_4 in memory. (b) Backward generation of set T_b. (c) Sparse Evaluation Graph.

valid and defined by the C standard. The wraparound behavior can be used, for instance, in the implementation of hash-functions or pseudo-random generators, because it gives developers a cheap surrogate for modular arithmetics. Our guards might change the semantics of the instrumented program, if this program contains legitimate uses of integer overflows in variables that are used to index or allocate memory. We cannot distinguish an intentional use of an integer overflow from a bug. Thus, whoever uses our analyses must be aware that overflows are not allowed on any operation that might influence memory allocation or indexing. We believe that this requirement is acceptable for two reasons. First, ensuring the absence of this phenomenon in memory-related operations greatly improves the precision of our analyses. Secondly, we believe that the occurrence of integer overflows in such operations is a strongly indication of a coding bug. Our belief is backed by our empirical results: we have not found one single integer overflow in the memory-related operations used in SPEC CPU 2006. On the other hand, Dietz *et al.* have found over 200 occurrences of integer overflows in SPEC CPU 2000, and Rodrigues *et al.* [26] have pointed out over 300 sites in SPEC CPU 2006 where overflows took place. We speculate that these overflows are not related to memory allocation or indexing.

Correctness of Integer Overflow Sanitiziation. We say that T_b models P whenever T_b is a solution to the constraints seen in Figure 8, when applied on P. We can infer a number of interesting properties about T_b. Firstly, we know that the variables in T_b control the list of allocated blocks, as we state in Theorem 4.1. Furthermore, we also know that T_b controls which positions of the memory heap are updated, as we state in that theorem. In other words, T_b does not determine the exact values stored in the heap, but it determines in which heap cells these values go. In Theorem 4.1 we let $\xrightarrow{c^* e}$ to denote a sequence of applications of the relation $\xrightarrow{c}$ that end with one application of $\xrightarrow{e}$. These two relations have been defined in Figure 5.

Theorem 4.1 *Let S_1 and S_2 be two configurations of the memory stack of a program P, such that, for any $v \in T_b$, we have that $S_1(v) = S_2(v)$. For any $\mathtt{pc}$, L and Q, we have that, if $(\mathtt{pc}, S_1, H, L, Q) \xrightarrow{c^*e} (S_1', H_1', L_1', Q_1')$, and $(\mathtt{pc}, S_2, H, L, Q) \xrightarrow{c^*e} (S_2', H_2', L_2', Q_2')$, then: (i) $L_1' = L_2'$; (ii) For each address i, $H_1'[i] = \bot \Leftrightarrow H_2'[i] = \bot$, where $H[i] = \bot$ if $H[i]$ has not been assigned a value throughout $\xrightarrow{c^*e}$.*

Proof: We shall define a *Memory Execution Trace* (MET) as a sequence of instructions that contains any of $\mathtt{free}(v)$, $v_1 = \mathtt{alloc}(v_2)$, $v_2 = *v_1$, $*v_1 = v_2$ or any assignment that defines a variable in T_b. Let's denote the i-th instruction in a given MET M as $M[i]$, and a subsequence within M as $M[i:j]$. A MET is a subsequence of instructions within the actual execution trace of a program. If E is an execution trace, we write that $M \subseteq E$. We shall prove that under the assumptions of the theorem, the two execution traces generated by the stacks S_1 and S_2 encode the same METs. We refer to these traces as E_1 and E_2, and to the METs as M_1 and M_2. The proof is by induction on the length of the METs. Henceforth, we assume that we are dealing with *strict programs*, i.e., programs in which variables are defined before being used. We assume that the property holds for any MET with n instructions, e.g., $M_1[1:n] = M_2[1:n]$. First, we prove that instructions at similar indices have the same *opcode* [5]; later, we prove that they read the same parameters.

Let's assume that $M_1[n+1]$ and $M_2[n+1]$ have different opcodes. There are two cases to consider: either these instructions are controlled by predicates with different values, or they are controlled by predicates with the same values. If we follow the first case, then there exists a branch $\mathtt{br}(v, l)$ that separates M_1 and M_2, because v has different values in E_1 and E_2. This separation, due to our induction hypotheses, happens after the execution of $M_1[n]$. We know that v must have different values in E_1 and E_2. We have that $M_1[n+1]$ is controlled by a vector $\overline{p}$ of predicates, which includes v, by the definition of control dependence. A quick inspection of the rules in Figure 8 shows that these predicates are all placed in T_b, for any memory instruction. By the induction hypothesis, they must have the same values - a contradiction.

If $M_1[n+1]$ and $M_2[n+1]$ are controlled by predicates with the same values, then, in compiler jargon, we say that these instructions are in the same *basic block*. Instructions within a basic block always execute in a fixed order. Thus, by the induction hypothesis, the first instruction that depends on any parameter defined in $M_1[1:n]$ must be the same as the first instruction that depends on a value defined in $M_2[1:n]$.

Let's assume that $M_1[n+1]$ and $M_2[n+1]$ have the same opcodes. From this assumption, we proceed by case analysis on the possible shapes of $M_1[n+1]$:

[5] The opcode of an instruction denotes the operation that it represents.

```
1 void read_matrix(int* data, char w, char h) {
2   char buf_size = w * h;
3   if (buf_size < BUF_SIZE) {
4     int c0, c1;
5     int buf[BUF_SIZE];
6     for (c0 = 0; c0 < h; c0++) {
7       for (c1 = 0; c1 < w; c1++) {
8         int index = c0 * w + c1;
9         buf[index] = data[index];
10      }
11    }
12    process(buf);
13  }
14 }
```

```
BUF_SIZE = 120char
strlen(data) = 132char
buf_size = -124char
```

```
w = 0 0 0 0 0 0 1 1 0 = 6char
h = 0 0 0 0 1 0 1 1 0 = 22char
h * w = 1 0 0 0 0 0 1 0 0 = -124char
```

Figure 10. A situation in which an integer overflow would invalidate our symbolic range analysis through a control dependence.

1. $\mathtt{free}(v)$: by induction, v is the same for $M_1[n+1]$ and $M_2[n+1]$, because it is in T_b, as Figure 8 shows.
2. $v_1 = \mathtt{alloc}(v_2)$: by induction, v_2 is the same for both instructions, because it is in T_b according to Figure 8.
3. $*v_1 = v_2$: by induction, v_1 is the same for both, because it is in T_b
4. $v_2 = *v_1$: by induction, v_1 is the same for both, because it is in T_b. Furthermore, any alias of v_1 is also in T_b, and by induction must contain the same values.
5. $v = v_1$: by induction, v_1 is the same for both, because it is in T_b. The other types of assignments are treated in a similar way.

□

We would like to emphasize that handling control dependences is essential to ensure the correcteness of our analysis. This phenomenon might invalidate our analyses, as Figure 10 illustrates. The function `read_matrix` copies a matrix, stored in linear format, to a buffer. The size of this matrix is expected to be given by the product of arguments `w` and `h`, which are eight-bit integers. If we have that $\mathtt{w} = 6$ and $\mathtt{h} = 22$, then $\mathtt{w} \times \mathtt{h} = -124$, due to an integer overflow. The test at line 3 would be true, but we would have "$6 \times 22 - 120 = 12$" invalid accesses at line 9. Notice, in this case, there exists no direct data dependence between inputs and memory indexing. An adversary can, nevertheless, force, through an integer overflow, a bad memory accesss, if we assume that variables `w` and `h` are part of the program's input.

4.3 Symbolic Range Analysis

Range analysis, as originally defined by Cousot and Cousot [9], associates variables with integer intervals. This approach enables several compiler optimizations, but it is not effective to validate memory accesses, as demonstrated by

Logozzo and Fähndrich [19]. The program in Figure 1 illustrates this deficiency. However, a traditional range analysis will not find, in this program, constants onto which to rely upon, to prove that variables `i` and `j` only access valid positions of array `p`. To handle this program, we need a symbolic algebra expressive enough to lets us show that $\texttt{c0}\times\texttt{w}+\texttt{c1} \leq \texttt{w}\times\texttt{h}$, as long as $0 \leq \texttt{c0} < h$ and $0 \leq \texttt{c1} < w$.

To deal with the limitations of range analysis, we adopt the notion of symbolic ranges, originally defined by Blume and Eigenmann [5]. We define the *symbolic kernel* of a program as the set formed by either constants known at compilation time, or variables defined as input values, such as the formal parameters of functions. Henceforth, we will use the general term *symbol* to denote a variable in a symbolic kernel. In this paper, elements of the symbolic kernel are produced by read operations, e.g., $v = \bullet$, or load operations, as we do not keep track of values in memory. We use the abstract interpretation framework [9] in order to generate invariants as *symbolic ranges* over the symbolic kernel of the program. Basically, we will extract a set of (interval) constraints from the program, and then perform a fixpoint algorithm until convergence. The result will be an upper appoximation of "actual values" of the program variables, in all possible execution of the program.

Symbolic expressions We say that E is a symbolic *expression*, if, and only if, E is defined by the grammar below. In this definition, s is a symbol and $n \in \mathbb{N}$.

$$E \quad ::= \quad n \mid s \mid \min(E,E) \mid \max(E,E) \mid E - E \mid E + E \mid E/E \mid E \bmod E \mid E \times E$$

We shall be performing arithmetic operations over the partially ordered set $S = S_E \cup \{-\infty, +\infty\}$, where S_E is the set of symbolic expressions. The partial ordering is given by $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots + \infty$. There exists no ordering between two distinct elements of the symbolic kernel of a program. To define the ordering between two expressions we need the notion of the *valuation* of a symbolic expression. If $M : s \mapsto \mathbb{Z}$ is a map of symbols to numbers, then we define the value of E under M as $(M, E) = n, n \in \mathbb{Z}$. The integer n is the number that we obtain after substituting symbols in E by their values in M. To obtain a *valuation* (E) of a symbolic expression E, we replace its symbols by numbers in $\mathbb{Z}$. We say that $E_1 < E_2$ if any valuation of both E_1 and E_2 under any map M always gives us $(M, E_1) < (M, E_2)$.

Lattice of symbolic expression intervals A *symbolic interval* is a pair $R = [l, u]$, where l and u are symbolic expressions. We denote by $R_\downarrow$ the lower bound l and $R_\uparrow$ the upper bound u. We define the partially ordered set of (symbolic) intervals $S^2 = (S \times S, \sqsubseteq)$, where the ordering operator is defined as:

$$[l_0, u_0] \sqsubseteq [l_1, u_1], \text{ if } l_1 \leq l_0 \wedge u_1 \geq u_0$$

$$\begin{array}{rcl} v = \bullet & \Rightarrow & R(v) = [v, v] \\ v = o & \Rightarrow & R(v) = R(o) \\ v = v_1 \oplus v_2 & \Rightarrow & R(v) = R(v_1) \oplus^I R(v_2) \\ v = \phi(v_1, v_2) & \Rightarrow & R(v) = R(v_1) \sqcup R(v_2) \\ \text{other instructions} & \Rightarrow & \emptyset \end{array}$$

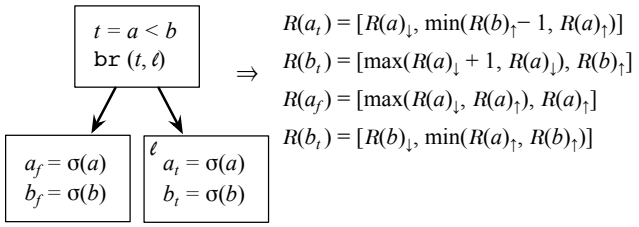


Figure 11. Constraints for the symbolic range analysis

We then define the semi-lattice SymBoxes of symbolic intervals as $(S^2, \sqsubseteq, \sqcup, \emptyset, [-\infty, +\infty])$, where the join operator "$\sqcup$" is defined as:

$$[a_1, a_2] \sqcup [b_1, b_2] = [\min(a_1, b_1), \max(a_2, b_2)]$$

Our lattice has a least element $\emptyset$, so that:

$$\emptyset \sqcup [l, u] = [l, u] \sqcup \emptyset = [l, u]$$

and a greatest element $[-\infty, +\infty]$, such that:

$$[-\infty, +\infty] \sqcup [l, u] = [l, u] \sqcup [-\infty, +\infty] = [-\infty, +\infty]$$

Clearly, this lattice is infinite; therefore, in order to end up the computation of the set of constraints we use a widening operator defined by (under the assumption $R_1 \sqsubseteq R_2$):

$$R_1 \nabla R_2 = [l, u], \text{ where } \begin{cases} l = R_{1\downarrow} & \text{if } R_{1\downarrow} = R_{2\downarrow} \\ l = -\infty & \text{otherwise} \\ u = R_{1\uparrow} & \text{if } R_{1\uparrow} = R_{2\uparrow} \\ u = +\infty & \text{otherwise} \end{cases}$$

This is the extension of the classical widening on intervals to symbolic intervals. Like the classical widening on interval, a lower (resp. upper) bound of a given symbolic interval can only be stable or diverge towards $-\infty$ (resp $+\infty$), thus our widening operator will ensure the convergence of our analysis.

Abstract interpretation on the E-SSA form. To apply the abstract interpretation framework, we also have to give an interpretation of the operations of the program. This is done in Figure 11.

- assigments after reads give only symbolic information.
- assigments to expressions causes the expression to be evaluated in the "symbolic range world": see Figure 12.

- $[l_0, u_0] +^I [l_1, u_1] = [l_0 + l_1, u_0 + u_1]$
- $[l, u] +^I \emptyset = [l, u]$
- $[l_0, u_0] \times^I [l_1, u_1] = [\min(T), \max(T)]$, where $T = \{l_0 \times l_1, l_0 \times u_1, u_0 \times l_1, u_0 \times u_1\}$
- $[l, u] \times^I \emptyset = [l, u]$

Figure 12. Range Computation library

	0^{th}	1^{th}	2^{nd}	$3^{rd}(+\nabla)$
N	$\emptyset$	$[0, N]$	$[0, N]$	$[0, N]$
i_0	$\emptyset$	$[0, 0]$	$[0, 0]$	$[0, 0]$
m_0	$\emptyset$	$[0, 0]$	$[0, 0]$	$[0, 0]$
j_0	$\emptyset$	$[-1, N-1]$	$[-1, N-1]$	$[-1, N-1]$
i_1	$\emptyset$	$[0, 0]$	$[0, max]$	$[0, max]$
j_1	$\emptyset$	$[-1, N-1]$	$[-1, N-1]$	$[-1, N-1]$
m_1	$\emptyset$	$[0, 0]$	$[0, 1]$	$[0, +\infty]$
i_t	$\emptyset$	$[0, N-2]$	$[0, N-2]$	$[0, N-2]$
j_t	$\emptyset$	$[1, N-1]$	$[1, N-1]$	$[1, N-1]$
i_2	$\emptyset$	$[1, N-1]$	$[1, N-1]$	$[1, N-1]$
j_2	$\emptyset$	$[0, N-2]$	$[0, N-2]$	$[0, N-2]$
m_2	$\emptyset$	$[1, 1]$	$[1, 2]$	$[1, +\infty]$
i_f	$\emptyset$	$-$	$-$	$[-1, max]$
j_f	$\emptyset$	$-$	$-$	$[-1, max]$

Figure 13. Symbolic range analysis in program of Figure 6

- ϕ functions are "join nodes", thus we perform a (symbolic) interval union.
- On intersection nodes (tests), we perform a (symbolic) interval intersection.

We solve the system of contraints using a Kleene iteration on the system of constraints. The analysis is sparse since we do not need the control points any more. The information is only attached to variables. Widening is applied on a ϕ node only after 3 iterations of symbolic evaluation ("delayed widening"). This decision is arbitrary, and a larger number of iterations might increase the precision of our results. However, in our experiments we only observed very marginal gains by using more iterations.

Example 1 (continuing from p. 2) *Figure 13 shows the steps that our analysis performs on the program of Figure 6. The order in which we evaluate constraints is given by the reverse post-ordering of the program's control flow graph. This ordering tends to reduce the number of iterations of our fixed point algorithm [23, p.421]. However, any order of evaluation would led to the same result. In this example, we let "max" be* $max(0, N-1)$.

On the evaluation of symbolic expressions. As shown in the preceeding paragraphs, we have to evaluate symbolic expressions (simplification of expressions like $max(e_1, e_2)$, equality tests, ...). We rely on GiNaC [4], a library for symbolic manipulation, to perform these operations[6]. For the equality test, if GiNaC is not able to prove a given equality between symbolic expressions, we conservatively assume that the two expressions are not comparable. As a consequence, we may widen an expression to $+\infty$ even if it was stable.

Correctness of the Symbolic Range Analysis. Abstract values of the SymBoxes domain are functions R that associate to each variable v of the program a symbolic interval. The concretisation of a given symbolic range is the set of valuations (variables and symbols) that satisfy the induced constraints:

$$\gamma_{\mathsf{SymBoxes}}(R) = \{(\sigma, M), \sigma \text{ valuation of variables and } M \text{valuation of symbols} \text{ s.t. } \forall v \in V, (M, R(v)_\downarrow) \leq \sigma(v) \leq (M, R(v)_\uparrow)\}$$

Here the union of the valuations σ and M corresponds to the stack (S) defined in the semantics of the core language (Figure 5). The order of application of σ and M does not matter.

Theorem 4.2 *The former analysis always returns an over-approximation of the actual ranges of the variables of the program (no matter the valuations of the symbols would be).*

Proof: The abstract interpretation framework implies that any valuation of the variables and symbols of the program (Stack) belongs to the concretisation of the intervals that our analyses finds ($\gamma_{\mathsf{SymBoxes}}(R)$). □

4.4 Symbolic Region Analysis

As we mention in Section 6, the vast majority of previous algorithms to eliminate array bound checks have been designed for the Java programming language. In Java, arrays are associated with their sizes, and this information is available at runtime. In other words, Java programs often uses the `array.length` attribute to iterate over an array, and we can rely on the semantics of this field to eliminate more access guards. In C, arrays are not packed together with size information. We have to infer this size automatically. To solve this problem, we resort to static analysis. We have designed a novel *region analysis*, which binds each pointer to an interval of *valid offsets*.

Our analysis of regions rests on the semi-lattice SymRegion $(S^2, \sqsubseteq, \sqcap, [-\infty, +\infty], \emptyset)$, which is the inverse of the structure used in the symbolic range analysis of Section 4.3. Here, we have a meet operator "$\sqcap$", such that:

$$[a_1, a_2] \sqcap [b_1, b_2] = \begin{cases} \emptyset, & \text{if } a_2 < b_1 \text{ or } b_2 < a_1 \\ [\max(a_1, b_1), \min(a_2, b_2)], & \text{otherwise} \end{cases}$$

[6] GinaC is available at `http://www.ginac.de/`

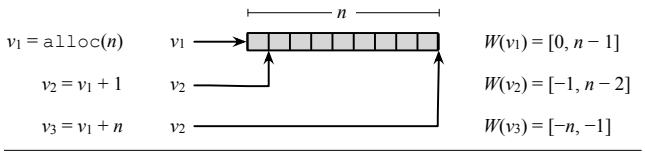


Figure 14. Semantics for "valid offsets"

The least element of our semi-lattice is $\emptyset$, so that:

$$\emptyset \sqcap [l, u] = [l, u] \sqcap \emptyset = \emptyset$$

And the greatest is $[-\infty, +\infty]$, which we define as follows:

$$[-\infty, +\infty] \sqcap [l, u] = [l, u] \sqcap [-\infty, +\infty] = [l, u]$$

Therefore, whereas in Section 4.3 we were always expanding ranges, here we are always contracting them. In other words, the symbolic range analysis finds the largest ranges covered by variables, i.e., it is a *may* analysis. On the other hand, the symbolic region analysis finds the narrowest regions that pointers can dereference, i.e., it is a *must* analysis. In the abstract interpretation jargon, we say that we are computing *under-approximations*.

Like the symbolic range analysis from Section 4.3, the region analysis also associates with each variable an abstract state given by an interval. However, here this abstract state has a very different interpretation. If we say that $W(p) = [l, u]$, then we mean that all the addresses between the offsets $p+l$ and $p+u$ are valid. Figure 14 clarifies this semantics. The first instruction of Figure 14 allocates n words in memory, and assigns the newly created region to pointer v_1. Thus, given a stack S, if $b_1 = S(v_1)$ is the value of v_1, then any address between $b_1 + 0$ and $b_2 + n - 1$ is valid. The second instruction increments v_1, and calls the new address v_2. Similarly, if $b_2 = S(v_2)$, then the address $b_2 - 1$ is valid, and the address $b_2 - n - 2$ is also valid. As we state in Theorem 4.3, it is always safe to dereference a pointer if it includes the address zero among its valid offsets.

We describe an instance of region analysis as a set of constraints. These constraints are extracted from the program's source code, according to the rules in Figure 15. The figure naturally distinguish between *scalars* and *pointers*. Pointers are variables that have been initialized with the `alloc` instruction, or that are computed as functions of other pointers. Scalars are all the other variables; hence, they are always bound to the region $\emptyset$. For the last constraint, we must encode the fact that we cannot add a pointer and a variable, thus v_3 must be a scalar. Therefore, if p_1 is defined as $p_1 = p_2 + v$, then its region is given by the admissible region of p_2 addressed with all the possible values of v. That is why we must consider the interval range of v. Before moving on, we draw the reader's attention to the abstract semantics of the $\texttt{free}(v)$ instruction. As we explained in Section 4.1, this instruction leads us to rename every alias of v, so that all of these new names will be bound to the new region $\emptyset$.

$$\begin{array}{rcl}
v = c, c \in \mathbb{N} & \Rightarrow & W(v) = \emptyset \\
v_1 = \texttt{alloc}(v_2) & \Rightarrow & W(v_1) = [0, R(v_2)_{\downarrow} - 1] \\
a = b & \Rightarrow & W(a) = W(b) \\
\texttt{free}(v); & & \\
v'_1 = 0; v'_2 = 0 \ldots & \Rightarrow & W(v'_1) = W(v'_2) = \ldots = \emptyset \\
v_1, v_2, \ldots \in \Pi(v) & & \\
v = \phi(v_0, v_1) & \Rightarrow & W(v) = W(v_0) \sqcap W(v_1) \\
v_1 = v_2 + n & & \\
\text{with } n \in \mathbb{N} & \Rightarrow & W(v_1) = \begin{cases} \emptyset \text{ if } W(v_2) = \emptyset, \text{else} \\ [W(v_2)_{\downarrow} - n, W(v_2)_{\uparrow} - n] \end{cases} \\
v_1 = v_2 + v_3 & & \\
\text{with } v_3 \text{ scalar} & \Rightarrow & W(v_1) = \begin{cases} \emptyset \text{ if } W(v_2) = \emptyset \\ \text{else } [W(v_2)_{\downarrow} - R(v_3)_{\downarrow}, \\ \qquad W(v_2)_{\uparrow} - R(v_3)_{\uparrow}] \end{cases}
\end{array}$$

Figure 15. Constraint generation for the symbolic region analysis.

Our region analysis uses a widening operator for ϕ-functions, to ensure that our algorithm terminates in face of pointer arithmetics. Because this operator reduces ranges, it is called a *lower widening* [7] , under the terminology of [21]. This operator is defined as follows:

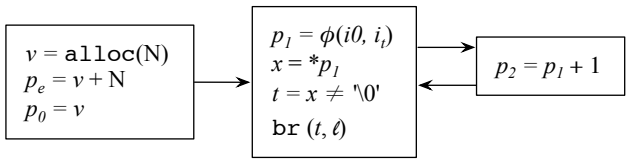
$$R_1 \nabla R_2 = \begin{cases} \emptyset & \text{if } R_{2\downarrow} > R_{1\downarrow} \text{ or } R_{2\uparrow} < R_{1\uparrow} \\ R_2 & \text{otherwise} \end{cases}$$

Example 1 (continuing from p. 2) *If we run our region analysis on the program of Figure 6, then we get that $W(p) = [0, N-1]$, $W(p_i) = [0, 1]$, $W(p_j) = [-1, 0]$, and $W(p_m) = [0, -\infty] = \emptyset$. These ranges correctly tells us, for instance, that the largest (safely) addressable offset from address p is $N - 1$.*

Example 3 *Figure 16 shows how widening ensures that our region analysis converges. This program implements a typical character search in a string, e.g., `for(v = p; p != '0'; p++)`. In this example we have widened after p_1, the variable defined by a ϕ-function, had changed twice. After widening we reach a fixed-point in the third round of abstract interpretation.*

Example 1 (continuing from p. 2) *The ranges that we find for the program in Figure 6 let us remove bound-checks for the memory accesses at lines 18 and 20. The region of p_i tells us that p_i and $p_i - 1$ are safe addresses. The region of p_j indicates that p_j and $p_j + 1$ are also safe addresses. On*

[7] Let us point out the fact that, unlike the proposition of [21], we directly widen to $\emptyset$ when one of the bounds is not stable.



	0^{th}	1^{st}	2^{nd}	$\triangledown$	3^{rd}
v	$[-\infty, +\infty]$	[0, N−1]	[0, N−1]	[0, N−1]	[0, N−1]
p_e	$[-\infty, +\infty]$	[−N, −1]	[−N, −1]	[−N, −1]	[−N, −1]
p_0	$[-\infty, +\infty]$	[0, N−1]	[0, N−1]	[0, N−1]	[0, N−1]
p_1	$[-\infty, +\infty]$	[0, N−1]	[0, N−2]	∅	∅
p_2	$[-\infty, +\infty]$	[−1, N−2]	[−2, N−3]	[−2, N−3]	∅

Figure 16. An example where we have used widening to ensure that region analysis converges.

the other hand, the range of p_m tells us that it is not safe to dereference this pointer without a bound-check. In this case, we have a false-positive, because we conservatively assume that a memory access is unsafe.

Correctness of the Symbolic Region Analysis. Abstract values of the SymRegion domain are functions W that associate to each variable p a symbolic interval. The concretisation of W is the set of tuples (S, H, L) (program context) which correspond to adresses in the ranges $[p_i + \ell_i, p_i + u_i]$:

$$\gamma_{\mathsf{SymRegion}}(W) = \{(S, H, L, Q) \text{ such that } \forall p \in V, \forall \alpha, \text{ if } H(S(p) + S(W(p))_{\downarrow})) \leq \alpha \leq H(S(p) + S(W(p))_{\uparrow}))) \text{ then } \mathbf{inBlock}(L, \alpha)\}$$

Theorem 4.3 states the key property of our symbolic region analysis. We have defined the theorem for loads, but it is also true for instructions such as $*v_1 = v_2$, which store the contents of v_2 into the address pointed by v_1.

Theorem 4.3 *Let P be a program and $\mathtt{pc} \in \mathbb{N}$, such that $P[\mathtt{pc}]$ is $v_2 = *v_1$. If $0 \in W(v_1)$, then P cannot be stuck at $\mathtt{pc}$.*

Proof: (Sketch) The abstract interpretation framework implies that our concretisation is a subset of the actual valid addresses in L. Thus $0 \in W(v_1)$ garanties that v_1 is a valid address. □

4.5 Tainted Flow Analysis

We have included an optional tainted flow analysis in our framework. If we are interested in preventing only buffer overflows caused by malicious inputs, then we can restrict our attention to operations that are *influenced* by *values* produced by external functions. To illustrate this observation, we use Figure 17. Figure 17 (a) shows a program that reads two values, `m` and `n` and use them to control the writing of an array `v`. Variable `n` controls the maximum extension of the array that is written. If this variable is larger than the size of the array, bad memory accesses will take place. The variable

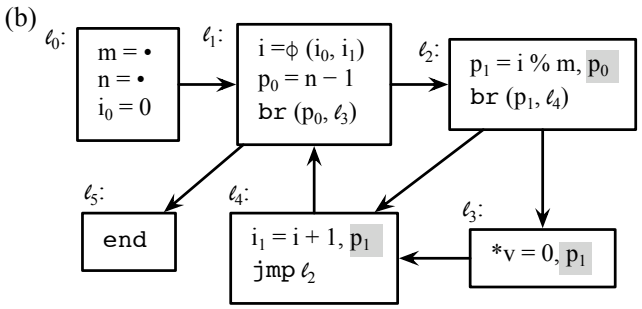
(a)

```
1 m = read();
2 n = read();
3 int i = 0;
4 for (; i < n; i++) {
5     if (i % m)
6         v[i] = 0
7 }
```

(d)

$T_f = \{$ m, p_0, Π[v], n, p_1, i_1, i $\}$

(b)



(c)

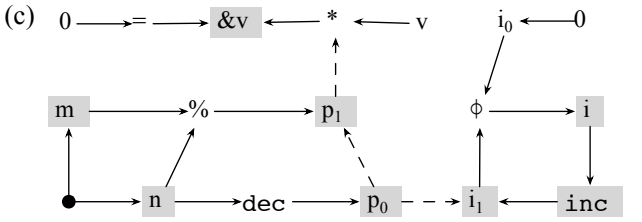


Figure 17. (a) Program containing implicit tainted flow that could cause out-of-bounds access. (b) Intermediate representation. Predicates propagating implicit information are shown in gray. (c) Dependence graph. Control flow edges are dashed. (d) Tainted set T_f.

`n` is part of the input of the program; hence, we assume that it can be controlled by a malicious user. By feeding the program with a large value of `n`, this user can cause an invalid access at line 6 of Figure 17 (a). Our taint analysis identifies which memory accesses can be manipulated by malicious users, in such a way that the other accesses do not need to be guarded. Our tainted flow analysis considers the following set of external functions:

- functions not declared in any file that is part of the compiled program;
- functions without a body;
- functions that can be called via pointers.

The idea behind this tainted flow analysis is as follows: we want to instrument stores, loads and memory allocations that might be controlled by external sources of data. In this case, we are assuming that every source of external data is untrusted.

Tainted flow analysis has been discussed exhaustively in the literature [28]; thus, it is not a novel contribution of this

$$
\begin{array}{rcl}
v = \bullet, \overline{p} & \Rightarrow & T_f \supseteq \{v\} \\
v = v_1, \overline{p} & & \\
v = v_1 \oplus v_2, \overline{p} & \Rightarrow & \dfrac{\exists v_i \in T_f \vee \overline{p} \cap T_f \neq \emptyset}{T_f \supseteq \{v\}} \\
v = \phi(v_1, \ldots, v_n), \overline{p} & & \\
v_1 = \texttt{alloc}(v_2), \overline{p} & \Rightarrow & \dfrac{v_2 \in T_f \vee \overline{p} \cap T_f \neq \emptyset}{T_f \supseteq \{v_1\}} \\
*v_1 = v_2, \overline{p} & \Rightarrow & \dfrac{v_2 \in T_f \vee \overline{p} \cap T_f \neq \emptyset}{T_f \supseteq \Pi(v_1)} \\
v_2 = *v_1, \overline{p} & \Rightarrow & \dfrac{(\Pi(v_1) \cup \overline{p}) \cap T_f \neq \emptyset}{T_f \supseteq \{v_2\}} \\
\texttt{br}(v, l), \overline{p} & \Rightarrow & \dfrac{p \cap T_f \neq \emptyset}{T_f \supseteq \{v\}} \\
\text{other instructions} & \Rightarrow & T_f \supseteq \emptyset
\end{array}
$$

Figure 18. Constraints for the forward slice which finds the set T_f of variables that are influenced by input values. The analysis is parameterized by a points-to set Π.

work. We run tainted flow analysis on the same dependence graph that we have used to implement our integer overflow slice from Section 4.2. Therefore, the tainted flow analysis determines a slice in the program dependence graph, which handles both implicit and explicit flows of information. The differences from the analysis of Section 4.2 are as follows:

- The tainted flow analysis is a forward slice of the program. The overflow analysis is a backward slice.
- The sources of the tainted flow slice are the instructions that use values produced by external sources. The sources of the integer overflow slice are the memory operations.
- The sinks of the tainted flow slice are memory accesses. The sinks of the overflow slice are integer arithmetic operations.

Like the other algorithms from Section 4, our tainted flow analysis is sparse. We also track control dependences between branches and instructions through predication. As an example, Figure 17 (b) shows how each instruction of a program is predicated. Notice that we only have to predicate the instructions in block l_4 with p_1, and not with p_0 and p_1, due to transitivity: p_0 already predicates p_1. Such transitivities are discovered using Ferrante *et al.*'s techniques [15]. The sparse evaluation graph in Figure 17 (c) shows the control dependence edges that this predication generates. Hence it associates with the entire live range of a variable one of two states: *tainted* or *clean*. Memory accesses that are indexed only by clean variables do not need to be guarded against overflows. The other accesses will still be analyzed by our other algorithms of Section 4, and only if our symbolic analyses cannot prove that they are safe, we will have to maintain bound-checks.

The forward flow analysis given in Figure 18 builds the set T_f of tainted variables. Like the backward analysis of Section 4.2, this analysis is parameterized by a points-to set $\Pi : V \mapsto 2^V$. If P is a program, and T_f is the taint set produced by our analysis, then we say that T_f models P. The set T_f for our example is given in Figure 17. We use a lattice with two elements: either a variable is *tainted*, in which case it belongs to T_f, or it is *clean*. Therefore, our taint analysis reduces to traditional program slicing [33], which we solve via graph traversal, as Figure 17(c) illustrates. Theorem 4.4 states the key non-interference property of our tainted flow analysis.

Theorem 4.4 *Let Q_1 and Q_2 be two input queues, such that, for some $\texttt{pc}$, S, H and L we have $(\texttt{pc}, S, H, L, Q_1) \xrightarrow{c*e} (S_1, H_1, L_1, Q_1')$, and $(\texttt{pc}, S, H, L, Q_2) \xrightarrow{c*e} (S_2, H_2, L_2, Q_2')$. For any variable v, such that $v \notin T_f$, we have that $S_1(v) = S_2(v)$.*

Proof: (Sketch) This proof is similar to the proof of Theorem 4.1, and so we omit its details. We need the notion of an *Input Execution Trace*, which is formed by all the instructions that define variables in T_f. The proof follows by induction on the length of these traces, alongside case analysis on the Rules seen in Figure 18. □

4.6 Inter-procedural Analysis

All the analyses that we describe in this paper are *inter-procedural*; albeit context-insensitive for the sake of scalability, as we do not consider the state of the function stack when analyzing calls. To analyze a program, we visit functions in an order defined by its call-graph. This graph contains a vertex for each function, and an edge from function f to function g if f contains a call to g among its instructions. We visit functions following a depth-first traversal of the program's call graph. If the program does not contain recursive calls, then its call-graph is a directed acyclic graph. In this case, the depth-first traversal, starting from `main`, the entry point of a C program, gives us the topological ordering of functions. In other words, we only visit a function f after we have visited any function that calls f. As we will explain shortly, this ordering lets us compute all the information that we need to analyze a function, before visiting the function itself. On the other hand, it forces us to assume that values returned from functions are always part of the symbolic kernel, as we have no information about them.

After we analyze the body of a function g, we have enough information to compute a "Less-Than" relationship between variables declared in its body. We compute such relationship for every pair of variables that are used as actual parameters of other functions called in the body of g. Let $f(v_1, v_2, \ldots, v_n)$ be an instruction, within the body of g, that calls a certain function f with actual parameters $v_i, 1 \leq i \leq n$. We determine the "Less-Than" relation between

```
1 void keep(int *s, int i_s, int N_s, int* d, int i_d, int N_d){
2   int x = 0 ;
3   if (i_s < N_s) {
4     x = s[i_s];
5   }
6   if (i_d < N_d) {
7     d[i_d] = x;
8   }
9 }
10
11 int main() {
12   unsigned N_x = •;
13   unsigned N = N_x + 1;
14   int* p  = alloc(N);
15   unsigned N_y = •;
16   unsigned N_0 = N_y + 1;
17   int* p_0 = alloc(N_0);
18   keep(p_0, 0, N_0, p, 0, N)
19   unsigned N_1 = 4;
20   int* p_1 = alloc(N_1);
21   keep(p_1, 1, N_1, p, 1, N)
22 }
```

(a)

$R(i_s) = [0, s]$	$W(d) = [0, d]$
$W(s) = [0, s + 1]$	$R(i_d) = [0, 1]$
$R(N_s) = [0, s + 2]$	$R(N_d) = [0, d + 1]$

(b)

	p_0	0	N_0	p	0	N
p_0	=	<	>	?	<	?
0	>	=	>	>	=	>
N_0	<	<	=	?	<	?
p	?	<	?	=	<	>
0	>	=	>	>	=	>
N	?	<	?	<	<	=

(c)

	p_1	1	N_1	p	1	N
p_1	=	<	>	?	?	?
0	>	=	>	>	>	>
N_1	<	<	=	?	?	?
p	?	<	?	=	?	>
1	>	<	>	?	=	?
N	?	<	?	<	?	=

Figure 19. Example of inter-procedural analysis. (a) Symbolic bounds of formal parameters. "Less-than" analysis on the actual parameters of (b) first call to `keep`, and (c) second call to `keep`.

each $v_i, v_j, 1 \leq i, j \leq n$. Additionally, we also determine minimum lower bounds to all the scalar variables v_i.

We implement the less-than check differently, depending on the actual parameters being pointers or scalars. Our less-than test between scalars v_i and v_j amounts to checking if $R(v_j)_\uparrow < R(v_i)_\downarrow$. If v_i is a pointer, then we check if $R(v_j) \subseteq W(v_i)$. We perform these checks through `GiNaC`, the same library that we mentioned in Section 4.3. We have augmented `GiNaC` with operations to handle inequalities involving the symbols *min* and *max*, because these expressions are part of our framework, and did not exist originally in that library. To determine the lower bounds of a formal parameter v_i that is a scalar, we compute the minimum among all the actual instances of v_i. Example 4 illustrates our technique.

Example 4 *Figure 19 shows how we handle function calls. In the figure, we have two calls of `keep`. Each call generates different "less-than" relationships, which we show in part (b) and (c) of the figure. We use the question mark to indicate that we do not know size relation between two variables.*

To propagate information computed for actual parameters to formal parameters we use a simple *meet* operator. Let $f_1(v_{11}, \ldots, v_{1n}), \ldots, f_m(v_{m1}, \ldots, v_{mn})$ be all the calls of a function f throughout a program P, and let $f(v_1, \ldots, v_n)$ be the declaration of f in P. If, for every call, we have that $v_{ij} < v_{ik}, 1 \leq j, k \leq n, 1 \leq i \leq m$, then we let $v_j < v_k$ when analyzing f. Each v_i is part of the symbolic kernel of f, thus, these variables are assigned to symbolic bounds. To determine these bounds, we sort all the formal parameters v_i according to the Less-Than relation. We call a sequence of variables sorted in this way a *Less-Than Chain*. If v_i is the first, i.e., smallest, variable in such a chain, we let its bound to be $R(v_i) = [min(v_{ki}), s]$, where $min(v_{ki})$ is the minimum bound among all the actual parameters v_{ki}, and s is a fresh symbol. If v_i is a pointer, then we let its region be $W(v_i) = [0, s]$. If v_j is the second element in this chain, we let its range to be $R(v_j) = [min(v_{kj}), s + 1]$, and so on. In the presence of recursion we initialize all the formal parameters of mutually recursive functions with fresh symbols, without implementing the less-than check.

Example 4 (continuing from p. 14) *Figure 19 (a) gives the symbolic states - ranges and regions - of all the formal parameters of function `keep`. We have two less-than chains in this program: (i_s, s, N_s) and (d, N_d). The first chain leads us to initialize the symbolic bounds of i_s, s and N_s with $s, s + 1$ and $s + 2$ respectively.*

Discussion: our approach is different than the traditional way to implement a context-insensitive inter-procedural analysis. Before initializing the abstract state of a formal parameter v_i, the traditional approach joins the state of all the actual parameters v_{ki}, and copies this state to v_i, following the technique that Nielson *et al* call the *naive inter-procedural analysis* [23, p.88]. This is what we do in the abstract interpretation of ϕ-functions, as defined in Figures 11 and 15. However, when analyzing function calls, we change information, before performing the naive propagation. Instead of doing a join of symbolic ranges, we join less-than relations. We believe that our approach is better than the naive inter-procedural method, as we explain in Example 4.

Example 4 (continuing from p. 14) *Had we performed a naive join of abstract states in Figure 19, we would have gotten $R(i_s) = [0, 0] \sqcup [1, 1] = [0, 1]$, and $W(s) = W(p_0) \sqcap W(p_1) = [0, min(N_y + 1 - 1, 4 - 1)] = [0, min(N_y, 3)]$. Given that we do not know that $N_y > 1$, we would not be able to prove that the memory access at line 4 is always safe. Our less-than relations let us deal with this shortcoming of the naive approach.*

4.7 Implementation details

We have combined the three static analyses of Section 4 into a framework that we call *GreenArrays*. Figure 20 shows how these analyses are organized. The tainted flow analysis, which we have described in Section 4.5, is optional. As we show in Section 5, it increases by a small margin the number of bound checks that we can eliminate.

GreenArrays works on top of AddressSanitizer. AddressSanitizer is an extension of the well-known LLVM compiler [18], that produces memory safe binaries out of C programs. This tool relies on a modified memory allocation library that generates enough meta-information to support runtime access checks. AddressSanitizer shadows every chunk of memory that it allocates. At runtime, each memory

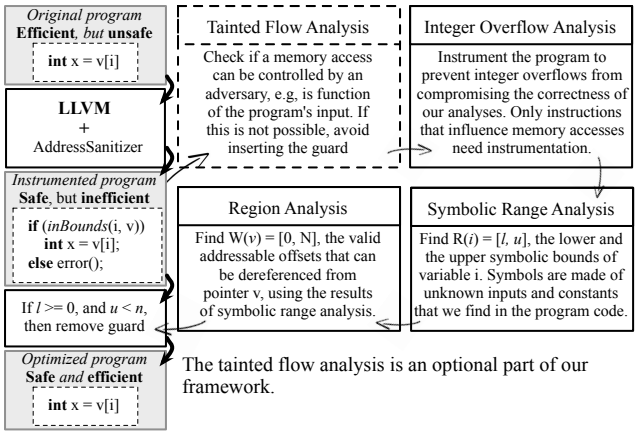


Figure 20. Overview of our pipeline of static analyses.

access is matched against its shadow area, and an attempt to read or write unallocated data triggers an exception. Our modified version of AddressSanitizer uses the analyses from Section 4 to prove that some memory accesses are always safe. These accesses need not not to be guarded; hence, we eliminate them.

5. Experiments

In this section we report experiments that we have performed using our GreenArrays framework. The goals of this section are i) to show that our approach is practical - all our analyses run in acceptable time; ii) to show that it is effective - we can eliminate a reasonable number of bound-checks, hence speeding up safe code and reducing its energy consumption; and iii) to show that our technique is competitive with state-of-the-art approaches to bound-check elimination. Unless stated otherwise, we run our experiments in a twelve-core Intel(R) Xeon(R) CPU E5-2620 at 2.00GHz, with 15,360KB of cache, and 16GB of RAM. Neither our compiler, nor our benchmarks, run in parallel.

Benchmarks. We chose to report numbers for the integer programs available in the SPEC CPU 2006 benchmark suite. Figure 21 presents static data about them. We order the benchmarks by number of LLVM three-address instructions, in this table, and in every chart that follows. In total, SPEC gave us 2,194,727 instructions to analyze. The LLVM IR has five instructions that may overflow: ADD, SUB, MUL, TRUNC (also bit-casts) and SHL (left shift). These opcodes account for 3.5% (76,294) of the instructions in our benchmark suite. Not all of them influence memory allocation, but 2.5% (59,362) of them do. Hence, we had to instrument 77.8% (59K/76K) of the arithmetic instructions against integer overflows, using the analysis of Section 4.2.

Run Time of Analyses. Figures 22 and 23 show the time to run our analyses. We show also the time to build the graph of data dependences, and the graph of control dependences. Except for the tracking of implicit flows, all our analyses

Benchmark	I	TI	A	OA	OA/I
gcc	801,918	94%	15,548	12,233	0.015
xalancbmk	593,895	71%	12,868	9,594	0.016
perlbench	284,039	94%	10,342	8,175	0.029
gobmk	145,670	71%	12,208	9,350	0.064
omnetpp	91,822	68%	2,631	2,041	0.022
h264ref	144,266	80%	13,458	10,292	0.071
hmmer	67,735	81%	3,670	3,202	0.047
sjeng	30,275	89%	2,442	2,077	0.069
bzip2	17,439	94%	1,700	1,196	0.068
astar	8,662	82%	649	572	0.066
libquantum	6,446	89%	614	511	0.079
mcf	2,560	85%	164	119	0.046

Figure 21. Static results for the programs in SPEC CPU 2006. **I**: number of LLVM instructions. **TI**: percentage of tainted instructions (Section 4.5). **A**: number of arithmetic instructions. **OA**: memory sensitive arithmetic instructions (which we have instrumented according to Section 4.2).

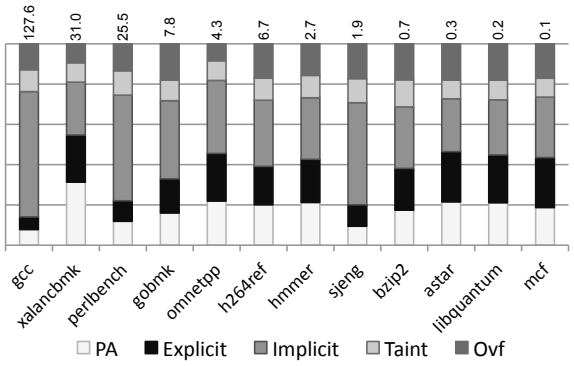


Figure 22. Time taken by static analyses (first part). **PA**: Pointer analysis. **Explicit**: build dependence graph with explicit dependences. **Implicit**: updated graph with control flow dependences. **Taint**: analysis of Section 4.5. **Ovf**: analysis of Section 4.2. Numbers above bars give total execution times, in seconds.

seem to be linear in practice. For the SPEC data-set, all the coefficients of correlation (CC) between number of assembly instructions and time to run the analysis were above 0.98. A coefficient close to 1.0 indicates a linear behavior. The algorithm that adds control edges to the dependence graph has CC = 0.88. This is understandable: we are adding edges from a predicate to all the variables defined at basic blocks that it controls; hence, the final graph is likely to be dense.

The run time of our analyses compares favorably to the run time of equivalent algorithms, as Figure 24 shows. The figure compares the runtime of our symbolic range analysis, Rodrigue *et al.*'s range analysis, and Pentagons, for the 60 largest benchmarks in the LLVM test suite and SPEC CPU 2006. These programs gave us over 5.18 million bytecodes to analyze. We tried to our best to be as faithful to the

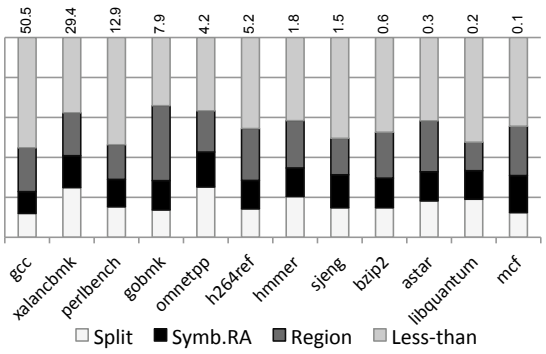


Figure 23. Time taken by static analyses (second part). **Split**: live range splitting (Sections 4.3 and 4.4). **Symb.RA**: Symbolic range analysis (Section 4.3) **Region**: Region analysis (Section 4.4). **Less-than**: Less-than test (Section 4.6).

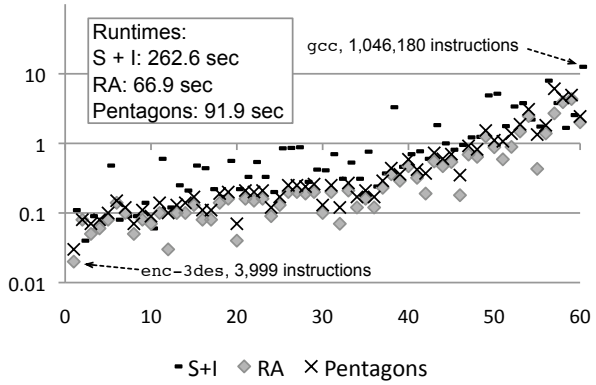


Figure 24. Runtime comparison of three non-relational analysis. **S**: symbolic range analysis, Section 4.3. **I**: integer overflow elimination, Section 4.2. **RA**: Rodrigues *et al.*'s range analysis [26]. **Pentagons** [19]. Y axis is runtime, in seconds. Each X point is a benchmark, sorted by size.

original description of Pentagons as possible. As we have mentioned in Section 2, Pentagons are the union of two abstract domains, L and I. The former encodes "less-than" relations between variables, the latter encodes ranges of integers. We use the range analysis of Rodrigues *et al.* to find I. We have used the "less-than" graph of Bodik *et al.* [6] to speed up the resolution of queries when building L. Rodrigues's algorithm leverages some precision by running on Bodik's *Extended SSA* form [6], but it does not use the extensive live range splitting techniques that we have described in section 4.1.

Effectiveness. Figure 25 shows the percentage of bound-checks that we eliminate using different techniques: ours, and Pentagons. We are more effective than pentagons on every sample of this experiment. We can remove every

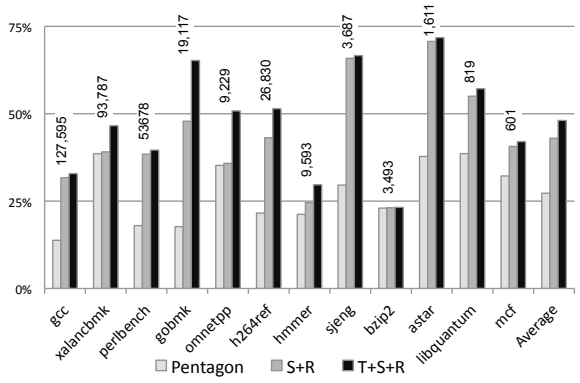


Figure 25. Percentage of bound checks eliminated. The higher, the better. **Pentagon**: Logozzo *et al* [19]. **T**: Tainted flow analysis (Section 4.5). **S**: Symbolic range analysis (Section 4.3). **R**: Region analysis (Section 4.4). *Averages*: pentagons = 27.2%, S+R = 43.0%, T+S+R=48.1%. Numbers above bars give quantity of bound-checks that AddressSanitizer inserts in each benchmark.

guard that pentagon does, but the inverse is not true. The tainted flow analysis of Section 4.5 improves our algorithm by 6.1%. In our best scenario, SPEC's astar, we have removed 73% of all guards. In our worst scenario, bzip2, we have removed only 24% of all the bound-checks.

Speedup. Figure 26 shows how much we speed up the binaries produced by AddressSanitizer with our analyses. On average, we achieve a speedup of 17%. This is less than the 48% of bound-checks that we eliminate, but these instructions account for only a small part of the entire execution cost. The compulsory cost of a guard consists of two loads, two comparisons, and two branches. By eliminating these guards, we improve also the quality of the code produced by the register allocator, as we increase the average size of basic blocks. We are still 15.2% slower than unsafe programs, i.e., programs without runtime bound-checks.

Energy consumption. The elimination of bound-checks has the beneficial effect of reducing the energy footprint of safe C programs. This energy saving is a natural consequence of run time reduction. To estimate it, we have ported some of our benchmarks to one of our system-on-a-chip devices, which is based on an Intel Cedar Trail processor, at 1.86 GHz, with 1M Cache, running Linux Fedora 14. We run LLVM, AddressSanitizer, plus all our analyses, in the device itself. We could not run the two largest benchmarks in this setup, due to the lack of storage space to fit the assembly program, in ASCII format together with its inputs. We measure energy consumption using an National Instruments USB 6009 digital acquisition device, which can perform 24K samples per second. This power meter converts analog samples to digital data, which we analyze with a C++ driver of our own craft. We can measure energy at a very

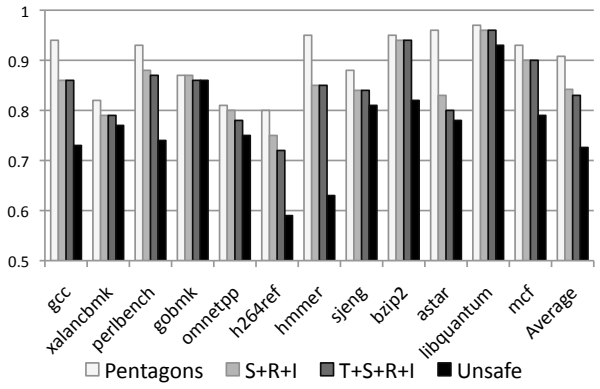


Figure 26. Run time of program after bound-check elimination, compared to run time of program with all the bound-checks. The shorter the bar, the faster the program. **Pentagon**: Logozzo *et al* [19]. **T**: Tainted flow analysis (Section 4.5). **I**: Integer overflow analysis (Section 4.2). **S**: Symbolic range analysis (Section 4.3). **R**: Region analysis (Section 4.4). *Average speedup*: pentagons = 9.1%, S+R+I = 15.8%, T+S+R+I=17.0%, Unsafe = 26.4%.

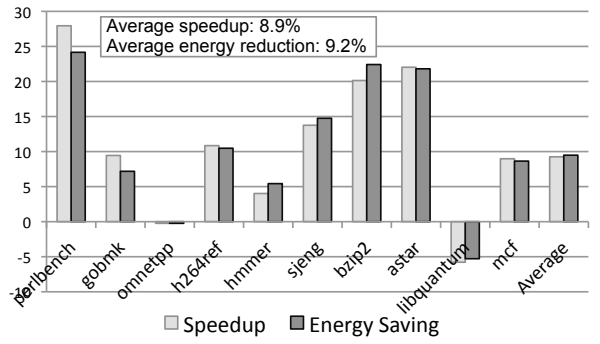


Figure 27. Performance analysis in the Cedar Trail processor. Bars show gains (in percentage) of GreenArrays over original AddressSanitizer.

fine grain, because we determine the start and end of the sampling period by firing signals through the *ready-to-send* (RTS) port of the device. We tried to emulate the methodology of Singh and Kaiser [31] as much as possible. Figure 27 shows the results of our measurements. We are reporting energy consumption for the entire device, instead of focusing on individual components. We perform each measurement nine times, and in each case, variations represent less than 1% of the smallest sample. We found no correlation between run times in the Cedar Trail, and in our Xeon. In particular, we have observed a slowdown in optimized libquantum.

Discussion: Symbolic Ranges vs Pentagons. In terms of implementation, the only difference between our approach and Pentagons are the symbolic range analysis plus the overflow checks. Both implementations use the same live range splitting strategy and the same region analysis. We are more precise than pentagons due to our ability to perform more complex "less-than" checks. We can establish more complicated less-than relations between variables because (i) our algebra is not hindered by integer overflows, (ii) we can deal with more complex polynomials than Pentagons and (iii) we have more information when performing the checks.

Concerning the first item, the original implementation of Pentagons seems to be limited due to the assumptions that integer overflows might happen. The original description of the method does not provide much details on the transfer functions associated with multiplication, for instance [19]; hence, part of this discussion is based on our understanding of that domain. In our implementation of pentagons, we can prove that $i \subseteq 4 \times i$ if, but only if, the interval range of i is known at compilation time. For example, if we know that $R(i) = [0, 4]$, then we know that $R(4 \times i) = [0, 16]$, and we include i in the less-than set of $4 \times i$. If the range of i is symbolic, then we cannot assume any less-than relation, because an integer overflow can cause $4 \times i$ to be less than i. In our approach, on the other hand, we know that overflows will not happen, for we have guarded the program against them. Thus, our symbolic manipulation library can easily infer that $i < 4 \times i$ if $R(i)_{\downarrow} > 0$.

Additionally, we can handle expressions substantially more complex than $4 \times i$. For example, we deal with quadratic equations, such as $n^2 - n > 0$. They appear, for instance, due to matrices that are treated as linear vectors, as in the loop `for(i = 0; i < n*n; i += n)`. To handle these equations, we have augmented GiNaC's original implementation with Baskhara's formula to solve quadratic equalities. According to this formula, if $ax^2 + bx + c = 0$, then $x = (-b \pm \sqrt{b^2 - 4ac})/2a$. We have also added Tartaglia's formula to our test, to solve cubic equations, but we have not found occasion to use it in our experiments. Finally, upon doing a less-than test, we have usually more information than Pentagons. In this sense, our symbolic analysis is a "lazy" implementation of Pentagons: we accumulate unsolved arithmetic expressions in the symbolic ranges of variables, and only evaluate them after we have computed all the ranges in the program. Pentagons, on the other hand, must determine less-than relationship upon interpreting abstractly the instructions that constitute the program.

6. Related Works

Most of the literature on array bound checks elimination targets Java [6, 19, 34], and not C. This lack of interest has trivial explanation: binaries produced out of C programs have no bound-checks to be eliminated. Nevertheless, there exist several efforts to protect C programs against out of bounds accesses [1, 13, 22, 29, 30]. These projects change the memory allocation library used by C programs using different approaches: splay trees [13], shadowing [29], size information indexed through tables [22, 30], validation through align-

ment constraints [1], etc. Our work is complementary to this previous research. We are not developing a safe memory allocation library; instead, we have designed a technique to conservatively eliminate access checks.

A work due to Dietz *et al.* [14] has made us aware of the danger of integer overflows. There exist techniques to prove that this kind of overflow cannot happen [26], or to prevent them from corrupting sensitive program information [7]. However, several efforts to validate memory accesses in C programs do not mention this phenomenon. Even widely used tools, such as those studied by Wilander and Kamkar, seem to be oblivious to this issue [36]. Two notable exceptions exist. The first is Cousot *et al.*'s Astrée [11], which gives the user the option to be warned about integer arithmetic overflows. Yet, Astrée analyzes a subset of C, and uses very expensive techniques, such as abstract interpretation on relational lattices. The second is Ronne *et al.*'s VBCE system [34], which removes bound checks from Java programs while these programs execute. VBCE incorporates the semantics of integer overflows into its constraint system. Nevertheless, VBCE differs from our GreenArrays in several ways. Firstly, it analyzes Java; hence, it does not need to infer array sizes. Secondly, it applies part of its analyses at run time, when some values are already known.

There exist ways to perform flow-sensitive range analyses (symbolic or not), without live range splitting, as long as the supporting data-structures keep track of the abstract state of variables. Such approach is adopted, for instance, by Ferrara *et al.* [16], and seems to be favored by the abstract interpretation community, as in the recent work of Oh *et al.* [24]. We have not implemented the framework of Oh *et al.*, but we believe that our implementation is sparser. As an example, let us consider the code `if (?) then` l_1`: a=•; else` l_2`: a=•; endif; if (?) then` l_3`: x=a; else` l_4`: x=a; endif`. In this scenario, Oh et al.'s framework creates four dependence links between $\{l_1, l_2\}$ and $\{l_3, l_4\}$. Our method, on the other hand, converts the program to SSA form; hence, creating two names for variable a.

The inspiration for our symbolic region analysis came from a work by Rugina and Rinard [27]. They have used region analysis to attack problems such as race detection and automatic parallelization. Our work is substantially different. Most fundamentally, Rugina computes the size of arrays backwardly, i.e., he draws information from array accesses, we do it forwardly, i.e., we draw information from allocation sites. In other words, from an array access such as $*a = e$, Rugina assigns to a a minimum index given by $e_\downarrow$, and a maximum index given by $e_\uparrow$. In our case, an instruction such as $a =$ `alloc`(e) leads us to assign to a the offsets $[e_\downarrow, e_\uparrow]$. Furthermore, we deal with more complex symbolic expressions, involving min, max, and products between variables, and, contrary to Rugina and Rinard, do not assume that variables involved in array dereferences are positive, as we prevent overflows from happening. Finally, Rugina and Rinard's method has been applied on small programs, and we speculate that it would be too expensive to be used in our setting, due to two reasons. First, they solve constraints using integer linear programming, which is more expensive than our algorithm; second, they have a larger number of constraints, as their analysis is dense.

The algorithms that we discuss in this paper differ from previous approaches to array bound check elimination, including Pentagons [19], ABCD [6] and MemSafe [30]. We focus on these latter two systems, as Section 5 already contains an extensive comparison with Pentagons. Our first implementation of GreenArrays was based on Bodik *et al.*'s ABCD algorithm. ABCD implements array bounds checks on demand: it runs queries for each access that must be sanitized. The ABCD approach is similar to Pentagons, because it relies on the less-than domain. However, contrary to Pentagons, ABCD does not use integer ranges to refine its results. Thus, we speculate that Pentagons are more precise, yet slower than ABCD. MemSafe is a library, together with companion optimizations, designed to safeguard C programs against invalid memory accesses. This framework contains a suite of optimizations that mitigate the impact of verifying memory accesses. Yet, these optimizations have a local scope and do not rely on abstract domains as we do.

7. Conclusion

This paper has presented a suite of static analyses that successfully eliminates several runtime checks used to guard C programs against out-of-bounds memory accesses. Our analyses have been able to speed up code instrumented by AddressSanitizer, an industrial-quality tool, substantially. Our work is built on four decades of improvements on static analyses, yet, our algorithms for symbolic range and region analysis, live range splitting, less-than comparisons and the full combination of all these theories are novel contributions of this work. The concrete result of these contributions is an effective and useful method to generate safe executables out of C programs, which is publicly available. As future work, we plan to augment our analyses with the ability to track information through pointers in the heap. This extension will let us handle arrays of arrays, for instance.

Acknowlegment: We thank Fabrice Rastello, Paul Feautrier and Damien Massé for useful discussions during the preparation of this paper. We thank the Oopsla referees for suggestions and comments that greatly improved the quality of our manuscript. The GreenArrays project is supported by the Intel Corporation (the eCoSoC project), FAPEMIG (the Feps II project) and CAPES (AEX).

Software: Our GreenArrays framework is publicly available at `https://code.google.com/p/ecosoc/`

References

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible de-

fense against out-of-bounds errors. In *SSYM*, pages 51–66. USENIX, 2009.

[2] S. Ananian. The static single information form. Master's thesis, MIT, September 1999.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[4] C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *J. Symb. Comput.*, 33(1):1–12, 2002.

[5] W. Blume and R. Eigenmann. Symbolic range propagation. In *IPPS*, pages 357–363, 1994.

[6] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

[7] D. Brumley, D. X. Song, T. cker Chiueh, R. Johnson, and H. Lin. RICH: Automatically protecting against integer-based vulnerabilities. In *NDSS*. USENIX, 2007.

[8] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55–66. ACM, 1991.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.

[11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Form. Methods Syst. Des.*, 35(3):229–264, 2009.

[12] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[13] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI*, pages 144–157. ACM, 2006.

[14] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *ICSE*, pages 760–770. IEEE, 2012.

[15] J. Ferrante, J. Ottenstein, and D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3): 319–349, 1987.

[16] P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code for .net. *SIGPLAN Not.*, 43(10):329–346, 2008.

[17] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299. ACM, 2007.

[18] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[19] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.*, 75(9):796–807, 2010.

[20] A. Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100, 2006.

[21] A. Miné. Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. *Science of Computer Programming*, 2013.

[22] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *PLDI*, pages 245–258. ACM, 2009.

[23] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2005.

[24] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for c-like languages. In *PLDI*, pages 1–11. ACM, 2012.

[25] A. A. Rimsa, M. D'Amorim, F. M. Q. Pereira, and R. Bigonha. Efficient static checker for tainted variable attacks. *Science of Computer Programming*, 80(1):91–105, 2014.

[26] R. E. Rodrigues, V. H. S. Campos, and F. M. Q. Pereira. A fast and low overhead technique to secure programs against integer overflows. In *CGO*. ACM, 2013.

[27] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *TOPLAS*, 27(2):185–235, 2005.

[28] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *S&P*, pages 1–15. IEEE, 2010.

[29] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: a fast address sanity checker. In *USENIX ATC*, pages 28–28. USENIX Association, 2012.

[30] M. S. Simpson and R. K. Barua. Memsafe: Ensuring the spatial and temporal memory safety of c at runtime. *Softw. Pract. Exper.*, 43(1):93–128, 2013.

[31] D. Singh and W. J. Kaiser. The atom LEAP platform for energy-efficient embedded computing. Technical Report 88b146bk, UCLA, 2010.

[32] A. L. C. Tavares, B. Boissinot, F. M. Q. Pereira, and F. Rastello. Parameterized construction of program representations for sparse dataflow analyses. In *Compiler Construction*, pages 2–21. Springer, 2014.

[33] F. Tip. A survey of program slicing techniques. Technical report, CWI, 1994.

[34] J. von Ronne, A. Gampe, D. Niedzielski, and K. Psarris. Safe bounds check annotations. *Concurr. Comput. : Pract. Exper.*, 21(1):41–57, 2009.

[35] M. Weiser. Program slicing. In *ICSE*, pages 439–449. IEEE, 1981.

[36] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS*, pages 1–12. USENIX, 2003.

[37] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of ssa-based optimizations for llvm. In *PLDI*, pages 175–186. ACM, 2013.