Practical and Effective Symbolic Analysis for Buffer Overflow Detection

Lian Li Cristina Cifuentes Nathan Keynes
Sun Labs, Oracle
Brisbane, Australia
{lian.li,cristina.cifuentes,nathan.keynes}@oracle.com

ABSTRACT

Although buffer overflow detection has been studied for more than 20 years, it is still the most common source of security vulnerabilities in systems code. Different approaches using symbolic analysis have been proposed to detect this vulnerability. However, existing symbolic analysis techniques are either too complex to scale to millions of lines of code (MLOC), or too simple to effectively handle loops and complex program structures.

In this paper, we present a novel symbolic analysis algorithm for buffer overflow detection that applies simple rules to solve relevant control and data dependencies. Our approach is path-sensitive and effectively handles loops and complex program structures. Scalability is achieved by using a simple symbolic value representation, filtering out irrelevant dependencies in symbolic value computation and computing symbolic values on demand.

Evaluation of our approach shows that it is both practical and effective: the analysis runs over 8.6 MLOC of the OpenSolaris^{TM} Operating system/Networking (ON) codebase in 11 minutes and finds hundreds of buffer overflows with a false positive rate of less than 10%.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging–Symbolic execution

General Terms

Reliability, Security

1. INTRODUCTION

Although buffer overflow detection has been studied for more than 20 years, it is still the most common source of security vulnerabilities in systems code. In a study of bugs that lead to publicly reported vulnerabilities, MITRE reports that 19% of all security vulnerabilities over the period 2001-2006 were due to buffer overflows [6]. Different approaches have been proposed to detect this vulnerability, including both dynamic [26, 25, 20] and static [12, 32, 10, 17] techniques.

Symbolic analysis [16] is a static analysis technique that represents the values of program variables and computations with symbolic values. Recently, different symbolic analysis approaches [29, 4, 32, 14, 10, 15, 31, 1, 2, 17, 22, 5] have been proposed to detect buffer overflows and promising results were reported. However, these approaches are still either too complex to scale to millions of lines of code (MLOC), or too simple to effectively handle loops and complex program structures.

Complex approaches that do not scale well to MLOC include [4, 1, 22, 5]. These approaches exhaustively traverse all possible execution paths and use an external constraint solver or a theorem prover to try to solve all data and control dependencies on each path. Although they can be very precise, these approaches are hard to scale to large applications due to the unbounded number of execution paths and the complexity of solving all dependencies on each path. As a result, they are mostly used in unit testing [22, 5, 23].

Simpler approaches that trade precision for scalability include: not being path-sensitive and only providing very limited support for control dependencies [29, 14], only supporting very limited computation between symbolic values [12, 32], i.e., limited support for data dependencies, or using simple heuristics to handle loops and complex control structures [32, 2]. These tradeoffs not only miss many bugs, they may also introduce numerous false alarms.

In this paper, we propose a new symbolic analysis technique for buffer overflow detection. Here buffer overflow refers to buffer bounds violation via both read and write accesses. Compared to previous approaches, we apply simple rules to solve relevant data and control dependencies iteratively. The analysis is path-sensitive and handles loops and complex program structures effectively. Scalability is achieved by using a simple symbolic value representation, filtering out irrelevant dependencies in symbolic value computation, and computing symbolic values on demand.

Our demand-driven algorithm has been implemented in Parfait [8], a scalable bug-checker built on top of LLVM [18]. As shown in Section 6, our experimental results against large systems code show that it is both practical and effective.

In summary, this paper makes the following contributions:

- We propose a new symbolic analysis technique for buffer overflow detection. The technique is simple yet effective. It can be easily implemented and integrated with other analysis techniques.

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-60558-791-2/10/11 ...$10.00.

- We develop a new demand-driven symbolic analysis algorithm to compute the symbolic value of a variable efficiently.
- We evaluate our implementation using several large applications, including the OpenSolaris^{TM} Operating system/Networking (ON) consolidation. Experimental results show that this technique is practical and effective. It analyzes 8.6 MLOC of the OpenSolaris ON codebase in 11 minutes on an Intel E8600 3.33GHz processor and identifies hundreds of buffer overflows with a false positive rate of less than 10%.

The rest of the paper is organized as follows. Section 2 reviews related work. In Section 3, we show an example to motivate our approach. The symbolic analysis technique is described in Section 4 and the algorithm is presented in Section 5. We evaluate the scalability and effectiveness of our approach in Section 6 and Section 7 concludes the paper.

2. RELATED WORK

Symbolic analysis was introduced in the '70s [16] and has recently been applied in many approaches to detect buffer overflows [29, 14, 24, 10, 32, 15, 17, 2, 22, 5]. In symbolic analysis, the program is executed with symbolic values as input. During symbolic execution, values of program variables are computed as symbolic expressions by manipulating program expressions involving symbolic values, i.e., solving *data dependencies* symbolically. Control statements such as branch instructions are executed by (conceptually) following both branches, and maintaining the control predicate information, i.e., *control dependencies*, on each branch.

Existing symbolic analysis techniques differ from each other in their different representations of symbolic values and how the symbolic values are computed. In [29, 14], the symbolic values are simplified as integer ranges and the authors map the range analysis problem into an integer linear programming (ILP) problem, which is exponential. Rugina and Rinard [24] reduce the ILP problem to a linear programming problem by using a different symbolic representation: the symbolic value of a variable is abstracted as a linear function of the set of input variables, then a linear constraint solver is used to solve the linear representation for each variable. However, for large applications with millions of LOC, it will be very time-consuming for the linear constraint solver to solve the representations of all program variables together. These approaches are not path-sensitive in that control dependencies are either not used at all or are discarded at instructions where different execution paths merge, leading to imprecision in the buffer overflows reported (i.e., high false positive rate).

Xie et al. [32] use linear derivations of a single symbol to represent symbolic values, which are then computed by interpreting the program. In their approach, computations between symbolic values are not supported and loops with non-constant iterations are handled by simply unrolling them once and terminating them with an assumption that the loop test has failed. This simplification in handling loops, although adopted in various approaches [13, 31, 2], can miss many bugs and may cause some false positives as well. In [10], the authors formalized symbolic analysis as a special case of abstract interpretation [9]. They defined a set of abstract domains to represent symbolic values at different levels of precision and the users can select which abstract domain to use. The more precise the abstract domain is, the more expensive the analysis will be. Their approach can scale to **tens of thousand** LOC as described in the paper. The technique described in this work can be integrated as a different abstract domain in their frameworks.

Compared to the above approaches, symbolic analysis as applied in unit testing [1, 22, 5] is often more precise by exhaustively traversing all possible execution paths and using an external constraint solver or theorem prover to solve all data and control dependencies on each path, where control dependencies are often represented as extra constraints on each path. Given that the number of possible execution paths is unbounded in the presence of loops, and that the execution path can be very long, restrictions are often introduced to limit the number of iterations to traverse a loop and the number of constraints on each execution path.

The authors in [17] improve the scalability of the above approaches by being demand-driven: instead of symbolically executing the program, they exhaustively traverse all possible execution paths backwards from each buffer access instruction. During the backward traversal, extra data dependencies and control dependencies are analyzed incrementally by an external constraint solver, allowing for the analysis of **hundreds of thousand** LOC. As in [1, 22, 5], loop bounds and path restrictions are applied.

In this paper, we use a different tradeoff for precision and scalability by iteratively solving relevant data dependencies and control dependencies when computing the symbolic values of a program variable. Instead of exhaustively traversing all execution paths, we only consider linearly-related control dependencies which can be efficiently solved. Since the values of array index variables often only depend on data and linearly-related control dependencies, it is very effective in finding buffer overflows. As a result, our approach is both scalable and effective as shown in our experimental results, scaling well to **millions** LOC.

3. A MOTIVATING EXAMPLE

We present the C-code fragment in Figure 1(a) to illustrate our approach. The function tosunds_str produces a new string by adding a letter CAPCHAR in front of every capital letter in the input string. It firstly creates a new buffer buf with size n (lines 5 and 6). Then, in the for loop (lines 8 - 13), every letter in the input string str is processed and copied to buf, where a letter CAPCHAR will be inserted in front of the processed letter if it is in uppercase. After the for loop, the null string terminator will be appended at the end of buf as shown in lines 14 - 16.

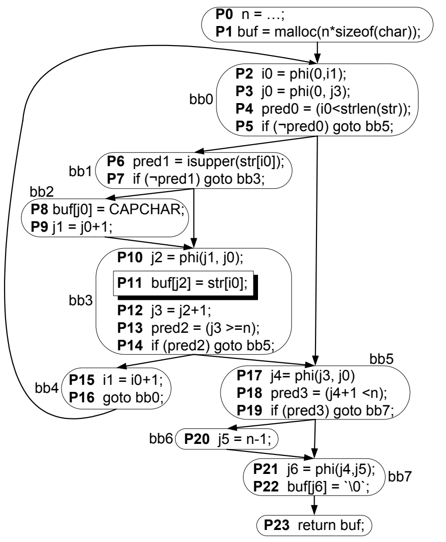
In the above example, a buffer overflow may occur at line 11 as highlighted in the shadow box. The for loop exits in line 12 if the index variable j is larger than or equal to the buffer size n. However, the loop will keep executing if j is equal to n-1. In that case, in the next iteration of the loop, the assignment to buf in line 11 may overflow.

3.1 The Intermediate Representation

Figure 1(b) shows the control flow graph (CFG) and the single static assignment (SSA) representation [11] for the program in Figure 1(a), where each instruction is given a unique label and only relevant basic blocks are shown. In SSA form, all variables have a single definition and at join points, where different paths in the CFG merge, a *phi* instruction is introduced as the new definition of a variable

```
     /* Puts CAPCHAR in front of uppercase characters
      * For example (assuming CAPCHAR = '%'):
      * if str = ABc, it returns %A%Bc */
1     char * tosunds_str(char * str) {
2         int i, j, n;
3         char * buf;
4         . . . ;
5         n = . . . ;     //n is defined by external function
6         buf = malloc(n * sizeof(char));
7         j = 0;
8         for (i = 0; i < strlen(str); i++) {
9             if (isupper(str[i]))
10                buf[j++] = CAPCHAR;
11            buf[j++] = str[i];
12            if (j >= n) break;
13        }
14        if (j+1 >= n )
15            j = n-1;
16        buf[j] = '\0';
17        return buf;
18    }
```

(a) The code snippet



```
P0  n = ...;
P1  buf = malloc(n*sizeof(char));
bb0
P2  i0 = phi(0,i1);
P3  j0 = phi(0, j3);
P4  pred0 = (i0<strlen(str));
P5  if (¬pred0) goto bb5;
bb1
P6  pred1 = isupper(str[i0]);
P7  if (¬pred1) goto bb3;
bb2
P8  buf[j0] = CAPCHAR;
P9  j1 = j0+1;
bb3
P10  j2 = phi(j1, j0);
P11  buf[j2] = str[i0];
P12  j3 = j2+1;
P13  pred2 = (j3 >=n);
P14  if (pred2) goto bb5;
bb4
P15  i1 = i0+1;
P16  goto bb0;
bb5
P17  j4= phi(j3, j0)
P18  pred3 = (j4+1 <n);
P19  if (pred3) goto bb7;
bb6
P20  j5 = n-1;
bb7
P21  j6 = phi(j4,j5);
P22  buf[j6] = `\0`;
P23  return buf;
```

(b) The intermediate representation (IR) for the example code in (a)

Figure 1: A motivating example abstracted from usr/src/cmd/fs.d/autofs/ns_ldap.c in OpenSolaris ON.

if it has been defined along distinct paths. For example, in bb3 in Figure 1(b), where the buffer overflow is located (line 11 in Figure 1(a)), a phi instruction j2 is introduced at P10 to represent the new definition of index variable j.

As is common practice in modern compilers such as GCC and LLVM, only scalar variables whose addresses are never taken are represented in SSA form. Variables which may be referenced by pointers are accessed via load and store instructions. Program values include constants and variables defined by SSA or load instructions.

For illustration purposes, we explicitly label each *control predicate* as a separate instruction at P4, P6, P13 and P18 in Figure 1(b). Control predicates are those conditional expressions used in instructions that alter the flow of control, i.e., the conditional branch instructions at P5, P7, P14 and P19. All control predicates are normalized as relational expressions in the form $pred{=}(Op_1{\sim}Op_2)$, where $pred$ is the label of the control predicate, Op_1 and Op_2 are the left and right hand operands and $\sim$ is a relational operator.

3.2 Data and Control Dependencies

In our intermediate representation, each variable is defined by a unique instruction. For a variable V, let D_V be the unique instruction where V is defined. At different instructions, the possible values of V can be different (due to control dependencies of the instruction), and we use $S_{V,P}$ to represent the symbolic value of V at instruction P. For buffer overflow detection, at each instruction P where V is used as an index variable into a buffer, we will try to compute $S_{V,P}$ to detect buffer overflows.

Both data and control dependencies need to be considered to compute $S_{V,P}$. Data dependencies refer to the incoming operands of the instruction by which V is defined. Control dependencies refer to those control predicates that need to be satisfied for P to be executed.

Consider our motivating example. To detect the buffer overflow in line 11 in Figure 1(a), we need to compute the value of index variable j2, i.e., $S_{\mathrm{j2,P11}}$. The data dependencies between variables j3, j0, j1, and j2, and the control dependency pred2 at P13 need to be considered. Note that these dependencies form a cycle and need to be solved iteratively. The control predicate pred2=(j3>=n) must be false to allow the use of j3 at P3 in bb0 to be executed. As a result, we have $S_{\mathrm{j3,P3}}$ and $S_{\mathrm{j0,P3}}$ must be less than n. Then the values of j1, j2, and j3 can be iteratively computed and the buffer overflow at P11 (line 11 in Figure 1 (a)) via index variable j2 can be identified.

The example shows that both data and control dependencies need to be considered in computing the symbolic value of a variable at an instruction. Those dependencies often form a cycle and we need to iteratively compute the values of program variables to address cyclic dependencies. However, not all dependencies are useful in the computation. For example, pred0 at P4 in bb0, pred1 at P6 in bb1 are not useful in computing $S_{\mathrm{j2,P11}}$. The key therefore is to determine which dependencies need to be considered and how the value of a program variable can be computed based on those dependencies.

4. SYMBOLIC ANALYSIS

Based on the observations in Section 3, we have developed a new technique to compute values of program variables symbolically. We use symbolic ranges to represent values of program variables at each instruction. The bounds of the symbolic ranges are defined in a simple symbolic expression domain as explained in Section 4.1.

The simple symbolic expression domain enables us to compute the symbolic ranges of program variables at each instruction very efficiently. In Section 4.2, simple rules are introduced for computing the symbolic ranges of program variables according to various control and data dependencies. The symbolic ranges of program variables can then be compared with buffer sizes to detect buffer overflows.

4.1 Symbolic Value Representation

Symbolic ranges are introduced to represent the values of program variables at each instruction. The symbolic range for variable V at instruction P, $S_{V,P}$, is defined by its lower and upper bound, denoted as $S_{V,P_{min}}$ and $S_{V,P_{max}}$, respectively. Both $S_{V,P_{min}}$ and $S_{V,P_{max}}$ are symbolic values defined in a simple symbolic expression domain as explained below.

4.1.1 A Simple Symbolic Expression Domain

The set of symbolic expressions $\mathcal{E}$ includes a set of *atomic symbols* and their affine functions as derivations. The symbolic expression set $\mathcal{E}$ has the following property:

PROPERTY 1. *Every symbolic expression E in $\mathcal{E}$ is an affine function of an atomic symbol in $\mathcal{E}$.*

Constant value 0 is regarded as an atomic symbol from which all constant values are derived. PROPERTY 1 is enforced by how symbolic expressions are introduced into $\mathcal{E}$. Initially the set is $\emptyset$. During the analysis, new atomic or derived symbolic expressions may be introduced into $\mathcal{E}$ to represent computations between symbolic expressions. A new atomic symbol will be introduced if the computation results cannot be represented as an affine function of existing atomic symbols in $\mathcal{E}$. For example, if we try to add two symbolic expressions $E_1{=}X+1$ and $E_2{=}2Y+3$, where X and Y are atomic symbols, a new atomic symbol $Z{=}X+2Y$ will be *introduced* into $\mathcal{E}$ if no existing atomic symbol is affine to Z.

This simple strategy in performing computation between symbolic expressions is very efficient and easy to implement. However, it is also very effective in finding buffer overflows as demonstrated in our experimental results in Section 6.

The symbolic expression set $\mathcal{E}$ is partially ordered by $\prec$, where for any two symbolic values E_1 and E_2 in $\mathcal{E}$, $E_1{\prec}E_2$ iff their subtraction E_1-E_2 is no larger than 0. By convention, $\top$ and $\bot$ are introduced in $\mathcal{E}$. $\forall E\in\mathcal{E}$, we have $\bot{\prec}E$ and $E{\prec}\top$. The two operations meet $\sqcap$ and join $\sqcup$ are defined as follows:

$$E_1 \sqcap E_2 = \begin{cases} E_1 & \text{if } E_1{\prec}E_2 \\ E_2 & \text{if } E_2{\prec}E_1 \\ \bot & \text{otherwise} \end{cases} \qquad E_1 \sqcup E_2 = \begin{cases} E_2 & \text{if } E_1{\prec}E_2 \\ E_1 & \text{if } E_2{\prec}E_1 \\ \top & \text{otherwise} \end{cases} \tag{1}$$

4.1.2 Symbolic Ranges

The symbolic range of variable V at instruction P, $S_{V,P} = [S_{V,P_{min}}, S_{V,P_{max}}]$, is defined by its lower bound $S_{V,P_{min}}$ and upper bound $S_{V,P_{max}}$. Both $S_{V,P_{min}}$ and $S_{V,P_{max}}$ are symbolic expressions defined in $\mathcal{E}$ and it implicitly implies that at instruction P, we have $S_{V,P_{min}}{\prec}S_{V,P_{max}}$. The symbolic range $S_{V,P}$ represents the set of all symbolic expressions E in $\mathcal{E}$ such that $S_{V,P_{min}}{\prec}E$ and $E{\prec}S_{V,P_{max}}$ may be true. The set is $\emptyset$ if no symbolic expression in $\mathcal{E}$ satisfies such condition and we use $[\top,\bot]$ to represent an empty range. The union and intersect of two symbolic ranges can be computed as follows:

$$\begin{aligned} S_1 \cup S_2 &= [S_{1_{min}} \sqcap S_{2_{min}}, S_{1_{max}} \sqcup S_{2_{max}}] \\ S_1 \cap S_2 &= [S_{1_{min}} \sqcup S_{2_{min}}, S_{1_{max}} \sqcap S_{2_{max}}] \end{aligned} \tag{2}$$

Let B be the size of the buffer and let V be the index variable, a buffer overflow will be reported if $S_{B,P_{max}}{\prec}\, S_{V,P_{max}}$ or $S_{V,P_{min}}{\prec}$ -1[1]. A buffer access cannot overflow if $S_{V,P_{max}}{\prec}\, S_{B,P_{min}}{-}1$ and $0{\prec}S_{V,P_{min}}$. Note that such buffer accesses are guaranteed to be within buffer bounds since the symbolic ranges are computed conservatively as described in Section 4.2. Otherwise, we consider the buffer access to be a *potential buffer overflow*, that is our analysis cannot determine with certainty whether or not a vulnerability exists. Potential buffer overflows are not reported as bugs.

[1]We assume that the accessed buffer is indexed from 0, as for C.

Symbolic Range Algebra $S_1 \bullet S_2$
$S_1+S_2=[S_{1min}+S_{2min}, S_{1max}+S_{2max}]$ $S_1-S_2=[S_{1min}-S_{2max}, S_{1max}-S_{2min}]$
$S\times E=[S_{min}\times E, S_{max}\times E]\cup[S_{max}\times E, S_{min}\times E]$ $S_1\times S_2=S_1\times S_{2min}\cup S_1\times S_{2max}$
$S\div E=[S_{min}\div E, S_{max}\div E]\cup[S_{max}\div E, S_{min}\div E]$ $S_1\div S_2=\begin{cases}[\bot,\top] & 0\in S_2\\ S_1\div S_{2min}\cup S_1\div S_{2max} & \text{otherwise}\end{cases}$
$S\ \%\ E=\begin{cases}[1+E\sqcap 1-E, 0] & S_{max}{\prec}0\\ [0, E-1\sqcup -1-E] & 0{\prec}S_{min}\\ [1+E\sqcap 1-E, E-1\sqcup -1-E] & \text{otherwise}\end{cases}$ $S_1\ \%\ S_2=\begin{cases}[\bot,\top] & 0\in S_2\\ S_1\ \%\ S_{2min}\cup S_1\ \%\ S_{2max} & \text{otherwise}\end{cases}$

Figure 2: Simple integer arithmetic operations between symbolic ranges. E represents a symbolic expression in $\mathcal{E}$ and $\bullet$ represents an integer arithmetic operation.

4.2 Computing Symbolic Ranges

We support common integer arithmetic computations, including $+,-,\times,\div$ and $\%$, between two symbolic ranges. The resulting range represents the set of symbolic expressions obtained by computing the operation over any two symbolic expressions selected from each input range:

$$\forall E_1 \in S_1, \forall E_2 \in S_2 \rightarrow E_1 \bullet E_2 \in S_1 \bullet S_2$$

where $\bullet$ represents an arithmetic operation.

Figure 2 shows the simple algebraic equations. The equations for $+$ and $-$ are self-explanatory. For the operation $\times$, let us first look at the simple case of applying the multiplication operator to a symbolic range S and a symbolic expression E, $S\times E$. The product could be in the range $[S_{min}\times E, S_{max}\times E]$ or $[S_{max}\times E, S_{min}\times E]$, depending on whether E is positive or negative. Then the range of $S_1\times S_2$ can be computed as shown in Figure 2. Similar equations are derived for the $\div$ operator. Note that we conservatively assume that any range divided by 0 will result in the symbolic range $[\bot,\top]$, which includes all symbolic expressions in $\mathcal{E}$. In the equation for $S\ \%\ E$, where S is a symbolic range and E is a symbolic expression, the remainder will carry the same sign as the dividend as specified in the C99 standard. The equation for applying the modulo operation to two symbolic ranges is derived in the same way.

Given the simple symbolic range algebra, $S_{V,P}$, the symbolic range for variable V at instruction P, is then computed according to data and control dependencies. Two steps are involved in computing $S_{V,P}$. Firstly we compute the symbolic range of V, S_V, according to data dependencies. Then $S_{V,P}$ is computed by refining S_V with relevant control dependencies. We call S_V the *define range* of V, and $S_{V,P}$ the *use range* of V at P. Note that variable V will have different use ranges S_{V,P_1} and S_{V,P_2} at instructions P_1 and P_2, if different control dependencies are considered.

In this section, we explain how to select the relevant dependencies and introduce the simple rules for computing symbolic ranges according to those dependencies. The computed symbolic range is conservative in that for each variable, its symbolic ranges are always a superset of its possible values. More precise symbolic ranges can be computed if more accurate dependency information is provided, either via compiler analysis or user annotation.

4.2.1 Data Dependencies

Data Dependencies	Define Ranges S_V
$V{=}Op_1 \bullet Op_2$ $V{=}phi(Op_1, Op_2)$	$S_{Op_1,D_V} \bullet S_{Op_2,D_V}$ $S_{Op_1,D_V} \cup S_{Op_2,D_V}$
$V{=}Load(A, I)$ (A is constant aggregate)	$[\ A[S_{I,D_{V_{min}}}] \ldots \sqcap \ldots A[S_{I,D_{V_{max}}}],$ $A[S_{I,D_{V_{min}}}] \ldots \sqcup \ldots A[S_{I,D_{V_{max}}}]\ ]$
$V{=}Input$	$[V.Type.min, V.Type.max]$
$V{=}Unsolved$	$[\mathcal{V}, \mathcal{V}]$

Figure 3: The rules for computing the define range of V, S_V, according to data dependencies. D_V is the instruction where V is defined, $\bullet$ represent an arithmetic instruction. $V.Type.min$ and $V.Type.max$ represent the minimal and maximal value that could be represented by the type of V, i.e., $V.Type$. The instruction $V{=}Unsolved$ represents the set of instructions that will not be handled by our analysis.

Figure 3 outlines the rules to compute the define range of variable V, S_V, according to data dependencies. For variables defined by arithmetic instructions $+, -, \times, \div$ and %, their define ranges are computed according to the symbolic range algebra in Figure 2. Similar rules with small extensions can be applied to variables defined by arithmetic shift instructions. The define range of a phi instruction is the union of the use ranges of all its operands from their corresponding incoming blocks, assuming that it can hold any value from its incoming operands.

Memory dependencies are solved only if the variable is loaded from a constant buffer. Its define range will be computed by checking the range of the index variable and all possible values in the buffer. This may sound restrictive. However, it is effective in detecting vulnerabilities such as buffer overflows, where index variables are often either scalars or loaded from constant arrays (as indirect array accesses). Finally, the define range of a variable defined by user input includes any value that can be represented by its type.

Note that we do not try to solve the data dependencies for all variables. Instead, for a variable whose define range cannot be solved ($V{=}Unsolved$), such as library call returns or bit-mask operations, we *introduce* a unique atomic symbol into the symbolic expression set $\mathcal{E}$ and the define range of the variable is defined by the introduced symbol as both its lower and upper bound.

For illustration, Figure 3 only shows the rules for computing data dependencies intra-procedurally. However, these rules can be easily extended to inter-procedural analysis by propagating the values of function actual arguments to callee functions and function return values to caller functions.

4.2.2 Control Dependencies

At instruction P, the use range $S_{V,P}$ is computed by refining the define range of V according to relevant control dependencies such that P may be executed given any value of V in $S_{V,P}$. A control dependency will be considered in computing $S_{V,P}$ if it controls both the execution of instruction P and the value of V. We conservatively define that a predicate *pred* controls the execution of instruction P if 1) *pred strictly dominates* P, i.e., every path from the entry of the CFG to P includes *pred*, and 2) P is only reachable from one successor of the conditional branch instruction that uses *pred*. Then the outcome of *pred* must be true or false depending on which successor can reach P.

Control Dependencies *pred=*	Use Ranges $S_{V,P}$
$Op_1 = Op_2$ $Op_1 \leq Op_2$ $Op_1 \geq Op_2$	$S_V \cap C_1 \times S_{Op_2,pred} + C_2$ $S_V \cap [\perp, C_1 \times S_{Op_2,pred_{max}} + C_2]$ $S_V \cap [C_1 \times S_{Op_2,pred_{min}} + C_2, \top]$
$Op_1 \neq Op_2$	$\begin{cases} S_V \setminus \{S_{V_{min}}\} & V{=}S_{V_{min}} \rightarrow Op_1{=}Op_2 \\ S_V \setminus \{S_{V_{max}}\} & V{=}S_{V_{max}} \rightarrow Op_1{=}Op_2 \\ S_V & otherwise \end{cases}$

Figure 4: The rules for computing $S_{V,P}$ according to direct control dependencies. Control predicates are in the form of $pred{=}(Op_1 \sim Op_2)$, where Op_1 and Op_2 are the left and right hand operands and $\sim$ is a relational operator. The predicates are normalized in such a way that $V = C_1 \times Op_1 + C_2$ and $C_1 > 0$.

Control predicates are in the form of $pred{=}(Op_1{\sim}Op_2)$, where $\sim$ is a relational operator. A control predicate may control the value of V directly or indirectly, depending on how its operands relate to V. Op_1 is *directly related* to V if it is defined as an affine function of V, i.e., $Op_1 = C_1 \times V + C_2$ where both C_1 and C_2 are constants, or if V is defined as an affine function of Op_1. Op_1 is *indirectly related* to V if Op_1 is loaded from a buffer with the index variable directly related to V, or if V is loaded from a buffer with the index variable directly related to Op_1. Accordingly, those dependencies are called *direct* or *indirect* control dependencies. Similarly to memory data dependencies, indirect control dependencies can be transformed to direct control dependencies by examining the content of the buffer.

Figure 4 shows how to compute $S_{V,P}$ according to direct control dependencies. All control predicates are normalized in such a way that Op_1 is directly related to V. The define range S_V is intersected with a range defined by the affine function between Op_1 and V, the relational operator $\sim$ and the use range $S_{Op_2,pred}$. If multiple control predicates need to be considered in computing $S_{V,P}$, then S_V will be refined multiple times according to the rules described in Figure 4.

4.3 Enhancements

Our analysis can be improved if more detailed dependency information is provided, either via compiler analysis or user annotation. Here we introduce two important compiler analysis enhancements.

4.3.1 Loop Induction Variable Analysis

Loop induction variables refer to those variables whose values take on a simple progression in successive iterations of a loop. Loop induction variable analysis [3] tries to represent such variables as functions to the loop iteration number, thus dependencies between induction variables and loop iteration numbers can be considered when computing their symbolic ranges. This analysis is important for buffer overflow detections since induction variables commonly appear as loop indices and in subscript computations. In our analysis, we only consider induction variables which are affine functions of the loop iteration number.

4.3.2 Path Sensitive Analysis

The analysis so far is predicate-aware but it is not fully path sensitive. We have considered the control dependencies of a variable's uses, but have not considered the control dependencies of its definitions. At a phi instruction, where values defined along different paths merge, path-sensitive analysis will check from which paths a definition will reach the phi instruction. Then the define and use ranges of a phi instruction can be more precisely computed by also considering the control dependencies along the paths of its incoming definitions.

We compute gated single assignment (GSA) [21] form for path-sensitive analysis as needed. In GSA form, the phi instruction V=*phi*$(V1, V2)$ is extended to a binary decision tree called a γ tree. The incoming definitions $V1$ and $V2$ are leaf nodes in the γ tree with predicate nodes as internal nodes in the tree to guide which definition to choose. When computing the symbolic range for a phi instruction, we recursively traverse the γ tree and compute the use ranges of its incoming operands according to the control dependencies of the phi instruction as well as those control predicates in the γ tree. By only computing the GSA form on demand for relevant phi instructions, the runtime overhead is minimized.

5. THE DEMAND-DRIVEN ALGORITHM

Algorithm 1 Compute the use range $S_{V,P}$

```
1: procedure ComputeUseRange($V, P$)
2:     ComputeDefRange($V$)
3:     $S_{V,P}$ :=RefineDefRange($V, P$)
4: end procedure
```

As explained in Section 4, the use range $S_{V,P}$ is determined by firstly computing the define range S_V according to data dependencies, then refining it with relevant control dependencies. Algorithm 1 outlines the procedure, where we first call ComputeDefRange to compute S_V then refine it in RefineDefRange.

ComputeUseRange works in a demand-driven fashion. $S_{V,P}$ is computed only when ComputeUseRange is invoked with variable V and instruction P as the two input arguments. We use a data structure to cache all define ranges that have been computed. At instruction P, the use range $S_{V,P}$ can then be efficiently computed on the fly by refining the cached define range of V with relevant control dependencies.

When computing the define range S_V, for all operands that define V, we need to compute their use ranges at D_V as shown in Figure 3. Similarly, when refining S_V according to the predicate *pred*=$(Op_1 \sim Op_2)$, we need to compute the use range of Op_2 at *pred*, $S_{Op_2,pred}$, as shown in Figure 4. If those dependencies form a cycle, our algorithm will recognize those cyclic dependencies and will iteratively solve them.

5.1 Iteratively Solving Data Dependencies

The data structure $SymTab$ is introduced to cache the computed define range S_V for each variable V. Initially it is set to $\emptyset$. When S_V is to be computed, S_V and all define

Algorithm 2 Compute the define range S_V

```
    $SymTab := \emptyset$
    $NewValSet := \emptyset$
 1: procedure ComputeDefRange($V$)
 2:     if $V$ has an entry in $SymTab$ then return
 3:     end if
 4:     $NewValSet := NewValSet \cup \{V\}$
 5:     $SymTab[V] := [\bot, \top]$
 6:     for each operand $Op_i$ used to compute $V$ do
 7:         ComputeUseRange($Op_i, D_V$)
 8:     end for
 9:     Compute $S_V$ according to Figure 3
10:     $SymTab[V] := S_V$
11:     UpdateDefRange($V$)
12:     $NewValSet := NewValSet \setminus \{V\}$
13: end procedure
14: procedure UpdateDefRange($V$)
15:     for $W \in NewValSet$ do
16:         if $S_W$ is dependent on $S_V$ then
17:             for each operand $Op_i$ to compute $S_W$ do
18:                 ComputeUseRange($Op_i, D_W$)
19:             end for
20:             Compute $S_W$ according to Figure 3
21:             $SymTab[W] := SymTab[W] \cap S_W$
22:             if $SymTab[W]$ has been updated then
23:                 UpdateDefRange($W$)
24:             end if
25:         end if
26:     end for
27: end procedure
```

ranges required to compute S_V (if not computed yet), denoted as S_W, will be computed on demand and inserted into $SymTab$. Note that S_W may be used directly in computing S_V as data dependencies, or it could be used as control dependencies to refine a define range that will be used in computing S_V. There are cyclic dependencies if S_V is required to compute S_W.

Let $NewValSet$ be the set of variables whose define ranges are inserted into $SymTab$ when computing the define range S_V for variable V. We have the following property:

Property 2. *Cyclic dependencies can only exist between V and variables in $NewValSet$.*

If a define range is already cached in $SymTab$, then S_V is not required to compute it. Similarly, if a variable has no entry in $SymTab$ after S_V has been computed, then its define range is not required to compute S_V. In either case, there is no cyclic dependency in between.

In algorithm 2, the procedure ComputeDefRange returns if S_V has already been computed (lines 2 and 3). Otherwise, ComputeUseRange will be recursively invoked to compute the required use ranges as data dependencies (lines 6 - 8). The define range S_V can then be computed and cached in $SymTab$ (lines 9 and 10). In line 11, we invoke UpdateDefRange to iteratively solve cyclic dependencies (if there are any) until a fixed point is reached.

In UpdateDefRange, we recompute S_W if it is dependent on S_V (lines 16 - 21). Recall that S_V may be used directly in computing S_W as data dependencies, or it could be used as control dependencies to refine a range that is used directly in computing S_W. If S_W has been updated, then UpdateDefRange is recursively invoked to update those symbolic ranges dependent on S_W (lines 22 - 24). The algorithm terminates when no more updates can be made to $SymTab$. At this point, we have reached a fixed point for S_V and all define ranges required to compute S_V. These ranges will be cached in *SymTab* and they do not need to be recomputed thereafter. Since we only update *SymTab* if a smaller symbolic range can be computed, the algorithm is guaranteed to terminate.

5.2 Computing Control Dependencies

Algorithm 3 Refine the define range S_V at program point P

```
1: procedure RefineDefRange($V, P$)
2:     $S_{V,P} := SymTab[V]$
3:     Let $PredSet$ be the set of predicates that controls
           both the execution of $P$ and value of $V$
4:     for every $pred$=($Op_1$~$Op_2$) in $PredSet$ do
5:         ComputeUseRange($Op_2$, $pred$)
6:         Refine $S_{V,P}$ according to Figure 4
7:     end for
8:     return $S_{V,P}$
9: end procedure
```

The function RefineDefRange (Algorithm 3) computes the use range $S_{V,P}$ by refining the define range S_V according to control dependencies. As stated in Section 4, a control predicate will be considered only if it controls both the execution of instruction P and the value of V (line 3).

The use range $S_{V,P}$ will be computed by refining the define range S_V with all predicates in *PredSet* (lines 4 - 7). In our implementation, when refining a define range with the range defined by a control dependency, a simple rule is applied if the bounds of the two symbolic ranges are not comparable. The constant bound is selected first. Otherwise, if the define range is unsolvable, the bound of the range defined by the control dependency will be used. When refining the range according to a predicate *pred*=(Op_1~Op_2), ComputeUseRange will be recursively invoked to compute the use range $S_{Op_2,pred}$ (line 5). The define range S_{Op_2}, if not cached yet, will be computed on demand and inserted into *SymTab*. If S_V is used in computing S_{Op_2}, those cyclic dependencies are iteratively solved as shown in UpdateDefRange.

The control predicate *pred* always strictly dominates instruction P by definition and only those control predicates that strictly dominate *pred* will be used in computing $S_{Op_2,pred}$. As a result, the algorithm is guaranteed to progress.

5.3 The Motivating Example

Figure 5 shows the relevant dependencies for our motivating example in Figure 1 as well as how they are solved in ComputeUseRange. In our algorithm, data dependencies are cached in *SymTab* while control dependencies are solved on the fly. As shown in Figure 5, data dependencies are grouped into gray boxes. Each box is given a number, representing the order of when they are computed.

The computation order is essentially a topological order in the reduced dependency graph (including only relevant dependencies) with all cyclic dependencies being reduced into a single node. The define range of n will be solved first since it is not dependent on any other values. The define range of j0, j1, j2 and j3 form a cycle which is control dependent on n. They will be iteratively solved together. Then the define range of j4 and j5 are computed one by one. Note that this order is implicitly enforced in our algorithm by recursively calling ComputeUseRange to compute all dependencies on demand.

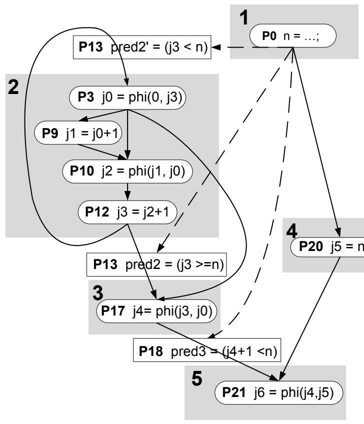


Figure 5: Reduced dependency graph (including only relevant dependencies) for the motivating example in Figure 1. All data dependencies are highlighted in oval boxes and control dependencies are placed in white rectangular boxes on the path from the definition of a variable to the instruction where it is used. Dashed lines denote that the dependency is used in computing control dependencies. Solid lines denote data dependencies.

6. EXPERIMENTAL RESULTS

We implemented our algorithm for buffer overflow detection in Parfait [8], a scalable bug checker built on top of LLVM. Parfait processes an application in two steps:

1. Source C/C++ files are parsed by the LLVM front-end, which generates LLVM bitcode files. The LLVM linker is used to link bitcode files, as any normal linker would do.

2. Bitcode files are loaded into memory, a few simple optimizations are applied on the code (e.g., constant propagation), and our analysis is applied to the bitcode representation to find buffer overflows.

The results reported in this paper use LLVM bitcode files as the starting point. The runtime includes the time in loading bitcode files to memory, performing simple optimizations, as well as the time in performing our analysis.

Benchmark	NC-LOC (lines of code)			Bitcode
	C	C++	Total	files
Asterisk 1.6.0.3	292K	6K	298K	25MB
OpenJDK^{TM} 7 b51	406K	492K	898K	2.39GB
OpenBSD 4.4	1.72M	0	1.72M	129MB
OpenSolaris ON b105	8.5M	87K	8.6M	1.44GB

Table 1: Summary of the benchmark data.

Several large open source system applications are used in our experiment as shown in Table 1. OpenSolaris ON and OpenBSD (kernel only) are general-purpose operating systems, OpenJDK is an implementation of the Java^{TM} virtual machine and Asterisk [28] is the implementation of a telephone private branch exchange (PBX). For each benchmark, we list its version or build number, the number of non commented lines of C/C++ code as reported by the SLOCCount [30] tool, and the size of the bitcode files generated by the LLVM front-end.

We evaluate our implementation of the symbolic analysis algorithm in the following ways: Section 6.1 evaluates the precision of our algorithm, Section 6.2 reports the performance of the algorithm, Section 6.3 reports on our attempts to compare the precision of the analysis against other bug checking tools, and Section 6.4 studies the limitations of our analysis by reporting the percentage of buffer accesses that cannot be solved by our analysis.

6.1 Precision

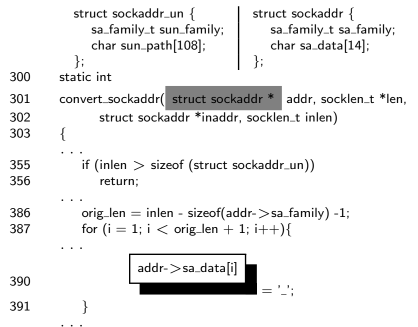
We manually verified all bugs reported by our tool, computing the true positive (i.e., bugs that are correctly reported) and the false positive (i.e., bugs that are incorrectly reported) rates. All the true positives were reported to the maintainers of each application, who further verified our reports and in turn filed bugs into their bug tracking system. Some bugs have already been fixed while others are in the process of being fixed.

Benchmarks	#Reports	#TP	FP (%)
Asterisk 1.6.0.3	35	30	14.3%
OpenJDK 7 b51	9	8	11.1%
OpenBSD 4.4	38	23	39.5%
OpenSolaris ON b105	378	351	7.14%
Total	460	412	10.4%

Table 2: Precision results of our tool. #Reports is the number of total reports, #TP is the number of correctly reported bugs and FP (%) is the false positive rate.

Table 2 summarizes the precision of the symbolic analysis implementation. Overall, our tool is able to find 412 true bugs with a false positive rate of 10.4%. The false positive rate is less than or close to 10% for all applications except OpenBSD, where we have observed a false positive rate of close to 40%. The false positives reported in OpenBSD, as well as in other benchmarks, are due to the limitation of our implementation in handling pointer types and memory dependencies, which are explained next. These limitations are currently being addressed in our implementation.

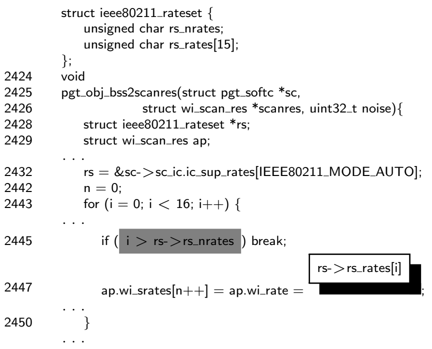
False Positives



```
            struct sockaddr_un {                | struct sockaddr {
                sa_family_t sun_family;         |     sa_family_t sa_family;
                char sun_path[108];             |     char sa_data[14];
            };                                  | };
300     static int
301     convert_sockaddr( struct sockaddr * addr, socklen_t *len,
302                 struct sockaddr *inaddr, socklen_t inlen)
303     {
        . . .
355         if (inlen > sizeof (struct sockaddr_un))
356             return;
        . . .
386         orig_len = inlen - sizeof(addr->sa_family) -1;
387         for (i = 1; i < orig_len + 1; i++){
        . . .
390                 addr->sa_data[i] = '_';
391         }
        . . .
```

Figure 6: False positive example from OpenSolaris ON, file usr/src/lib/brand/lx/lx_brand/common/socket.c

In Figure 6, the buffer access addr->sa_data[i] in line 390 is reported as buffer overflow as highlighted in the shadow box. The upper bound of the loop index i and orig_len can be computed as 107, while the buffer size for addr->sa_data is only 14. However, the type sockaddr is effectively a polymorphic type. In this case, the object that addr points to is actually of type sockaddr_un and the buffer access to addr->sa_data cannot overflow. To recognize such false positives, we need very precise points-to information. Such information is not provided in the LLVM infrastructure. Alternatively, we can rely on user annotations as in [12, 15, 31]. For this example, a simple cast to struct sockaddr_un would be the obvious annotation, and would improve the code readability. Many of the false positives in OpenBSD were reported due to this reason[2].



```
            struct ieee80211_rateset {
                unsigned char rs_nrates;
                unsigned char rs_rates[15];
            };
2424        void
2425        pgt_obj_bss2scanres(struct pgt_softc *sc,
2426                        struct wi_scan_res *scanres, uint32_t noise){
2428            struct ieee80211_rateset *rs;
2429            struct wi_scan_res ap;
            . . .
2432            rs = &sc->sc_ic.ic_sup_rates[IEEE80211_MODE_AUTO];
2442            n = 0;
2443            for (i = 0; i < 16; i++) {
            . . .
2445                if ( i > rs->rs_nrates ) break;
2447                ap.wi_srates[n++] = ap.wi_rate = rs->rs_rates[i];
            . . .
2450            }
            . . .
```

Figure 7: False positive example from OpenBSD, file usr/src/sys/arch/dev/ic/pgt.c

In Figure 7, the buffer access rs->rs_rates[i] in line 2447 is falsely reported as a bug. The range of index variable i is computed as [0,15] and the buffer size of rs->rs_rates is 15. When computing symbolic range of i, the control dependency (i>rs->rs_nrates) is not considered since the value rs->rs_nrates cannot be computed. In this case, rs->rs_nrates is at most 8 and the reported buffer overflow will not be exposed. Similar to the example in Figure 6, precise points-to information is required to be able to compute these memory dependencies. Note that although it is classified as a false positive, it can be argued that the loop exit condition should be i<15 instead of i<16.

From our experience in verifying results against the various system applications, the majority of false positives were reported due to the above two cases. Usually, these false positives require very precise points-to information to eliminate them.

6.2 Performance

Benchmark	Time (min:sec)	#Max Variables	#Max Predicates
Asterisk 1.6.0.3	0:28	18	4
OpenJDK 7 b51	9:10	26	3
OpenBSD 4.4	1:18	34	10
OpenSolaris ON b105	10:37	80	77

Table 3: Performance results of our tool. #Max Variables is the number of variables in the largest cycle that is iteratively solved. #Max Predicates is the largest number of predicates used when computing the use range of a variable on the fly.

Table 3 summarizes the analysis times for different benchmarks on an Intel E8600 3.33GHz processor with 8GB of

[2]If we consider these type violations as true bugs, the false positive rate for OpenBSD would be 21%.

RAM. For each benchmark, we report its analysis time (Column 2), the number of variables in the largest cycle that is iteratively solved (Column 3) and the largest number of control predicates used when computing the use range of a variable on the fly (Column 4). The analysis time includes the time to load the bitcode files, build the representation in memory, perform the symbolic analysis, and report the results. For each benchmark, we ran our analysis 10 times and the user time of the slowest run is reported.

The symbolic analysis time is less than 11 minutes for OpenSolaris ON with 8.6 MLOC. OpenJDK is the second most time-consuming application, despite the fact that it is much smaller than OpenBSD (see Table 1). This is due to its extensive use of C++ templates, which are expanded in the LLVM bitcode files. As a result, the total size of all LLVM bitcode files in OpenJDK is actually much larger than that in OpenBSD (see Table 1).

Overall, the performance of the algorithm is largely due to the fact that we compute only relevant dependencies on demand, and use simple symbolic expression representations. As shown in Column 3 and Column 4 in Table 3, the largest cycle that is iteratively solved in COMPUTEDEFRANGE includes at most 80 variables and the largest number of control dependencies that is computed on the fly in REFINEDEFRANGE is 77.

6.3 Comparison with Other Tools

Since we do not have access to existing commercial tools, only open source tools were considered for comparison. Of the tools available, four of them (LLVM/Clang Static Analyzer [19], Saturn [31], Uno [27] and Splint [12]) are under active development and therefore used in our testing. The LLVM/Clang Static Analyzer mainly reports bugs that deal with Objective C errors, as such, it does not report any buffer overflow bugs in C code. Saturn is a system for static analysis for C programs. Its existing analysis assumes in-bound array accesses, as such, Saturn does not report buffer overflow bugs at present.

We tried to run Splint and Uno over the applications in Table 1. In our experience, Splint produces a large number of error reports and Uno produces very few reports. For Asterisk alone, Splint generates around 800 out-of-bounds errors and Uno reports 8 which look to be false positives. We estimate that to verify all the reports against the given applications in Table 1 would require many months of dedicated effort, even at a rate of 100 per day.

	#Reports	#TP	#Misses	FP (%)	FN (%)
Splint	1113	592	647	46.8%	52.2%
Uno	462	454	785	1.7%	63.4%
Parfait	914	914	325	0%	26.2%

Table 4: Precision results of different tools on the BegBunch suite. #Reports is the number of total reports, #TP is the number of correctly reported bugs, #Misses is the number of bugs present in the code but not reported. FP (%) is the false positive rate and FN (%) is the false negative rate.

Therefore, we evaluated our buffer overflow detection implementation against other bug checking tools for C/C++ using the BegBunch benchmarking suite [7]. The BegBunch suite includes a set of bug kernels extracted from existing open source applications and existing bug checking benchmarks. Each bug kernel has been manually inspected by the authors of [7] and the bug location is marked. It allows us to evaluate both precision and performance of our tool, as well as checking how many bugs are missed (i.e., false negatives).

Table 4 summarizes the results of different tools on the BegBunch suite. For the total 1239 buffer overflow bug kernels in the BegBunch suite, our tool reported 914 bugs, all of which were correctly reported with an overall 26.2% false negative rate. For this dataset, our tool is significantly better than Splint and Uno. It is able to find all true bugs that are reported by the other two tools, with a much better false positive and false negative rates. The bugs that were missed by our tool are mostly due to our limitation in handling memory dependencies, which require precise points-to analysis or user annotations. It is worthwhile noting that Splint's false positive and false negative rates can also be improved by adding annotations to the code to be analyzed.

6.4 Limitations

Recall that in our analysis, we only report a bug if $S_{B,P_{max}} \prec S_{V,P_{max}}$ or $S_{V,P_{min}} \prec$ -1, where V is the index variable and B is the size of the accessed buffer. In addition, a buffer access is guaranteed to be within buffer bounds only if $S_{V,P_{max}} \prec S_{B,P_{min}} - 1$ and $0 \prec S_{V,P_{min}}$. Otherwise the buffer access is a *potential buffer overflow* that cannot be solved by our analysis. In this section, we report on the percentage of potential buffer overflows, which suggests the limitations of our approach and gives us a hint about how many buffer overflows may be missed.

Benchmark	Potential buffer overflows (%)
Asterisk 1.6.0.3	25.1%
OpenJDK 7 b51	42.9%
OpenBSD 4.4	27.0%
OpenSolaris ON b105	31.1%

Table 5: The percentage of potential buffer overflows that cannot be solved by our analysis.

Table 5 summarizes the percentage of potential buffer overflows for all applications. For Asterisk, OpenBSD and OpenSolaris ON, close or below 30% of all buffer accesses were potential buffer overflows that could not be solved by our analysis. The percentage increased to just over 40% for OpenJDK. We manually looked into some of the potential buffer overflows and found that most of them were not solved due to the same limitation of our implementation in handling pointer types and memory dependencies. This also suggests the worst behavior in OpenJDK, where precise points-to information is harder to get because of its extensive usage of C++ templates.

7. CONCLUSION

In this paper, we presented a new symbolic analysis technique for buffer overflow detection and evaluated it using large systems applications. We demonstrated that the simple symbolic value representation is effective for buffer overflow detection and symbolic values could be precisely computed by iteratively solving data dependencies and linearly-related control dependencies together. We also showed that by being demand-driven and using simple algebraic rules, the symbolic values could be computed very efficiently.

Our experimental results against large systems applications suggested that this technique is simple yet effective –

it was easy to implement and found hundreds of buffer overflows in large, well-tested codebases with a false positive rate of around 10%. This makes the technique feasible for implementation in existing compilers, to identify vulnerabilities at the earliest stage of software development.

In addition to buffer overflows, our analysis can be applied to detect other important vulnerabilities such as integer overflows. It can also be applied to analyze the bounds of shared memory regions to detect potential data races.

8. REFERENCES

[1] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java pathfinder. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 134–138, Braga, Portugal, March 2007.

[2] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering*, pages 211–220. ACM Press, 2008.

[3] J. Birch, R. van Engelen, K. Gallivan, and Y. Shou. An empirical evaluation of chains of recurrences for array dependence testing. In *Proceedings of the 15th international conference on Parallel Architectures and Compilation Techniques*, pages 295–304. ACM Press, 2006.

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30:775–802, 2000.

[5] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th symposium on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[6] S. Christey and R. A. Martin. Vulnerability type distributions in CVE. Technical report, The MITRE Corporation, May 2007. Version 1.1.

[7] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz. BegBunch: Benchmarking for C bug detection tools. In *Proceedings of the 2009 international workshop on Defects in large software systems*, July 2009.

[8] C. Cifuentes and B. Scholz. Parfait – designing a scalable bug checker. In *Proceedings of the ACM SIGPLAN Static Analysis Workshop*, pages 4–11, 12 June 2008.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[10] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In *Proceedings of the 2005 European Symposium on Programming*, pages 21–30. Springer, April 2005.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[12] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, pages 42–51, January/February 2002.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.

[14] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th conference on Computer and Communications Security*, pages 345–354. ACM Press, 2003.

[15] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *Proceeding of the 28th International Conference on Software Engineering*, pages 232–241. ACM Press, 2006.

[16] J. C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7):385–394, 1976.

[17] W. Le and M. L. Soffa. Marple: a demand-driven path-sensitive buffer overflow detector. In *Proceedings of the 16th international symposium on Foundations of Software Engineering*, pages 272–282. ACM Press, 2008.

[18] LLVM. Low Level Virtual Machine. `http://www.llvm.org`, detail on website.

[19] LLVM/Clang Static Analyzer. `http://clang.llvm.org/StaticAnalysis.html`. Last accessed: 1 December 2008.

[20] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 conference on Programming Language Design and Implementation*, pages 245–258. ACM Press, 2009.

[21] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the 1990 conference on Programming Language Design and Implementation*, pages 257–271. ACM Press, 1990.

[22] M. Y. Patrice Godefroid and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the 11th Network and Distributed System Security Symposium*, pages 159–169, 2008.

[23] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 15–26. ACM Press, 2008.

[24] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the 2000 conference on Programming Language Design and Implementation*, pages 182–195. ACM Press, 2000.

[25] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Network and Distributed System Security Symposium*, pages 159–169, 2004.

[26] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2. USENIX Association, 2005.

[27] Uno Tool Synopsis. `http://spinroot.com/uno/`. Last accessed: 26 October 2007.

[28] J. Van Meggelen, J. Smith, and L. Madsen. *Asterisk: the future of telephony, 2nd edition.* O'Reilly, 2007.

[29] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Network and Distributed System Security Symposium*, pages 3–17, 2000.

[30] D. A. Wheeler. SLOC Count User Guide. `http://www.dwheeler.com/sloccount/`. Last accessed: 16 March 2009.

[31] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 29(3):16, 2007.

[32] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 11th international symposium on Foundations of Software Engineering*, pages 327–336. ACM Press, 2003.