Automatic Removal of Buffer Overflow Vulnerabilities in C/C++ Programs

Sun Ding[1], Hee Beng Kuan Tan[1] and Hongyu Zhang[2]
[1]*School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore, Singapore*
[2]*School of Software, Tsinghua University, Beijing, China*

Keywords: Buffer Overflow, Static Analysis, Automatic Bug Fixing, Security Vulnerability.

Abstract: Buffer overflow vulnerability is one of the commonly found significant security vulnerabilities. This vulnerability may occur if a program does not sufficiently prevent input from exceeding intended size or accessing unintended memory locations. Researchers have put effort in different directions to address this vulnerability, including creating a run-time defence mechanism, proposing effective detection methods or automatically modifying the original program to remove the vulnerabilities. These techniques share many commonalities and also have differences. In this paper, we characterize buffer overflow vulnerability in the form of four patterns and propose ABOR--a framework that integrates, extends and generalizes existing techniques to remove buffer overflow vulnerability more effectively and accurately. ABOR only patches identified code segments; thus it is an optimized solution that can eliminate buffer overflows while keeping a minimum runtime overhead. We have implemented the proposed approach and evaluated it through experiments on a set of benchmarks and three industrial C/C++ applications. The experiment result proves ABOR's effectiveness in practice.

1 INTRODUCTION

Buffer overflow in C/C++ is still ranked as one of the major security vulnerabilities (Abstract syntax tree, 2014), though it has been 20 years since this vulnerability was first exploited. This problem has never been fully resolved and has caused enormous losses due to information leakage or customer dissatisfaction (US-CERT, 2014).

A number of approaches have been proposed to mitigate the threats of buffer overflow attacks. Existing approaches and tools focus mainly on three directions (Younan et al., 2012):

1) Prevent buffer overflow attacks by creating a run-time environment, like a sandbox, so that taint input could not directly affect certain key memory locations;
2) Detect buffer overflows in programs by applying program analysis techniques to analyze source code;
3) Transform the original program by adding additional verification code or external annotations.

For approaches in the first direction, as modern programs are becoming more complex, it is difficult to develop a universal run-time defense (Younan et al., 2012). For approaches in the second direction, even if buffer overflow vulnerabilities are detected, the vulnerable programs are still being used until new patches are released. For the third direction, though it is well-motivated to add extra validation to guard critical variables and operations, the existing approaches will add considerable runtime overhead. For example, a recent novel approach that adds extra bounds checking for every pointer may increase the runtime overhead by 67% on average (Nagarakatte et al., 2009).

We noticed that though none of the existing methods can resolve the problem fully, they share many commonalities and also have differences. In this paper, we first integrate existing methods and characterize buffer overflow vulnerability in the form of four patterns. We then propose a framework—ABOR that combines detection and removal techniques together to improve the state-of-the-art. ABOR iteratively detects and removes buffer overflows in a path-sensitive manner, until all the detected vulnerabilities are eliminated. Unlike the related methods (Nagarakatte et al., 2009, Criswell et al., 2007, Dhurjati and Adve, 2006, Hafiz and

Ding S., Tan H. and Zhang H..
Automatic Removal of Buffer Overflow Vulnerabilities in C/C++ Programs.
DOI: 10.5220/0004888000490059
In *Proceedings of the 16th International Conference on Enterprise Information Systems* (ICEIS-2014), pages 49-59
ISBN: 978-989-758-028-4

Johnson, 2009), ABOR only patches identified code segments in a path-sensitive way; thus it can eliminate buffer overflows while keeping a minimum runtime overhead.

We have evaluated the proposed approach on a set of benchmarks and three industrial C/C++ applications. The results show that the proposed approach is effective. First it can remove all the detected buffer overflow vulnerabilities in the studied subjects. Second we also compare ABOR with methods that focus on buffer overflow removal. On average, it removes 58.28% more vulnerabilities than methods that apply a straight-forward “search & replace” strategy; it inserted 72.06% fewer predicates than a customized bounds checker.

The contribution of the paper is as following:

1) The proposed approach integrates and extends existing techniques to remove buffer overflows automatically in a path-sensitive manner.
2) The proposed approach guarantees a high removal accuracy while could keep a low runtime overhead.
3) The proposed approach contains an exhaustive lookup table that covers most of the common buffer overflow vulnerabilities.

The paper is organized as follows. Section 2 provides background on buffer overflows vulnerability. Section 3 covers the proposed approach that detects buffer overflows and removes detected vulnerability automatically. Section 4 evaluates the proposed approach and section 5 reviews the related techniques that mitigate buffer overflow attacks. Section 6 concludes the paper.

2 BACKGROUND

Buffer overflow based attacks usually share a lot in common: they occur anytime when a program fails to prevent input from exceeding intended buffer size(s) and accessing critical memory locations. The attacker usually starts with the following attempts (US-CERT, 2014; Vallentin, 2007): they first exploit a memory location in the code segment that stores operations accessing memory without proper boundary protection. For example, a piece of code allows writing arbitrary length of user input to memory. Then they locate a desired memory location in data segment that stores (a) an important local variable or (b) an address that is about to be loaded into the CPU’s Extended instruction Counter (EIP register).

Attackers attempt to calculate the distance between the above two memory locations. Once such locations and distance are discovered, attackers construct a piece of data of length (*x1+x2*). The first *x1* bytes of data can be any characters and is used to fill in the gap between the exploited location and the desired location. The second *x2* bytes of data is the attacking code which could be (a) an operation overwriting a local variable, (b) a piece of shell script hijacking the system or (c) a handle redirecting to a malicious procedure.

Therefore, to prevent buffer overflow attack, it is necessary to ensure buffer writing operations are accessible only after proper validations.

3 THE PROPOSED APPROACH

In this paper, we propose Automatic Buffer Overflow Repairing (ABOR), a framework which integrates and extends existing techniques to resolve the buffer overflow vulnerabilities in a given program automatically. Figure 1 demonstrates the overall structure of the framework. ABOR consists of two modules: vulnerability detection and vulnerability removal. ABOR works in an iterative way: once the vulnerability detection module captures a vulnerable code segment, the segment is fed to the removal module; the fixed segment will be patched back to the original program. ABOR repeats the above procedure until the program is buffer-overflow-free. In this section, we introduce the two modules of ABOR in detail.

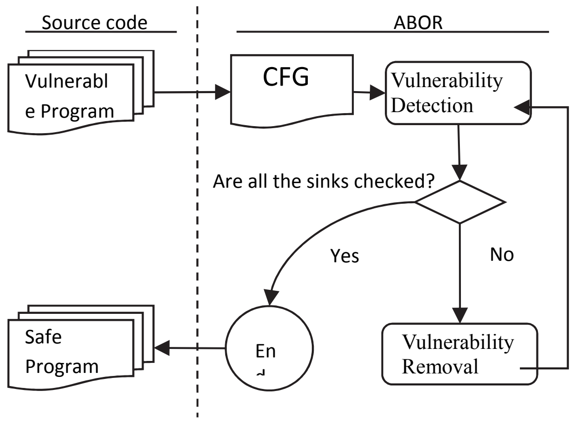

Figure 1: Overview of ABOR.

3.1 Buffer Overflow Patterns

We first review some basic definitions of static analysis (Sinha et al., 2001, Abstract syntax tree, 2014). A **control flow graph** (CFG) for a procedure is a graph that visually presents the control flow among program statements. A **path** is one single trace of executing a sequential of program statements. A variable in a program is called an **input variable** if it is not defined in the program solely from constants and variables. An **abstract syntax tree** (AST) is a tree structure that represents the abstract syntax of source code written in a particular programming language. We call a program operation that may cause potential buffer overflows as a **buffer overflow sensitive sink** (bo-sensitive sink). In this paper, we shorten the name "bo-sensitive sink" to sink. The node that contains a sink is called a **sink** node.

There are many methods to protect sinks from being exploited (Lei et al., 2008, Lin et al., 2007, Lhee and Chapin, 2003, Necula et al., 2005, Kundu and Bertino, 2011). One general way is to add extra protection constraints to protect sinks (Younan et al., 2012). After a careful review, we collect a list of common sinks and their corresponding protection constraints. We characterize them in a form of four patterns:

Pattern#1: A statement is a bo-sensitive sink when it defines or updates a destination buffer with an input from either (1) a C stream input function which is declared in *<stdio.h>*, or (2) a C++ input function inherited from the base class *istream*. The protection constraint shall ensure the length of the destination buffer is not smaller than the input data.

Pattern#2: A statement that copies/moves the content of a block of memory to another block of memory is a bo-sensitive sink. The protection constraint shall ensure that the copied/moved data is not larger than the length of the destination memory block.

Pattern#3: A statement is a bo-sensitive sink when it calls a C stream output function when (1) this function is declared in *<stdio.h>*; or (2) this function contains a format string that mismatches its corresponding output data; or (3) the function's output is data dependent on its parameters (Lhee and Chapin, 2003). The protection constraint shall ensure that:

- the output data should not contain any string derived from the prototype (C++Reference, 2014): *%[flags][width][.precision][length]specifier;*
- or all the character "%" in the output data has been encoded in a backslash escape style, such as "\%".

Pattern#4: A statement other than the above cases but referencing a pointer or array is a bo-sensitive sink. Before accessing this statement, there should be a protection constraint to do boundary checking for this pointer or array.

We use the above four patterns as a guideline to construct the buffer overflow detection and removal modules of ABOR. In order to specify these cases clearly, we use metadata to describe them exhaustively at the AST level. The metadata is maintained in Table 1 (Space lacks for a full table here, so we only show a fraction of Table 1. The full table is presented on our website (Ding, 2014)). Each row in Table 1 stands for a concrete buffer overflow case. The column *Sink* lists the AST structure of sinks. The column *Protection Constraint* specifies the AST structure of the constraint which could prevent the sinks being exploited. Additionally, in order to concrete the protection constraints ABOR needs to substitute in constants, local variables, expressions, and also two more critical data structures: the length of a buffer and the index of a buffer.

In C/C++, there is no universal way to retrieve a buffer's length and index easily. To solve this, ABOR creates intermediate variables to represent them. In Table 1, the last column *Required Intermediate Variable* records the required intermediate variables for each constraint.

Table 1: ABOR pattern lookup table.

Pattern	Sink	Protection Constraint	Required Intermediate Variable
1	gets(dst)	dst_length ≥ SIZE_MAX[1]	dst_length /*The destination buffer dst's length (bytes).*/
2	memcpy(dst, src, t)	dst_length ≥ t ; src_length ≥ t;	dst_length, //The destination buffer dst's length, in terms of bytes. src_length //The source buffer src's length, in terms of bytes.
……			
4	array[i]	sizeof(array)/sizeof(array[0]) ≥ i	N.A.
……			

[1] SIZE_MAX stands for the max value of unsigned long

3.2 Buffer Overflow Detection

Among the existing detection methods, approaches working in a path sensitive manner offer higher accuracy because they target at modeling the runtime behavior for each execution path:

1) For each path, path sensitive approaches eventually generate a path constraint to reflect the relationship between the external input and target buffer.
2) The path's vulnerability is verified through validating the generated path constraint.

In the current implementation of ABOR, we modified a recent buffer overflow detection method called Marple (Le and Soffa, 2008) and integrated it into ABOR. We chose Marple mainly because it detects buffer overflows in a path sensitive manner, which offers high precision.

Marple maintained a lookup table to store the common bo-sensitive sinks' syntax structure. For each recognized bo-sensitive sink node and the path passing through the sink node, Marple generates an initial constraint, called query and backward propagates the query along the path. The constraint will be updated through symbolic execution when encountering nodes that could affect the data flow information related to the constraint. Once the constraint is updated, Marple tries to validate it by invoking its theorem prover. If it is proved the constraint is unsatisfiable, Marple concludes this path buffer-overflow-vulnerable. Figure 2 demonstrates how ABOR integrates and extends Marple:

1) ABOR replaces Marple's lookup table with Table 1. We enforce Marple to search for the syntax structures listed in the column *Sink* of Table 1. Additionally, Marple will raise a query based on the column *Protection Constraint*.
2) ABOR uses a depth-first search to traverse a given procedure's control flow graph: each branch will be traversed once. If a bo-sensitive sink is found, the segment starting from the sink back to the procedure entry will be constructed to be a set of paths and each path will be fed to Marple for processing.
3) ABOR replaces Marple's constraint solver with Z3 (Z3, 2014), a latest SMT solver from Microsoft with strong solvability.
4) If one path is identified vulnerable, ABOR records the sub-path that causes the vulnerability or infeasibility (Le and Soffa, 2008). Later, paths containing any of such sub-paths will not be examined.
5) We follow the way Marple handles loop structures: we treat a loop structure as a unit and try to compute each loop's impact on the propagating constraint, if and only if such impact is linear.

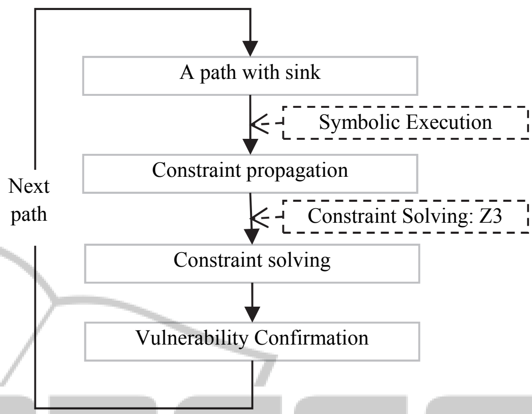

Figure 2: Vulnerability detection in ABOR.

We illustrate the above procedure with an example. In Table 4, *s1* is a sink node. Therefore, Marple raises a constraint as *pvpbuf_length* ≥ *req_bytes_length*. It propagates backward along the paths that pass through *s1* and tries to evaluate the constraint.

For example, along the path (*n1, n2, n3, n4, n5, n6, n7, n9, n10, n11, n12, n9, n13, n14, n15, n16, s1*), the constraint is updated at node *n13* and becomes *c2* ≥ *1024*. The variable *c2* is affected by the loop [*n9, n10, n11, n12, n9*]. The variable *c2* is used in the loop, and it is data dependent on the input--*ADDRSIZE*. There exists a counterexample to violate the constraint *c2* ≥ *1024*. So this path is identified as being vulnerable. The method introduced in section 3.3 will be used to remove this vulnerability.

3.3 Buffer Overflow Removal

The removal module takes an identified vulnerable path, analyzes the sink's AST and picks the corresponding constraint to protect the sink. The main challenge is to concrete the selected protection constraint into valid C++ code.

3.3.1 Intermediate Variable

It is important to trace the semantics of certain key operations along a path, including: buffer definition, buffer referencing, array indexing, pointer arithmetic and freeing memory. ABOR propagates backward along the given path to enable the intermediate variables simulating these semantics. Table 2 is used to assist this propagation. In Table 2, the column *Syntax Structure* stands for the AST structure ABOR searches for during the propagation. The column *Update Operation* stands for how ABOR updates the corresponding intermediate variables. (Space lacks for a full table here, so we only show a fraction of Table 2. The full table is presented on our website (Ding, 2014)).

3.3.2 Program Transformation for Vulnerability Removal

Table 3 shows the algorithm *removeVul* that describes ABOR's removal module. This algorithm takes an identified vulnerable path *pth* and a control flow graph *G* as parameters. It outputs a repaired control flow graph *G'* that is no longer vulnerable.

Table 2: Update intermediate variable during propagation.

Syntax Structure	Update Operation
Buffer definition	
• buffer = new *wildcard* T [n]; • T buffer[n];	buffer_length = n * sizeof(T);
Buffer referencing	
• T * p=buffer;	p_length=buffer_length; p_index=buffer_index;
Array index subscription	
• buffer[i] = *wildcard*	buffer_index=i;
Pointer arithmetic	
• p++; p--;	p_index ++; p_index--;
• p=p+n;	p_index = p_index +n;
Free memory	
• free (p); • delete [] p;	p_length=0; p_index=NULL;

The algorithm *removeVul* first identifies the sink along *pth* and then concretes the corresponding protection constraint by substituting required local variables, expressions, constants and intermediate variables. The concrete protection constraint is used to wrap the sink node to provide protections and, therefore, remove the vulnerability caused by the sink.

A full example of using ABOR is presented in Table 4. The vulnerable program is in the left side column, and the repaired program is in the right side column.

ABOR's buffer overflow detection module finds that the path (n1, n2, n3, n4, n5, n6, n7, n8, n14, n15, n16, s1) is vulnerable. The path will be passed to removeVul.

removeVul traverses the sink node s1's AST and determines that the first pattern shall be applied. The constraint is to validate that the length of pvpbuf is larger than or equal to the length of req_bytes.

ABOR inserts one node p1 into the CFG and creates two intermediate variables with an initial value of 0: pvpbuf_temp0_size for the length of pvpbuf and req_bytes_temp0_size for the length of req_bytes. ABOR inserts another three nodes p2, p3, and p4 into CFG, to manipulate the two intermediate variables to track the lengths of pvpbuf and req_bytes.

Table 3: Vulnerability removal in ABOR.

Input:	*G*: the CFG of a procedure; *pth*: the identified vulnerable path δ: the protection constraint for the sink
Global Variables:	{*v*}: the set of required intermediate variables
Output:	G': the CFG with the inserted defensive code

```
Algorithm removeVul (G , pth , s)
begin
    δ =NULL; {v}=∅; G'=G;
    CFGNode cur_node = NULL; //the current node traversed
    <δ, {v}> = LookupT1(sink ); /*this sub-procedure lookups in Table 1 and gets the metadata*/
    for (cur_node := From sink To pth's entry node) do
        // update intermediate variables
        for each v in {v} do
            Insert a CFGNode into G' as the its Entry Node's immediate post-dominator, to declare v and initialize v=0;
            if ( LookupT2(cur_node , v ) ==TRUE) then
                /*this sub-procedure lookups in Table 2 to check whether the current CFG Node contains AST matching the Syntax Structures in Table 2*/
                Insert a CFGNode into G' as cur_node's immediate post- dominator, to perform the corresponding update operation in Table 2;
            endIf
        endFor
    endFor
    Concrete δ with required local variables, expressions, constants and intermediate variables in {v};
    Use δ as condition to construct a predicate and insert this predicate node into G' as sink's immediate dominator's immediate post-dominator;
    Place sink and the rest part of G' on the predicate's TRUE branch;
    Add an exception-handling node on the predicate's FALSE branch and link this FALSE branch to G's exit node.
    G' → SrcFile' // convert G' back to source code
End
```

ABOR constructs the protection constraint as pvpbuf_temp0_size ≥ req_bytes_temp0_size and transforms the original program by wrapping statement s1 with statements p5, and p6.

At last, ABOR converts the modified CFG back to source code, which is listed in the right side column of Table 4. Therefore, the vulnerability has been removed. (Interested readers could refer to our website (Ding, 2014) for more examples of the

Table 4: An example of applying ABOR.

Vulnerable path	Sink	Pattern	Protection Constraint
(n1,n2,n3,n4,n5,n6,n7,n8,n14,n15, n16,s1)	/*s1*/ strcpy(pvpbuf, req_bytes)	Pattern 1	length(pvpbuf)≥length(req_bytes)

Vulnerable Program

```
#define min_size 10;
#define max_size 1024;
/*addr is an existing array with the size of ADDRSIZE,
n2 and n16 are two basic blocks that are not related to pvpbuf and req_bytes*/
int _encode (bool bslashmode, char *addr, int ADDRSIZE){

/*n1*/  do{
/*n2*/     BLOCK1;
/*n3*/  }while(…);

/*n4*/  int c1=0; int c2=0;
/*n5*/  int PSBUFSIZE= max_size;
/*n6*/  int *pvpbuf;
/*n7*/  if (bslashmode == TRUE){
/*n8*/     pvpbuf = new int [PSBUFSIZE];
       }else{

/*n9*/     while(c1<ADDRSIZE){
/*n10*/            if(addr[c1]!='#')
/*n11*/                  c2++;
/*n12*/            c1++;
           }
/*n13*/    pvpbuf = new int [c2];
       }

/*n14*/  char  req_bytes[1024];

/*n15*/  scanf("%1024s", req_bytes);
/*n16*/  BLOCK2;

/*s1*/   strcpy(pvpbuf,req_bytes);

}//END _encode
```

Repaired Program

```
#define min_size 10;
#define max_size 1024;
/*addr is an existing array with the size of ADDRSIZE,
n2 and n16 are two basic blocks that are not related to pvpbuf and
req_bytes*/

int _encode (bool bslashmode, char *addr, int ADDRSIZE){
/*p1*/   int pvpbuf_temp0_size=0;
         int req_bytes0_size=0;
/*n1*/  do{
/*n2*/     BLOCK1;
/*n3*/  }while(…);
/*n4*/  int c1=0; int c2=0;
/*n5*/  int PSBUFSIZE= max_size;
/*n6*/  int *pvpbuf;
/*n7*/  if (bslashmode == TRUE){
/*n8*/     pvpbuf = new int [PSBUFSIZE];
/*p2*/     pvpbuf_temp0_size=PSBUFSIZE;
       }else{
/*n9*/     while(c1<ADDRSIZE){
/*n10*/            if(addr[c1]!='#')
/*n11*/                  c2++;
/*n12*/            c1++;
           }
/*n13*/    pvpbuf = new int [c2];
/*p3*/     pvpbuf_temp0_size=c2;
       }
/*n14*/  char req_bytes[1024];
/*p4*/   req_bytes_temp0_size=1024;
/*n15*/  scanf("%1024s", req_bytes);
/*n16*/  BLOCK2;
/*p5*    if(pvpbuf_temp0_size >= req_bytes_temp0_size)
/*s1*/     strcpy(pvpbuf,req_bytes);
         else
/*p6*/     cerr<<"Attempt to cause buffer overflow reject";
}//END _encode
```


program transformation).

4 EVALUATION

4.1 Experiment Design

We implemented the proposed approach as a prototype system. The prototype has two parts: Program Analyzer and ABOR. The Program Analyzer receives C/C++ programs as input and utilizes CodeSurfer (CodeSurfer, 2012) to build an inline inter-procedural CFG. This CFG is then sent to the ABOR for the vulnerability detection and removal.

Nine systems are selected to evaluate ABOR's performance. Six of them are benchmark programs from Buffer Overflow Benchmark (Zitser et al., 2004) and BugBench (Lu et al., 2005), namely Polymorph, Ncompress, Gzip, Bc, Wu-ftdp and Sendmail. The rest three are industrial C/C++ applications, namely RouterCore, PathFinder, and RFIDScan (Celestvision, 2014).

For each system, we first run ABOR and then manually validate the results. The experiments are carried out on a desktop computer with Intel Duo E6750 2-core processor, 2.66 GHz, 4 GB memory and Windows XP system.

4.2 Experimental Results

4.2.1 System Performance

We evaluate ABOR's performance in terms of removal accuracy and time cost.

Removal Accuracy: For benchmark systems, the experimental results are shown in Table 5(a). A false negative case occurs if we manually find that the proposed method failed to remove one buffer overflow vulnerability. A false positive case occurs if we manually find the proposed method patched a piece of code that is actually buffer overflow free. We calculate the error rate of the proposed method by dividing the total number of vulnerabilities by the sum of the number of false positives and false negatives. The column KLOC stands for thousands of lines of code. The column #Reported records the real reported number of vulnerabilities while column #Repaired records the number of vulnerabilities removed by ABOR. The columns #FP and #FN stand for the numbers of false positives and false negatives respectively. For all the systems, the total number of reported buffer overflow vulnerabilities is 370. Therefore, ABOR can correctly remove all the vulnerabilities reported in previous work (Zitser et al., 2004, Lu et al., 2005).

For industrial programs, the results are shown in Table 5(b). The columns KLOC, #Repaired, #FN and #FP have the same meaning with Table 5(a). Additionally, the column #Detected stands for the number of detected vulnerabilities; and the column #Manual stands for the number of vulnerabilities discovered from manual investigations. The manual investigation double checked the detection result and analyzed the reason behind the cases that ABOR failed to proceed. As shown in Table 5(b), our approach detects 608 buffer overflow vulnerabilities and can successfully remove all of them. The results confirm the effectiveness of the proposed approach in removing buffer overflow vulnerabilities. However, due to implementation limitations, ABOR's detection modules didn't capture all the buffer overflow vulnerabilities in the industrial programs. On average, the error rate of our proposed method is 20.53%, which consists of 19.80% false negative cases and 0.73% false positive cases. The details of the error cases are discussed in section 4.2.3.

Table 5 (a): Vulnerabilities removed in benchmarks.

System	KLOC	#Reported	#Repaired	#FP	#FN
Polymorph-0.4.0	1.7	15	15	0	0
Ncompress-4.2.4	2.0	38	38	0	0
Gzip-1.2.4	8.2	38	38	0	0
Bc-1.06	17.7	245	245	0	0
Wu- ftdp-2.6.2	0.4	13	13	0	0
Sendmail-8.7.5	0.7	21	21	0	0
Total	30.7	370	370	0	0

Time Cost: We measured the time performance of the proposed approach on both benchmarks and industrial programs. Table 6 records the time spent on processing each program individually. ABOR is scalable to process large programs. The time cost over the entire nine systems is 4480 second, which is nearly 75 minutes. It is also discovered that a large amount of time is spent on vulnerability detection, which is 77.77% of the total time. The vulnerability removal process is relatively lightweight, which costs only 22.23% of the total time.

Table 5 (b): Vulnerabilities removed in industry programs.

System	KLOC	#Detected	#Repaired	#Manual	#FP	#FN	ErrorRate(%)
RouterC~	137.15	217	217	309	3	41	14.23
PathFinder	104.23	79	79	103	1	25	25.24
RFIDScan	219.36	312	312	406	2	96	24.13
Total	460.74	608	608	818	6	162	20.53

Table 6: The time performance of ABOR.

System	Total Time (ms)	Detection Time		Removal Time	
		Time (ms)	%	Time (ms)	%
Polymorph	95.25	81.91	85.99	13.34	14.01
Ncompress	214.32	160.81	75.03	53.51	24.98
Gzip	3698.16	2388.71	64.59	1309.45	35.41
Bc	149469.60	132026.90	88.33	17442.66	11.67
Wu- ftdp	221.13	185.33	83.81	35.80	16.19
Sendmail	134.82	103.76	76.96	31.06	23.04
RouterCore	2446656.17	1781649.13	72.82	665007.07	27.18
PathFinder	654987.43	564359.17	86.16	90628.22	13.84
RFIDScan	1224366.67	1003880.72	81.99	220485.98	18.01
Total	4479843.55	3484836.44	77.77	995007.09	22.23

4.2.2 Comparison

There are another two types of commonly used removal methods (Younan et al., 2012), which are "search & replace" and "bounds checker".

Search & Replace: The first category of methods replaces those common vulnerable C string functions with safe versions. If a program contains a large number of C string functions, this category of methods can achieve a good effect. Additionally, they are straight-forward for implementation (Miller and Raadt, 1999). But as the fast development of attacking techniques based on buffer overflows (Younan et al., 2012), the "search & replace", methods will miss many buffer overflows in real code.

Bounds Checker: The second category of methods chases high precision by inserting effective validation before every memory access. In practice, they are usually used by mission-critical systems (Younan et al., 2012) (Nagarakatte et al., 2009). However, they normally bring in high runtime overhead as a number of inserted validations are redundant.

ABOR is the method that only patches identified detected vulnerable code segment in a path-sensitive way. So it guarantees the removal precision while it can keep a low runtime overhead. Using the same benchmarks and industrial programs, we compare ABOR with the other two main categories of removal methods.

First, we compare ABOR with the "search & replace" category. In Figure 4, we compare ABOR with a recent "search & replace" method from Hafiz and Johnson (2009). It maintains a static database that stores common C/C++ vulnerable functions with their safe versions.

Figure 4 represents the histograms on the number of removed buffer overflows. It contains a pair of bars for each test case. The shadowed ones are for ABOR while the while bars are for the applied "search & replace" method. The Longer bars are better as they represent the higher removal precision. For the entire nine systems, in total, the "search & replace" method removes 570 vulnerabilities while ABOR removes 978 vulnerabilities (41.72% more). This is mainly because (1) many vulnerable codes are not covered by the static database of the "search & replace" method; (2) even after using a safe version of certain C/C++ functions, the vulnerabilities are not removed due to improper function parameters.

Figure 4: Comparison with a "Search & Replace" method.

Second, we compare ABOR with the bounds checker" category. At the current stage, we implemented a customized bounds checker following a novel approach from Nagarakatte et al., (2009). It will insert predicates to protect every suspicious sink.

Figure 5 represents the histograms on the number of inserted predicates. As with more inserted predicates, runtime overhead will increase. Figure 5 contains a pair of bars for each test case. The shadowed ones are for ABOR while the white ones are for the applied bounds checker. Shorter bars are better as they represent the fewer number of inserted predicates. For the entire nine systems, in total, the applied bounds checker inserts 3500 predicates. ABOR only patches confirmed sinks, so it only inserts 978 predicates, which are 72.05% less than the applied bounds checker.

Last but not least, though detection of buffer overflow is not a new research, till now, no approaches can detect buffer overflow with full coverage and precision. As the proposed approach uses these approaches, it is also limited by the accuracy of these approaches.

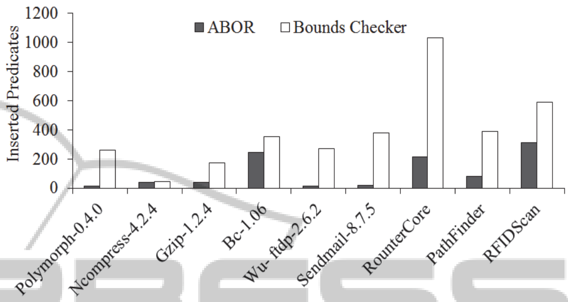

Figure 5: Comparison with a "Bounds Checker" method.

4.2.3 Discussion

It is also found that ABOR caused some false positive and false negative cases when processing the industrial programs. We further investigate these error cases and found the errors are mainly caused by implementation limitations. We categorize them into three types:

Error 1 - inaccuracy from alias analysis: it is difficult to implement a comprehensive alias analysis (CodeSurfer, (2012a)). Table 7(a) shows a false-negative example from RouterCore that is caused by inaccurate pointer analysis. In the loop, pointer ptr is incremented by one in each iteration until it equals to the address of pointer slt. So after the loop, ptr and slt are actually aliased. However, currently we cannot detect such alias relationship. The node n2 could overwrite the value of *ptr. Though node n1 does the boundary checking, it no longer protects node n3.

Error 2- inaccuracy from loop structure. So far only variables that are linearly updated within an iteration are handled by ABOR. Table 7(b) shows an example found in RouterCore. The variable x is non-linearly updated by using a bitwise operation. Therefore the corresponding constraint, which compares the size between buff and input, is beyond the solvability of our current implementation.

Error 3- platform-based data types: the industrial programs PathFinder and RFIDScan both involve external data types of Microsoft Windows SDK (e.g., WORD, DWORD, DWORD_PTR, etc.). This requires extra implementations to interpret them.

Additionally the value ranges of these data types specify implicit constraints. Table 7(c) shows a false-positive case from PathFinder. In this example, although accessing the buffer element via index at node n2 has no protection, there will be no buffer overflow vulnerability because index ranges only from 0 to 65535.

We summarize the error analysis in Table 8. In the future, further engineering effort is required to address these implementation difficulties.

Table 7: Examples of errors from processing the industrial programs.

(a) false-negative due to aliased pointer	(b) false-negative due to loop

void send(char mode){
// ptr and slt are two pointers
while(ptr < slt){
ptr++;
}
 	
/*n1*/ if(*ptr<sizeof(worduser)){
/*n2*/ cin.get(slt,80) ;
/*n3*/ worduser[*ptr] =mode; }
}
	……
	for(……){
	……
	x= x & y;
	}
	……
	char * buff = new char[x];
	……
	strcpy(buff, input);

(c) false-positive due to plat-based data type	
/*n1*/ WORD index=0; …… char buff[65536]; while(……){ …… /*n2*/ buff[index] = cin.get(); index++;}	

Table 8: Accuracy and error analysis.

System	Total Error	E1		E2		E3	
		#	%	#	%	#	%
RouterCore	44	23	52.27	10	22.73	11	25.00
PathFinder	26	16	61.54	0	0	10	38.47
RFIDScan	98	31	31.63	22	22.45	45	45.92
Total	168	70	41.67	32	19.05	66	39.29

5 RELATED WORK

We reviewed the recent techniques in addressing the buffer overflow vulnerability. We categorize them into three types: buffer overflow detection, runtime defense and vulnerability removal.

5.1 Buffer Overflow Detection

Buffer overflow vulnerability could be effectively detected by well-organized program analysis. The analysis could be performed either on source code or binary code. The current detection methods can be classified into path-sensitive approaches and non-path-sensitive approaches.

Path-sensitive approaches analyze given paths and generate path constraints according to the properties that ensure the paths are not exploitable for any buffer overflow attack. The path constraints are extracted using symbolic evaluation through either forward or backward propagation. A theorem prover or customized constraint solver is instrumented to evaluate the constraints. If a constraint is determined as unsolvable, the path is concluded as vulnerable. These methods pursue soundness and precision but usually include heavy overhead due to the use of symbolic evaluation. Typical examples include ARCHER (Xie et al., 2003), Marple (Le and Soffa, 2008), etc.

Non-path-sensitive approaches try to avoid complex analysis. They usually rely on general data flow analysis. These methods compute the environment, which is a mapping of program variables to values at those suspected locations. The environment captures all the possible values at a location L. Therefore, if any values are found to violate the buffer boundary at L; a buffer overflow vulnerability is detected. Choosing proper locations and making safe approximation requires heuristics and sacrifices precision. However, such a design gains more scalable performance, which is highly essential when dealing with large-scale applications. Typical examples include the work from Larochelle and Evans (2001) and CSSV (Dor et al., 2003).

5.2 Run-time Defence

This type of approaches adopt run-time defense to prevent exploiting potential vulnerabilities of any installed programs. These approaches aim to provide extra protection regardless of what the source program is. Normally such protections are implemented using three different kinds of techniques: code instrumentation, infrastructure modification, and network assistance.

The first aspect is to simulate a run-time environment partially, execute the suspicious program in a virtual environment and check whether malicious actions have taken place. Examples include StakeGuard and RAD (Wilander and Kamkar, 2003), which are tools that create virtual variables to simulate function’s return address. Suspicious code will be redirected to act over the virtual variables first.

The second aspect is to modify the underlying mechanism to eliminate the root of attacks: Xu et al. (2002) proposed a method to split stack and store data and control information separately; Ozdoganoglu et al., (2006) proposed to implement a non-executable stack. More details and corresponding tools of run-time defense could be found in recent surveys (Younan et al., 2012, Padmanabhuni and Tan, 2011).

The third aspect is to do a taint analysis for the input data from any the untrusted network source. This analysis compares network data with vulnerability signatures of the recorded attacks, or inspects payload for shell code for detecting and preventing exploitation. TaintCheck (Newsome and Song, 2005), proposed by Newsome and Song, tracks the propagation of tainted data that comes from un-trusted network sources. If a vulnerability signature is found, the attack is detected.

5.3 Attack Prevention through Auto Patching

The last category aims to removal the vulnerability from the source code. The objective is to add defensive code to wrap sink nodes so that no taint data could access the sinks directly. There are three strategies to design defensive code and insert them: search & replace, bounds checker, and detect & transform (Younan et al., 2012).

The first strategy is to search common vulnerable C string functions and I/O operations and replaces the found operations with a safe version. For example, vulnerable functions strcpy and strcat could be replaced as strncpy and strncat or strlcpy and strlcat (Miller and Raadt, 1999), or even with a customized version. Munawar and Ralph (Hafiz and Johnson, 2009) proposed a reliable approach to replace strcpy and strcat with a customized version based on heuristics. This strategy is straight-forward and easy to implement. However, it misses many complex situations.

The second strategy is aiming to insert effective validation before each memory access to perform an extra bounds checking. A typical design is to add validation before each pointer operation, named “pointer-based approach”. These approaches track the base and bound information for every pointer and validate each pointer manipulation operation against the tracked information. Examples include CCured, MSCC, SafeC, and Softbound (Nagarakatte et al., 2009). These approaches are designed to provide high precision. However, as nearly every pointer operation will be wrapped with additional checking, the code may grow largely and the runtime performance could be downgraded.

The third strategy is to locate the vulnerability first and then transform the vulnerable code segment into a safe version. Comparing with the second strategy, this helps reduce the size of added code without compromising the removal precision. Relatively few efforts have been put in this direction. For example, Wang et al. (Lei et al., 2008) proposed a method to add extra protection constraints to protect sink nodes. The method called model checker to verify the satisfiability of the inserted protection constraints. If the constraint fails to hold, that sink node will be recorded vulnerable, and the corresponding constraints will be left to protect the sink node. A similar example is from Lin et al., (2007) However, these methods are path-insensitive.

ABOR follows the third strategy and pushes the state-of-the-art one step ahead. It first detects the buffer overflows based on path sensitive information and then only add defensive code to repair the vulnerable code segment. Therefore, it adds limited code and has a low runtime overhead.

6 CONCLUSIONS

In this paper, we have presented an approach to remove buffer overflow vulnerabilities in C/C++ programs automatically. We first characterize buffer overflow vulnerability in the form of four patterns. We then integrate and extend existing techniques to propose a framework—ABOR that removes buffer overflow vulnerability automatically: ABOR iteratively detects and removes buffer overflows in a path-sensitive manner, until all the detected vulnerabilities are eliminated. Additionally, ABOR only patches vulnerable code segment. Therefore, it keeps a lightweight runtime overhead. Using a set of benchmark programs and three industrial programs written in C/C++, we experimentally show that the proposed approach is effective and scalable for removing all the detected buffer overflow vulnerabilities.

In the future, we will improve our approach and tool to minimize the number of wrongly detected cases. We will also integrate ABOR into existing testing frameworks, such as CUnit, GoogleTest, to further demonstrate its practicality.

ACKNOWLEDGEMENTS

The authors thank the JiangSu Celestvision from China for assisting this study and providing their internal programs for our experiment.

REFERENCES

Abstract syntax tree en.wikipedia.org/wiki/Abstract_ syntax_tree, 2014.

CodeSurfer, http://www.grammatech.com/, 2012.

C++Reference, http://www.cplusplus.com/reference/, 2014.

Celestvision, http://www.celestvision.com, 2014.

DingS-Buffer overflow detection. http://sunshine-nanyang. com/index.html, 2014.

US-CERT, http://www.us-cert.gov/, 2014.

Z3: SMT solver, http://z3.codeplex.com/, 2014.

Criswell, J., Lenharth, A., Dhurjati, D. and Adve, V. 2007. Secure virtual architecture: a safe execution environment for commodity operating systems. *SIGOPS Oper. Syst. Rev.*, 41, 351-366.

Dhurjati, D. and Aave, V. 2006. Backwards-compatible array bounds checking for C with very low overhead. *Proceedings of the 28th international conference on Software engineering*. Shanghai, China: ACM.

Dor, N., Rodeh, M. and Sagiv, M. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. PLDI '03: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003 San Diego, California, USA. ACM, 155-167.

Hafiz, M. and Johnson, R. E. 2009. Security-oriented program transformations. *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*. Oak Ridge, Tennessee: ACM.

Kundu, A. and Bertino, E. 2011. A New Class of Buffer Overflow Attacks. *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*. IEEE Computer Society.

Larochelle, D. and Evans, D. 2001. Statically detecting likely buffer overflow vulnerabilities. *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*. Washington, D.C.: USENIX Association.

Le, W. and Soffa, M. L. 2008. Marple: a demand-driven path-sensitive buffer overflow detector. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. Atlanta, Georgia: ACM.

Lei, W., Qiang, Z. and Pengchao, Z. Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking. Source Code Analysis and Manipulation, *2008 Eighth IEEE International Working Conference on*, 28-29 Sept. 2008. 165-173.

Lhee, K.-S. and Chapin, S. J. 2003. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exper.*, 33, 423-460.

Lin, Z., Jiang, X., Xu, D., Mao, B. & Xie, L. 2007. AutoPaG: towards automated software patch generation with source code root cause identification and repair. *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. Singapore: ACM.

Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P. and Zhou, Y. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

Miller, T. C. and Raadt, T. D. 1999. strlcpy and strlcat: consistent, safe, string copy and concatenation. *Proceedings of the annual conference on USENIX Annual Technical Conference*. Monterey, California: USENIX Association.

Nagarakatte, S., Zhao, J., Martin, M. M. K. and Zdancewic, S. 2009. SoftBound: highly compatible and complete spatial memory safety for c. *Proceedings* of the *2009 ACM SIGPLAN conference on Programming language design and implementation. Dublin,* Ireland: ACM.

Necula, G. C., Condit, J., Harren, M., Mcpeak, S. and Weimer, W. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27, 477-526.

Newsome, J. and Song, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

Ozdoganoglu, H., Vijaykumar, T. N., Brodley, C. E., Kuperman, B. A. and Jalote, A. 2006. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *Computers, IEEE Transactions on*, 55, 1271-1285.

Padmanabhuni, B. and Tan, H. 2011. Techniques for Defending from Buffer Overflow Vulnerability Security Exploits. *Internet Computing, IEEE*, PP, 1-1.

Sinha, S., Harrold, M. J. and Rothermel, G. 2001. Interprocedural control dependence. *ACM Trans. Softw. Eng. Methodol.*, 10, 209-254.

Vallentin, M. On the Evolution of Buffer Overflows. http://matthias.vallentin.net/course-work/buffer_ overflows .pdf, 2007.

Wilander, J. and Kamkar, M. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. Network and Distributed System Security Symposium(NDSS), 2003. 149-162.

Xie, Y., Chou, A. and Engler, D. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. ESEC/FSE-11: *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, 2003 Helsinki, Finland. ACM, 327-336.

Xu, J., Kalbarczyk, Z., Patel, S. and Ravishankar, I. Architecture Support for Defending Against Buffer Overflow Attacks. Second Workshop on Evaluating and Architecting System dependabilitY, 2002. 55-62.

Younan, Y., Joosen, W. and Piessens, F. 2012. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Comput. Surv.*, 44, 1-28.

Zitser, M., Lippmann, R. and Leek, T. 2004. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29, 97-106.