International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2021, 11(2): 121–147, doi:10.21655/ijsi.1673-7288.00246

International Journal of Software and Informatics

Research Article

Static Checking of Array Index Out-of-Bounds Defects in C Programs Based on Taint Analysis

Fengjuan Gao (高凤娟)[1,2], Yu Wang (王豫)[1,2], Tianjiao Chen (陈天骄)[1,2], Lingyun Situ (司徒凌云)[1,2], Linzhang Wang (王林章)[1,2], Xuandong Li (李宣东)[1,2]

[1] (Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[2] (State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing 210023, China)

Corresponding author: Linzhang Wang, lzwang@nju.edu.cn

Abstract During the rapid development of mobile computing, IoT, cloud computing, artificial intelligence, etc., many new programming languages and compilers are emerging. Nevertheless, C/C++ is still one of the most popular languages, and the array is one of the most important data structures of the C language. It is necessary to check whether the index is within the boundary of the array when it is used to access the elements of an array in a program. Otherwise, array index out-of-bounds will happen unexpectedly. When array index out-of-bounds defects are in programs, some serious errors may occur during execution, such as system crash. It is even worse that array index out-of-bounds defects open the doors for attackers to take control of the server and execute arbitrary malicious code by carefully constructing input and intercepting the control flow of the programs. Existing static methods for array boundary checking cannot achieve high accuracy and deal with complex constraints and expressions, leading to massive false positives. In addition, it will increase the burden of developers. In this study, a static checking method is proposed based on taint analysis. First, a flow-sensitive, context-sensitive, and on-demand pointer analysis is proposed to analyze the range of array length. Then, an on-demand taint analysis is performed for all array indexes and array length expressions. Finally, rules are defined for checking array index out-of-bounds defects and the checking is realized based on backward data flow analysis. During the analysis, in light of complex constraints and expressions, it is proposed to check the satisfiability of the conditions by invoking the constraint solver. If none statement for avoiding array index out-of-bounds is found in the program, an array index out-of-bound warning will be reported. An automatic static analysis tool, Carraybound, has been implemented, and the experimental results show that Carraybound can work effectively and efficiently.

Keywords array index out-of-bounds; static analysis; buffer overflow; constraint solving

Citation Gao FJ, Wang Y, Chen TJ, Situ LY, Wang LZ, Li XD. Static checking of array index out of bounds defects in C programs based on taint analysis, *International Journal of Software and Informatics*, 2021, 11(2): 121–147. http://www.ijsi.org/1673-7288/00246.htm

This is the English version of the Chinese article “基于污点分析的数组越界缺陷的静态检测方法. 软件学报, 2020, 31(10): 2983–3003. doi: 10.13328/j.cnki.jos.006063”.

Funding items: National Key Research and Development Program of China (2017YFA0700604); Program B for Outstanding PhD Candidates of Nanjing University; Postgraduate Research & Practice Innovation Program of Jiangsu Province of China

Received 2020-01-02; Revised 2020-04-04; Accepted 2020-05-09; IJSI published online 2021-06-22

Amid the rapid progress in mobile computing, Internet of Things (IoT), cloud computing, artificial intelligence, open-source software, open-source RISC-V instruction set and other fields, the development of related software and hardware is facing new opportunities and challenges. To adapt to this trend, substantial new programming languages and compilers have been emerging. Notwithstanding, as a general programming language of high efficiency, process orientation and abstraction, the C language is still widely applied in the development of system software. The system software with vulnerabilities may be maliciously exploited, seriously affecting people's production and life and even threatening the safety of life and property. As such, software security has emerged as an unavoidable challenge to software enterprises.

The C language is widely used in the development of the underlying software ecosystem, because C programs have higher running efficiency, and array is one of the most important data structures in the C language. When an array is employed in a program, the index used to access the array must be within a certain range, namely not less than 0 and smaller than the size of the array; otherwise, the index of the array will be out-of-bounds. Array index out-of-bounds defects are encountered frequently in compiled systems. Experiments reveal that mainstream the C compilers, including gcc and clang, do not strictly check the validity of array index ranges during compilation. Array index out-of-bounds are divided into reading/writing out-of-bounds. Reading out-of-bounds will lead to random values and then undefined behavior. In contrast, writing out-of-bounds will induce more serious consequences, including not only undefined behavior but also the interception of control flow, enabling attackers to execute arbitrary malicious code[1, 2]. As illustrated in Figure 1, according to CVE historical statistics, the top three types of vulnerabilities are denial of service, code execution and overflow[3], which are often accompanied with array index out-of-bounds. For example, the vulnerabilities of remote code execution and denial of service in Adobe Reader before 2017 are induced by the writing out-of-bounds that are possibly caused by external input as the array index (CVE-2017-16391, CVE-2017-16410). Meanwhile, researches[4–6] demonstrated 31% of buffer overflow was led by array index out-of-bounds. Accordingly, array index out-of-bounds defects still seriously threaten the security of system software. This paper mainly focuses on checking array index out-of-bounds and those caused by loops when the source code of C programs are given. For the higher security of system software, the program must check the boundaries of array indexes controlled by external input. However, developers may forget boundary checking or fail to perform correct checking, leaving the programs in an unfavorable state that can be exploited by attackers.

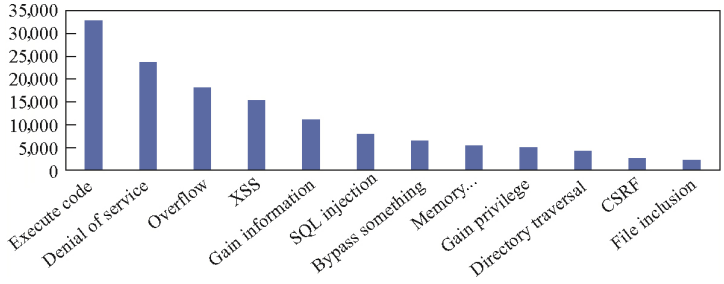


Figure 1 Statistics of the number of common types of CVE vulnerabilities

Some researchers have proposed to use static analysis and dynamic testing to check array index out-of-bounds. Because dynamic methods always depend on the integrity of test cases, they cannot achieve enough program coverage. Static analysis examines the defects in a program by scanning its source code. The current static analysis methods do not analyze the program

with high precision, for the sake of efficiency. Some of them cannot handle with array index out-of-bounds induced by loops. In addition, most of them are rule-based matching methods, which cannot deal with complex constraints and expressions, resulting in a large number of false positives and false negatives.

Therefore, for high-precision and efficient static checking of array index out-of-bounds defects, we propose to employ on-demand taint analysis to calculate the taint values of array indexes and array lengths. When the array length is tainted, even if the array index is not, it may still lead to array index out-of-bounds (such as the array access statement shown in Line 11 of Figure 2); when the array index is tainted, high-precision data flow analysis is required to check whether array index out-of-bounds will be induced. On the other side, we introduce an optimized constraint solver in the process of static analysis to process complex constraints and expressions related to array access, thereby markedly improving the accuracy of the checking method. With regard to array index out-of-bounds, we pay close attention to the satisfiability of conditions for them from program entry to array access statement. Introducing constraint solving into data flow analysis enables more accurate checking of array index out-of-bounds.

```
1.  typedef unsigned int UINT32;
2.  typedef struct {
3.      UINT32 noisy[12];
4.      UINT32 arr[15];
5.  }myStruct;
6.  myStruct s;
7.
8.  void f(UINT32 m,char* arr) {/*main->f: m<12*/
9.      UINT32 tmp[3]={1,2,3};
10.     UINT32 n=m;
11.     s.noisy[n]=0;
12.     arr[2]=0;
13.     for(UINT32 i=0;i<n;i++) {
14.         if(i>=15)
15.             break;
16.         s.arr[i]=tmp[i];
17.     }
18. }
19. int main(int argc, char** argv) {
20.     UINT32 j, k;
21.     scanf("%d %d",&j,&k);
22.     char* p = malloc(j);
23.     char* q = malloc(k);
24.     char** t;
25.     if (j>k)
26.         t=&p;
27.     else
28.         t=&q;
29.     if(argc+2<15){
30.         f(argc-1,*t); /*argc-1<12*/
31.     }
32.     return 0;
33. }
```

Figure 2 Example code of test.c

In this paper, we introduce a static analysis framework, Carraybound, which relies on static taint analysis, data flow analysis and constraint solving to check whether potential array index out-of-bounds defects exist in the program. In addition, Carraybound also provides array-boundary checking conditions to be added, helping programmers locate and confirm

the reported warnings and repair array index out-of-bounds in a more convenient and quicker manner. Experimental results demonstrate that Carraybound can check the array index out-of-bounds defects in programs efficiently and effectively, and it has performed better in finding those defects than existing static analysis tools including Cppcheck, Checkmarx and HP Fortify.

The main contributions of this paper include the following:

- Flow-sensitive, context-sensitive and on-demand pointer analysis is proposed to analyze the range of array length; on-demand taint analysis is introduced to calculate the taint values of array indexes and array lengths.
- A checking method of array index out-of-bounds defects is proposed on the basis of taint analysis, and a criterion rule is defined for array out-of-bounds defects. Then, according to the criterion rule, backward data flow analysis is adopted to detect array index out-of-bounds, and constraint solving is introduced into the process of data flow analysis to check the defects more effectively. At the same instant, the calls of the constraint solver are minimized through optimization to enhance analysis efficiency.
- A static analysis tool, Carraybound, is implemented to check array index out-of-bounds defects (including those caused by loops) in C programs, and the effectiveness of the tool is demonstrated by experiments.

Section 1 of this paper introduces the background knowledge of our work, and Section 2 elaborates on the static checking method of array index out-of-bounds, which is based on taint analysis. Section 3 presents the implementation tool, Carraybound, with its effectiveness and efficiency proved by experiments, and analyzes its shortcomings. Section 4 introduces the related work and Section 5 makes a summary and discusses the prospects for future work.

1 Background Knowledge

1.1 Array index out-of-bounds

An array is a vector of data of the same type stored continuously in memory. The arrays in the C language are split into static and dynamic ones. Static arrays are located in the stack area in memory, and their length is constant. When they are defined, a fixed length is allocated on the stack, which cannot be changed at runtime, as indicated by `char a`[7]. Writing out-of-bounds accesses to static arrays will cause buffer overflow on the stack. Dynamic arrays are located in the heap area in memory, and their length can be a variable, namely that the size can be dynamically allocated on the heap when the program runs, for instance `int *a=(int*)malloc(sizeof(int)*10)`. Writing out-of-bounds access to dynamic arrays will cause buffer overflow on the heap.

When an array is used in a program, the index accessing to the array must be within a certain range, namely not less than 0 and smaller than the size of the array; otherwise, the array index out-of-bounds will be induced[8]. As the array in Line 11 of Figure 2, because the length m of array arr originates from external input, the constant array index may also cause the out-of-bounds to access to the array.

As illustrated in Figure 3, we classify the problem of out-of-bounds access in C programs into the following two groups:

(1) Array index out-of-bounds, including reading/writing out-of-bounds: For example, `char c=a[5]` represents reading out-of-bounds, while `a[5]=0` belongs to writing out-of-bounds access to the array. Index writing out-of-bounds access to the array will lead to buffer overflow, as indicated by the intersection in Figure 3. Array index out-of-bounds also include the out-of-bounds caused by loops, such as `char a[5]; for (int i=0; i<6; i++) a[i]=0`.

(2) Buffer overflow, including that caused by API calls and index writing out-of-bounds access to the array, such as `strcpy(dest, src)` and `a[5]=0`.

The method proposed in this paper focuses on array index out-of-bounds (including those caused by loops). Our other work on automatic validation of static buffer overflow warnings highlights the buffer overflow caused by API calls[5].

Normally, programmers can limit the ranges of array indexes in specific ways to avoid array index out-of-bounds. Three common ways are as follows:

① `idx = idx % size;`
② `if (idx >= size || idx < 0)...`
③ `assert (idx >= 0 && idx < size);`

There may also be some complex constraints and expressions in the program to limit the ranges of array indexes, such as bitwise operations, linear operations with multiple operators, and even nonlinear constraints. These circumstances will make analysis more difficult, so that traditional methods sometimes fail to accurately identify the array out-of-bounds defects in programs.

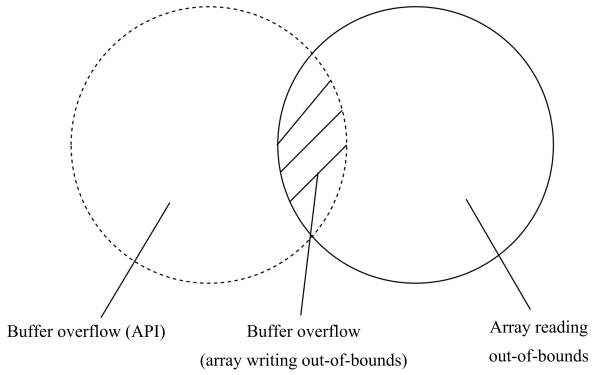


Figure 3 Buffer overflow vs. array index out-of-bound

1.2 Taint analysis

Taint analysis is a common technique for detecting program vulnerabilities[7, 9, 10]. If an attacker inputs some malicious data into a program that lacks proper protection measures, then the system may be unsafe. The data affected by external input is marked as taint for taint analysis, and external input includes user or file input and parameters of the main function. Taint analysis attempts to identify the variables in the program that can be tainted by user input and finally traces them to the statements that may lead to program defects. If the tainted data is directly used without checking before the statement, it is regarded as a program defect. Taint analysis is divided into static and dynamic modes. To be specific, dynamic taint analysis needs to execute programs, which cannot guarantee the coverage of source code; static taint analysis mainly relies on Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs) of programs to analyze data flow, which does not need to actually execute programs. Therefore, static taint analysis witnesses higher coverage of source code than dynamic taint analysis, but it may cause false positives and false negatives due to lack of runtime information.

1.3 Data flow analysis

Data flow analysis is usually applied in static code analysis, which is a technology to collect data flow information on the basis of CFGs. A simple method to analyze the data flow of a

program is to establish a data flow equation for each node of the CFG and perform iteration by repeatedly calculating the output at each node until the whole system reaches a fixed point[11, 12]. Data flow analysis has two different methods, i.e., forward and backward data flow analysis. Forward data flow analysis follows the normal execution path, starting from the entry node and ending at the target node. In this method, the exit state of a basic block is the result of the statements in the basic block acting on its entry state, and the entry state of a basic block is the combination of the exit states of all its predecessor basic blocks. On the contrary, the backward data flow analysis is opposite to the directed edge in the CFG, starting from the target node and ending at the entry node. In this method, the entry state of a basic block is the result of the statements in the basic block acting on its exit state, and the exit state of a basic block is the combination of the entry states of all its successor basic blocks.

1.4 Pointer analysis

The root of array index out-of-bounds defects is actually caused by pointer out-of-bounds access to memory. The array name represents a pointer to the first element of the array, and accessing the array element (e.g., `p[i]`) through the index is actually equivalent to that by moving the pointer from the first element of the array to a specific element (e.g., `*(p+i)`). Accordingly, pointer analysis is required to calculate the memory area that the array name actually points to.

Pointer analysis is a special data flow problem, which refers to calculating the set of pointer expressions pointing to the same memory area by program analysis. Pointer analysis has several important accuracy measurement attributes, such as flow sensitivity and context sensitivity. Flow-sensitive pointer analysis can distinguish the pointing information of pointer variables in different control flow positions. Context sensitivity reflects whether interprocedural analysis can distinguish the difference between the roles of contexts of calling points in procedure input, thus affecting procedure output.

1.5 Satisfiability of SMT

A Satisfiability Modulo Theories (SMT) solver is a program to judge the satisfiability of first-order logic formulas, which is the verification engine of multiple formal methods[13]. SMT solution technology is extensively applied to bounded model checking, program analysis based on symbolic execution, linear planning and scheduling, test case generation, circuit design and verification, etc.

Z3[14, 15] is a high-performance SMT solver developed by Microsoft Research Lab, which has been the SMT solver with the strongest comprehensive solving ability so far. Therefore, this paper makes use of the constraint solver Z3 to help Carraybound check array index out-of-bounds defects in a more accurate manner.

2 Carraybound: Static Checking of Array Index Out-of-bounds Defects Based on Taint Analysis

2.1 Methodological framework

The methods for checking array index out-of-bounds defects in this paper are mainly based on static taint analysis and data flow analysis, and these methods are mainly based on ASTs, call graphs and CFGs of programs. The methodological framework is illustrated in Figure 4. Firstly, ASTs are generated according to the source code of a program, and then call graphs and CFGs are constructed according to ASTs. Then, based on CFGs, ASTs and call graphs, taint analysis is performed to determine the array indexes that may be tainted. All the statements containing the array expressions of these tainted array indexes are located, with the array indexes symbolized, and the boundary information of each array index is obtained by boundary analysis.

Subsequently, backward data flow analysis is performed, and simple matching and constraint solving are provided to check whether corresponding expressions are in the program to ensure the boundary conditions of array indexes. If these expressions do not exist, a warning of array index out-of-bounds defects is reported.

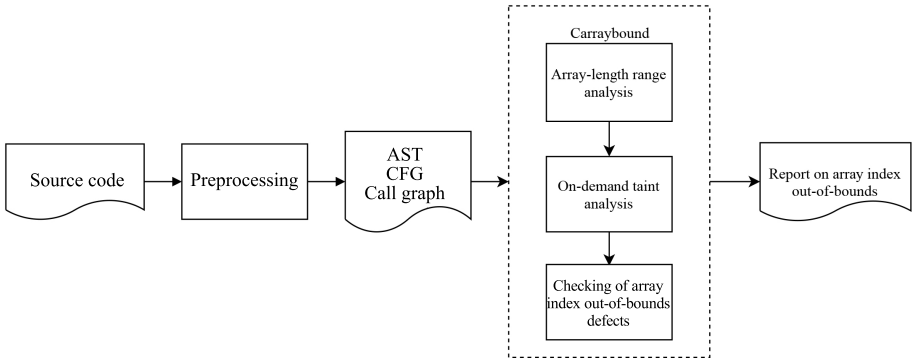


Figure 4 Overall framework of our method

2.2 Analysis of array length range

First, the statements for array index access are located, and then the alias of the concerned array name is obtained through backward data flow analysis. Then the statements for array declaration are located, and each statement corresponds to an array length. Because an array may correspond to multiple statements for array declaration, the range of array length will be obtained through analysis in this process. The specific steps are as follows:

Analysis of array length range: With regard to the array access statement $arr[idx]$, it is determined whether there is an array declaration size on the AST. If the array declaration size cannot be directly acquired, pointer analysis is performed, and it is found that arr is actually an alias of one or several static or dynamic arrays.

The flow-sensitive, context-sensitive and on-demand pointer analysis is designed in this paper. "On-demand" means this paper only performs pointer analysis on the concerned array names (those in the statements accessing array elements through tainted indexes) to calculate the static or dynamic array expressions that the array names actually correspond to. For each function f, the array expressions containing tainted array indexes in each basic block are counted first. Then, starting from the bottom layer containing the concerned array expressions, backward data flow analysis is carried out upwards. In backward data flow analysis, $OutState$ represents the state of a basic block at its exit, which is the union of $InState$ of all its successor basic blocks, as shown in Formula (1). Here, it refers to the pointer set to be analyzed that each concerned array name corresponds to in the basic block; $InState$ indicates the state of a basic block at its entry, as shown in Formula (2). Here, it refers to the result after the assignment statements of the basic block kill and generate the pointer set to be analyzed each concerned array name corresponds to based on its $OutState$ according to the following pointer processing rules:

$$\mathit{OutState}(\mathit{block}) = \cup_{s \in succ(block)} \mathit{InState}(s) \tag{1}$$

$$\mathit{InState}(\mathit{block}) = \mathit{Gen}(\mathit{block}) \cup (\mathit{OutState}(\mathit{block}) - \mathit{Kill}(\mathit{block})) \tag{2}$$

During pointer analysis, $InState$ and $OutState$ maintain the set of pointers to be analyzed that each concerned array name arr corresponds to in a certain basic block, i.e., $AliasSet =$

$\{p_1, p_2, \cdots, p_n\}$. At the initial time, $AliasSet = \{arr\}$. For each pointer p in $AliasSet$, it is first queried whether the size of static array declaration can be directly obtained on the ASTs; if so, the element p in $AliasSet$ is marked as the end point, and the pointer analysis of p is stopped; otherwise, the pointer analysis of p continues. In regard to the assignment statements, the specific pointer processing rules for backward array flow analysis are as follows:

- $p = malloc(size)$ replaces p in $AliasSet$ with $malloc(size)$ and marks the element $malloc(size)$ in $AliasSet$ as the end point; point analysis of $malloc(size)$ is stopped.
- $p = \&a$ replaces p in $AliasSet$ with $\&a$.
- $p = q$ replaces p in $AliasSet$ with q; subsequent data flow analysis continues with regard to the alias of q.
- $p = *q$ replaces p in $AliasSet$ with $*q$.
- $*p = q$ replaces p in $AliasSet$ with $\&q$.
- $p = g(\cdots)$ will enter the function g, and the pointer analysis will be started from the function return statement.

If the element in $AliasSet$ contains symbol &, such as $\&p$, the assignment expression of p is analyzed in the subsequent data flow analysis. If the new pointer is q, $\&p$ in $AliasSet$ is replaced with $\&q$; if the new pointer is $*q$, $\&p$ in $AliasSet$ is replaced with $\&(*q)$, namely replaced with q, and so on. In the rules, p in $AliasSet$ is replaced by q, which corresponds to $Kill(p)$ and $Gen(q)$ in $AliasSet$.

If the array declaration statement corresponding to the array name is not located in the function f, all callers of the function are analyzed by the same method, until it is located. Due to different contexts and branches, an array name may have multiple array declaration statements as aliases, with each array declaration statement corresponding to an array length. Therefore, for an array name arr, a group of array lengths $\{s_0, s_1, \cdots, s_n\}$ will be obtained by pointer analysis. In light of the support for flow sensitivity and context sensitivity, additional records will be kept during data flow analysis:

(1) the set of successor nodes of each basic block (calculation method as Formula (1)), denoted as $succs(block)$;

(2) the function call chain during the backward data flow analysis of function f, denoted as $succs(f)$.

With regard to the array name arr, during pointer analysis, the array length corresponding to p_0 is set as s_0 when an array declaration statement of arr is located as an alias in the basic block bb of function f (one of the end points in $AliasSet$ is marked as p_0). Then the valid function and basic block information corresponding to s_0 will be recorded. The valid function information is the function call chain of backward data flow analysis between procedures, namely $ValidFuncs(s_0) = succs(f)$; the valid basic block information is the set of successor nodes of this basic block, namely $ValidBBs(s_0) = succs(bb)$. At last, for each array name arr, we record a group of array lengths $\{s_0, s_1, \cdots, s_n\}$, and for each array length s_i, we record the scope of the value, namely valid function $ValidFuncs(s_0) = succs(f)$ and valid basic block $ValidBBs(s_0) = succs(bb)$. On this basis, the set of array lengths, $size(arr, f, bb) = \{s_i, s_j, \cdots, s_k\}$, is derived, to which the array name arr corresponds to in each basic block bb of each function f.

According to the set of array lengths, $size(arr, f, bb) = s_i, s_j, \cdots, s_k$, of array name arr in basic block bb of each function f, the maximum is taken as the upper bound up, while the minimum as the lower bound low. As a result, the range of array length is $[low, up]$. If the array length is an external input variable and the upper and lower bounds cannot be determined, the length set is reserved.

Case analysis: As the code example in Figure 2, traversal on the ASTs reveals four

locations of array expressions in function f, namely $s.noisy[n]$ in Line 11, $arr[2]$ in Line 12, as well as $s.arr[i]$ and $tmp[i]$ in Line 16. $s.noisy[n]$ and $s.arr[i]$ are both arrays of structure members. Their array declaration positions can be directly located in Lines 3 and 4, and the array lengths are 12 and 15, respectively. The array declaration position of $tmp[i]$ is in Line 9, and its array length is 3. $arr[2]$ comes from the function parameter; according to pointer analysis, dynamic arrays p and q are aliases of arr, so the array length set of arr is j, k.

2.3 On-demand taint analysis

On-demand taint analysis means that only static taint analysis is carried out in this paper with regard to array indexes and array lengths in the program, including intraprocedural and interprocedural taint analysis. In this paper, external input (including user or file input and parameters of the main function) is taken as the source of taint. Through taint propagation, the taint value $T(v)$ of each variable v of interest in the program can be obtained, which can be tainted or untainted, namely

$$T(v) \in \{tainted, untainted\}$$

The $tainted$ value can correspond to Boolean value 1, while the $untainted$ value to Boolean value 0. Therefore, the logical operator "$\vee$" can be used to calculate the sum of taint values. In other words, as long as one subexpression has $tainted$ value, the value of the whole expression is $tainted$.

2.3.1 *Rule of taint propagation*

For each statement encountered in taint analysis, the taint values of the statements will be calculated according to the following rule of taint propagation.

Constant: Each constant c is untainted, such as string constant, integer constant and floating point constant.

$$T(c) = untainted$$

Type conversion: The taint value of the expression $CastExpr(e)$ after type conversion is consistent with that of the expression e of the original type.

$$T(CastExpr(e)) = T(e)$$

Array index expression: It will be regarded as a whole. If an element of the array is tainted, the whole array is tainted, which is the same rule for a structure.

$$T(arr[i]) = T(arr), T(expr.elem) = T(expr)$$

Unary operation expression: The taint value of *op expr* is equal to that of the expression $expr$.

$$T(op\ expr) = T(expr).$$

Binary operation expression: The taint value of $expr1$ *op* $expr2$ is equal to the sum of taint values of its subexpressions $expr1$ and $expr2$.

$$T(expr1\ op\ expr2) = T(expr1) \vee T(expr2)$$

Ternary operation expression: The taint value of $expr1?expr2 : expr3$ is equal to the sum of taint values of its subexpressions $expr2$ and $expr3$.

Assignment expression: The assignment statement $expr1 = expr2$ will propagate the taint value of the right expression to the left variable.

$$T(expr1) = T(expr2)$$

Conditional statement: if c then $expr1$ else $expr2$ will propagate the taint value of conditional expression c in the conditional statement to the L-value in the assignment statement of the basic block.

Function call statement: If the function f has n parameters, the call statement for f will propagate the taint value of the i-th actual parameter p_i to the i-th formal parameter a_i.

$$\forall i \in [0, n), T(a_i) = T(p_i)$$

At the same time, the function call statement will propagate the taint state of the returned value of the called function to the assigned variable of the caller.

Function return statement: If a variable is returned, the taint value of the function return value is equal to that of the variable; if a constant (including null value) is returned, the taint value of the function return value is |untainted|.

2.3.2 *On-demand intraprocedural taint analysis*

Before taint analysis, all the functions related to array indexes and lengths in the program and all caller functions up to the entry function are counted to form an array-related function set FS. Intraprocedural taint analysis indicates the forward data flow analysis of each function in FS. For each basic block in a function, $InState$ represents the taint state of a basic block at its entry, which is the union of $OutState$ of all its predecessor basic blocks and indicates the taint state of all expressions before the basic block is executed; $OutState$ denotes the taint state of a basic block at its exit, which is the result of updating the taint state of expressions by statements in the basic block according to the taint propagation rules in the previous section on the basis of $InState$. $Kill$ will eliminate the taint state of expressions, while Gen will generate the taint state of them:

$$InState(block) = \cup_{p \in pred(block)} OutState(p) \tag{3}$$

$$OutState(block) = Gen(block) \cup (InState(block) - Kill(block)) \tag{4}$$

In regard to functions with loops, $InState$ and $OutState$ of each basic block will be iteratively calculated until the states of $InState$ and $OutState$ of the basic block are constant. The taint state of the function is the same as the $OutState$ of the basic block at the function exit. Then the taint relation between all expressions and the corresponding formal parameters in the function can be obtained, namely the taint summary $TS(f)$ of the function f.

For each function f, the list of formal parameters is $A = \{a_1, a_2, \cdots\}$. The taint state of each variable v in the function is recorded as $T(v)$, and its value may be $tainted$, $untainted$ or dependent on the formal parameters of the function, namely

$$T(v) = \begin{cases} tainted \\ untainted \\ \cup_{a \in A'} T(a), A' = \{a_i RelyOn(a_i, v), a_i \in A\} \end{cases} \tag{5}$$

2.3.3 *On-demand interprocedural taint analysis*

Firstly, the parameters of the entry function are marked as tainted. Then, with the entry function as the starting point, all functions of FS are analyzed according to the topological sequence on the call graph. Through the function call statement, the taint values of actual parameters are propagated to formal parameters of the function at the call point, and the taint value of each formal parameter in FS is calculated. If multiple functions call the same function, the taint value of the called function's parameters is the sum of the taint values of all its callers' actual parameters.

The taint value of the i-th formal parameter a_i^f of function f is the sum of the taint values of the actual parameters p_i corresponding to all callers $Caller_1, \cdots, Caller_j$ of f, namely

$$T(a_i^f) = \cup_{k=1}^{j} T(p_i^{caller_k})$$

When the taint state of an expression in the program is required to be queried, it will be returned immediately ($tainted/untainted$) if the taint value can be obtained directly; otherwise, the taint state of the expression depends on the formal parameter of the function. At this time, the taint value of the original expression can be obtained by substituting the taint value $T(a)$ of the function parameter into the third assignment of Formula (5).

2.3.4 *Case analysis*

Figure 2 presents a code snippet, and Figure 5 illustrates the CFG of function f. The number after the colon in Figure 5 represents the line number of entry statements. The analysis shows that the array indexes of four array expressions are n, 2 and i. The taint analysis of function f reveals that the taint values of variables n and i in f are consistent with the parameter m of f. Then taint analysis is performed on the $main$ function. $argc$ and $argv$ are external input and thus tainted. The $main$ function calls the function f through $argc - 1$, making the formal parameter m of f tainted; furthermore, the variables n and i in f are also tainted.

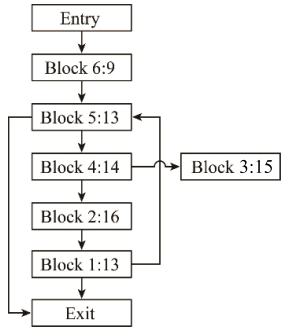


Figure 5 CFG of function f in test.c

2.4 Checking of array index out-of-bounds defects

For each array access statement $arr[idx]$, the list of lengths corresponding to array arr is $\{len_0, len_1, \cdots, len_n\}$, and the checking result of array index out-of-bounds defects is recorded as $W(arr[idx])$; $W(arr[idx]) \equiv T$ indicates that the array access statement will cause array index out-of-bounds, while $W(arr[idx]) \equiv F$ denotes an opposite situation.

Decision rule 1: For array access statement $arr[idx]$, it will cause array index out-of-bounds, if the array index idx is untainted but any value in the list of array lengths is tainted.

$$\begin{cases} T(idx) \equiv untainted \\ \exists i \in [0, n], T(s_i) \equiv tainted \end{cases} \Rightarrow W(arr[idx]) \equiv T \tag{6}$$

Decision rule 2: For array access statement $arr[idx]$, when the array index idx and each value in the list of array lengths are untainted, it will not cause array index out-of-bounds, if the array index is less than each value in the list of array lengths; otherwise, array index out-of-bounds will be induced by the array access statement.

$$\begin{cases} T(idx) \equiv untainted \\ \forall i \in [0, n], T(s_i) \equiv untainted \end{cases} \Rightarrow W(arr[idx]) \equiv \begin{cases} T, & \exists i \in [0, n], idx \geq s_i \\ F, & \forall i \in [0, n], idx < s_i \end{cases} \tag{7}$$

Decision rule 3: For array access statement $arr[idx]$, if the array index idx is tainted, it is assumed that n statements related to idx are in the program, in a sequence of $S_1, S_2, \cdots, S_n$. Every statement is converted into a constraint expression, and the sequence of constraint expressions is $c_1, c_2, \cdots, c_n$. Consume the i-th statement, S_i, is in the basic block bb of function f; then the index corresponding to the array arr is idx_i, and the list of array lengths is $LenSet_i = \{len_1^i, len_2^i, \cdots, len_m^i\}$. The array access statement will cause array index out-of-bounds if there is a statement S_i, based on which idx is greater than or equal to any length in $LenSet_i$ can be deduced (Condition A in Formula (8)); or there is a statement S_i, based on which idx is smaller than 0 can be deduced (Condition B in Formula (8)); or for all statements, it cannot be deduced that idx is smaller than all lengths in $LenSet_i$ (Condition C in Formula (8)) or greater than or equal to 0 (Condition D in Formula (8)). On the contrary, the array access statement will not cause array index out-of-bounds if there is a statement S_i, based on which idx is smaller than all lengths in $LenSet_i$ can be deduced (Condition E in Formula (8)), and a statement S_j, based on which idx is greater than 0 can be deduced (Condition F in Formula (8)).

$$\begin{cases} A : (\exists i, \exists k, !(c_i \rightarrow (idx \geq len_k^i)) \equiv \textit{UNSAT}) \\ B : (\exists i, !(c_i \rightarrow (idx < 0)) \equiv \textit{UNSAT}) \\ C : (\forall i, \forall k, !(c_i \rightarrow (idx < len_k^i)) \equiv \textit{SAT}) \\ D : (\forall i, !(c_i \rightarrow (idx \geq 0)) \equiv \textit{SAT}) \\ E : (\exists i, \forall k, !(c_i \rightarrow (idx < len_k^i)) \equiv \textit{UNSAT}) \\ F : (\exists j, !(c_j \rightarrow (idx \geq 0)) \equiv \textit{UNSAT}) \\ T(idx) \equiv tainted \Rightarrow W(arr[idx]) = \begin{cases} T, A \vee B \vee C \vee D \\ F, E \wedge F \end{cases} \end{cases} \tag{8}$$

According to the above three decision rules, this paper will refer to Algorithm 1 to check array index out-of-bounds in the program. With regard to each array access statement, the taint value of the array index is first queried. If the array index is untainted, it will be checked whether the array access statement causes the array index out-of-bounds according to Rules 1 and 2; if the array index is tainted, high-precision backward data flow analysis will be adopted to check whether it may cause array index out-of-bounds. In backward data flow analysis, $OutState$ indicates the state of a basic block at its exit, which is the union of $InState$ of all its successor basic blocks, as shown in Formula (4) (Section 2.3.2). Here, it represents the set of array access statements that may lead to array index out-of-bounds, which is to be checked in the basic block; $InState$ indicates the state of a basic block at its entry, as shown in Formula (5) (Section 2.3.2). Here, it refers to the result of killing and generating the set of array access statements, which is to be checked if it will cause array index out-of-bounds, by the statement of the basic block according to Rule 3 and Table 1. Intraprocedural backward data flow analysis starts from the basic block where the array access statement is located, traverses up every basic block in the function, and ends at the entry point. After a basic block bb is analyzed, its predecessor basic block $pred$ will be the next object. At this time, it will be decided whether to take the inverse conditions of the branch statement according to the branch that bb is located in $pred$. If the statement set to be checked is empty when the entry point is reached, the backward data flow analysis will be terminated; otherwise, that between procedures will continue. First, all the parent functions of the function f need to be acquired according to the function call graph of the program. For each parent function, its CFGs are traversed to find the statement that calls the array function f. For each array boundary condition, if its index is the same as one of the formal parameters of f, the corresponding actual parameters will be obtained to construct a new array boundary condition, as Line 7 of the $interABChecker$ algorithm. Then, the intraprocedural

backward data flow analysis continues in the parent functions. In addition, the interprocedural backward data flow analysis will be kept executing until the set of statements to be checked is empty or reaches the configured standard.

Table 1 Types of statements related to array index out-of-bounds checking

Statement type	Mode	Simple matching	Constraint solving
Declaration	Type $idx = expr$	$idx = const$	$!(idx == expr \rightarrow idx < len)$ $!(idx == expr \rightarrow idx >= len)$
		$idx = expr\%const$	$!(idx == expr \rightarrow idx < 0)$ $!(idx == expr \rightarrow idx >= 0)$
Assignment	$idx = expr$	*ditto*	*ditto*
Compound assignment	$idx\ op = expr$	$idx\% = const$	$!(idx_1 == idx_0\ op$ $expr \rightarrow idx_1 < len)$
Condition	$if(expr)$	$idx < const$ $idx <= const$	$!(expr \rightarrow idx < len)$
		$idx > const$ $idx >= const$	$!(expr \rightarrow idx > 0)$
for loop	$for(\cdots;expr;\cdots)$	*ditto*	*ditto*
while loop	$while(expr)$	*ditto*	*ditto*

Algorithm 1. Array index out-of-bounds checking

Function: $ABChecker(CallGraph, CFG, Depth)$.
Input: $CallGraph, CFG, Depth$;
Output: $Warnings$.

```
1.  foreach f in CallGraph do /*in backwards topological order*/
2.      bbSet = ∅
3.      foreach BB in CFG off do /*in backwards topological order */
4.          foreach ArrStmt in BB do
5.              if T(ArrStmt.idx) == untainted then
6.                  if T(ArrStmt.LenSet) == tainted then
7.                      Warnings.add(ArrStmt)
8.                  else
9.                      foreach Len in ArrStmt.LenSet do
10.                         if ArrStmt.idx ≥ Len then
11.                             Warnings.add(ArrStmt)
12.                         end if
13.                         break
14.                     end foreach
15.                 end if
16.                 else
17.                     OutState[BB].add(ArrStmt)
18.                     if OutState[BB] ≠ ∅ then
19.                         bbSet.add(all predecessors of BB)
20.                     end if
21.             end if
22.         end foreach
23.     end foreach
24.     ABCSet = intraABChecker(bbSet, OutState)
25.     if ABCSet ≠ ∅ then
26.         result = interABChecker(ABCSet, f, Depth − 1)
27.     end if
28. end foreach
29. Warnings.add(all ArrStmt in result)
```

Function: $intraABChecker(bbSet, OutState)$.
1. **while** $bbSet \neq \varnothing$ **do**
2. $BB = bbSet.pop()$
3. **foreach** $succ$ of BB **do**
4. $OutState[BB] += InState[succ]$
5. **end foreach**
6. $InState[BB] = \varnothing$
7. **foreach** $stmt$ inBB **do**
8. $checkStmt(stmt, \&OutState[BB])$
9. **end foreach**
10. **if** $OutState[BB] \neq InState[BB]$ **then**
11. $InState[BB] = OutState[BB]$
12. $bbSet.add$(all predecessors of BB)
13. **end if**
14. $ABCSet = OutState[BB]$
15. **end while**
16. **return** $ABCSet$

Function: $checkStmt(stmt, OutState[BB])$.
1. **foreach** $ABCinOutState[BB]$ **do**
2. **if** $imply(stmt, ArrStmt.idx, less, 0)$ **then**
3. $Warnings.add(ABC)$
4. $OutState[BB].remove(ABC, low)$
5. **else if** $imply(stmt, ArrStmt.idx, notless, 0)$ **then**
6. $OutState[BB].remove(ABC, low)$
7. **end if**
8. **end if**
9. **foreach** len in $ArrStmt.LenSet$ **do**
10. **if** $imply(stmt, ArrStmt.idx, less, len)$ **then**
11. $cnt++$
12. **else if** $imply(stmt, ArrStmt.idx, notless, len)$ **then**
13. $Warnings.add(ABC)$
14. $OutState[BB].remove(ABC, up)$
15. **break**
16. **end if**
17. **end if**
18. **if** $cnt == ArrStmt.LenSet.size()$ **then**
19. $OutState[BB].remove(ABC, up)$
20. **end if**
21. **end foreach**
22. **end foreach**

Function: $interABChecker(ABCSet, f, Depth)$.
1. **if** $Depth \leq 0$ or $ABCSet == \emptyset$ **then**
2. **return** $ABCSet$
3. **end if**
4. $result = \varnothing$
5. **foreach** $caller$ off **do**
6. $bbSet = \varnothing$
7. $bbSet.add(caller.callsite.BB)$
8. $OutState[callerBB] = update(ABCSet)$
9. $set = intraABChecker(bbSet, OutState)$
10. $set2 = interABChecker(set, caller, Depth - 1)$
11. $result += set2$
12. **end foreach**
13. **return** $result$

As indicated by the function $checkStmt$ in Algorithm 1, every statement |stmt| encountered during data flow analysis will be processed according to Table 1 and Rule 3. If a statement can

satisfy Condition A or E in Formula (8), then A, C and E will no longer be concerned in the subsequent data flow analysis; if a statement can meet Condition B or F in Formula (8), then B, D and F will no longer be concerned during backward data flow analysis. In the process of backward data flow analysis, the statements related to array indexes are the main concerns, as shown in the first two columns of Table 1, which mainly include conditional statements (including loop conditions) of the array index and assignment expressions (including expressions of declarative and compound assignment) for the array index. If the data type declared by $|idx|$ in the declarative assignment expression is unsigned, Condition F in Formula (8) holds, while Conditions B and D do not hold. In this paper, two methods of judgment, i.e. simple matching and constraint solving, are provided to check whether Conditions A–F are satisfied, and those conditions are uniformly expressed by $c_i \rightarrow (idx\ op\ expr)$. In the following description, we take as an example the conditional statement of the basic block where the array access statement is located.

Simple matching: It mainly deals with statements containing target array indexes and constants, namely with the conditional format as $(idx\ op\ const_1) \rightarrow (idx\ op\ const_2)$, and when the two operators *op* are consistent, the satisfiability of the condition can be judged by comparing two constants $const1$ and $const2$. In regard to different types of statements (the third column of Table 1), the specific processing rules are as follows:

- Assignment statement: It can only deal with the cases where the statements are $idx = const$ and $idx = expr\%const$. If $const$ in $idx = const$ is greater than 0, then Condition F in Formula (8) holds, while Conditions B and D do not hold; if the array length len in Condition E of Formula (8) is also a constant and the constant $const$ in the statement is less than or equal to all array lengths len, then Condition E holds, while Conditions A and C do not hold; if the constant $const$ in the statement is larger than any constant length len, Condition A holds, and a report of array index out-of-bounds is delivered. In case of statement as $idx = expr\%const$, only when the array length len in Condition E of Formula (8) is also a constant, if the constant $const$ is less than or equal to all array lengths len, Condition E holds, while Conditions A and C do not hold.
- Compound assignment statement: It can only handle the case where the statement is $idx\% = const$, and the method of judgment is the same as that of $idx = expr\%const$.
- Conditional statement: It can only be used when the statement conditions are $idx < const$, $idx <= const$, $idx > const$ and $idx >= const$. When the condition is $idx < const$, if the array length len in Condition E of Formula (8) is also a constant and the constant $const$ in the statement is less than or equal to all array lengths len, then Condition E holds, while Conditions A and C do not hold. When the condition is $idx <= const$, it is judged whether $const$ is smaller than array length len; $idx > const$, whether $const$ larger than –1; $idx >= const$, whether $const$ larger than 0.

Constraint solving: The condition $c_i \rightarrow (idx\ op\ expr)$ is directly taken as a constraint, and the inverse of the constraint (namely $!(cond \rightarrow idx < size)$) is given to the constraint solver to determine the satisfaction of the condition. If the result of constraint solving is UNSAT (unsatisfiable), the original constraint $c_i \rightarrow (idx\ op\ expr)$ is always true, namely that the current statement S_i implies $idx\ op\ expr$; if the result is SAT (satisfiable), the original constraint $c_i \rightarrow (idx\ op\ expr)$ is unsatisfiable. In regard to different types of statements, the specific processing rules are as follows:

- Assignment statement: As the fourth column of Table 1, the assignment statement $idx = expr$ and the to-be-checked array boundary condition, $idx < len / idx >= len / idx < 0 / idx >= 0$, constitute constraints, such as $!(idx == expr \rightarrow idx < len)$, which are then processed by the constraint solver.

- Compound assignment statement: As the fourth column of Table 1, the compound assignment statement $idx\ op = expr$ and the to-be-checked array boundary condition, $idx < len$, compose the constraint $!(idx_1 == idx_0\ op\ expr \rightarrow idx_1 < len)$ which is then processed by the constraint solver.
- Conditional statement: When statements with "if", "for" or "while" are encountered, the condition $expr$ in the statement and the to-be-checked array boundary condition $idx < len$ will be extracted to form the constraint $!(expr \rightarrow idx < len)$ which is then processed by the constraint solver.

Loop out-of-bounds checking: If the check on the corresponding array boundary cannot be found in the "for" or "while" condition, then whether the array index is a loop variable will be checked. If the"for" or "while" condition matches the mode $idx < var$, the conditions in Formula (8) will be updated by replacing idx with var in the subsequent data flow analysis. In other words, array index out-of-bounds is transformed into loop out-of-bounds for further checking.

When the analysis terminates, a warning of array index out-of-bounds will be given, mainly involving the information about each array index: files, line numbers, functions and their array expressions, and boundary checking conditions to be added. The detailed information can help programmers locate and confirm the warnings of array index out-of-bounds reported by tools in a more convenient and quicker manner, which can also be taken as repair recommendations for programmers.

Case analysis: As the code in Figure 2, with regard to the array access statement $arr[2]$, the array index 2 is untainted, while the array length j, k are tainted. Then it is determined as a defect of array index out-of-bounds according to Rule 1. For array access statements $s.noisy[n]$, $s.arr[i]$ and $tmp[i]$, the array indexes n and i are tainted. Backward data flow analysis will be employed to check array index out-of-bounds according to Rule 3. Because the arrays $noisy$, arr and tmp in the structure are of unsigned types, the indexes of the arrays must be no less than 0, and it is only necessary to check whether the upper bounds of the arrays are exceeded. As indicated by the CFG in Figure 5, the checking of array index out-of-bounds starts from the bottom basic block $Block2$ containing array expressions, namely from Line 16 in the source code. First of all, the basic block $Block2$ will be traversed, and no check on array index boundaries is found. Then the analysis continues upwards, and $Block4$, the predecessor block of $Block2$, is obtained. Subsequently, the successors ($Block2$ and $Block3$) of $Block4$ are acquired, and the array information to be checked in $Block4$ is gained, namely that $OutState$ is $i < 15$ of $s.arr[i]$ and $i < 3$ of $tmp[i]$ in Line 16. As $Block2$ is on the false branch of $Block4$, the if condition is $!(i >= 15)$, and $i < 15$ can be deduced. Therefore, it is found that the $s.arr[i]$ of Line 16 should satisfy the boundary check of $i < 15$, and the $s.arr[i]$ of Line 16 will be removed from the array set to be checked. In other words, $InState$ of $Block4$ is $i < 3$ of $tmp[i]$ in Line 16. After that, the analysis continues upwards to $Block5$. When the "for" statement is encountered in $Block5$, i of $tmp[i]$ in Line 16 happens to be a loop variable, so it is converted into a problem of loop out-of-bounds. In other words, it is checked whether the upper bound n of the loop exceeds the length of the array tmp above the loop, namely checking $n < 4$. The to-be-checked array information $OutState$ in $Block6$ is $n < 4$ of $tmp[i]$ in Line 16 and $n < 12$ of $s.noisy[n]$ in Line 11. When the assignment statement in $Block6$ is encountered, the array information to be checked will be updated to $m < 4$ of $tmp[i]$ in Line 16 and $m < 12$ of $s.noisy[n]$ in Line 11. Therefore, when the entry of the function f is encountered, the array information to be checked is not null. If the configured checking depth is 1, a warning of array index out-of-bounds will be issued; otherwise, backward data flow analysis between procedures will be performed.

In the function $main$, according to the actual parameters, the array information to be checked is updated as $argc < 5$ of $tmp[i]$ in Line 16 and $argc < 13$ of $s.noisy[n]$ in Line 11. The condition of the "if" statement is $argc + 2 < 15$. When the method of simple matching is employed, the array-boundary checking condition $n < 12$ of $p.noisy[n]$ in Line 11 can be satisfied. As such, the warning "$test.c$, $line$ 11, $p.noisy[n]$, $n < 12$" raised in simple matching is false, but the upper bound n of the loop in Line 12 should be less than 11 to ensure $i < 10$ of $arr[i]$ in Line 15. Then the warning "$test.c$, $line$ 12, $arr[i]$, $n < 11$" in simple matching and constraint solving are true.

Consequently, when simple matching is adopted for judgment, when the processable modes cannot be matched, the following will be reported:

```
test.c, line 11, s.noisy[n], n < 12;
  test.c, line 16, tmp[i], i < 3;
```

When constraint solving is employed for judgment, $!((argc+2 < 15) \rightarrow (argc < 13)) \equiv UNSAT$, $!((argc + 2 < 15) \rightarrow (argc < 5) \equiv SAT$ can be obtained, and then the following will be reported:

```
test.c, line 16, tmp[i], i < 3;
```

3 Implementation and Experimental Evaluation

This paper extends our previous work[16, 17] and implements a fully automatic cross-platform static analysis tool, Carraybound, for the checking of array index out-of-bounds defects. On-demand taint analysis is optimized and on-demand pointer analysis is added, so as to analyze the range of array length. Z3 theorem prover[15] is used to address the constraint solving problem in the checking process of array index out-of-bounds. The architecture of Carraybound is illustrated in Figure 6. The tool can run on Linux and Windows, with the bottom layer depending on Clang 3.6 and the constraint solver Z3, comprising modules of array-length range analysis, on-demand taint analysis and checking of array index out-of-bounds defects. We design configurable setting to enable users to adjust the accuracy of checking on demand. Users can configure the layers of function calls to control the depth of interprocedural data flow analysis and improve the efficiency of the tool through optimization of memory and solving, etc.

3.1 Optimized implementation

Memory optimization: Large-scale programs usually contain a large number of AST files. If all AST files are read in at once, including all their contents, a large part of memory will be consumed, seriously restricting Carraybound's scalability for large-scale programs. For example, PHP-5.6.16 contains 250,000 lines of source code and 211 AST files. When we try to read all AST files at once, they will fail to run on a machine with 2 GB memory. To support programs that scan 100,000 lines or even 1 million lines of code with limited memory resources, we implemented a strategy of memory optimization in Carraybound. The key idea of this strategy is to maintain an first-in-first-out AST queue to keep only the latest used AST files in memory. For example, only 200 AST files are kept, and fewer AST files results in smaller consumption of memory. In addition, the maximum capacity of the AST queue can be configured by users according to requirements and computer capacity. When analyzing AST contents, Carraybound will first check whether the corresponding AST is in memory. If it is in memory, Carraybound will move the AST to the end of the queue; if it is not in memory, Carraybound will read AST contents from the AST file. When the AST queue reaches its maximum capacity, Carraybound will remove the AST that read first. It should be noted that if

users set a smaller maximum AST number, Carraybound will read AST files more frequently. Therefore, if enough memory is available, users should choose a larger maximum AST number to avoid frequent read operations, thereby enhancing the efficiency of Carraybound.

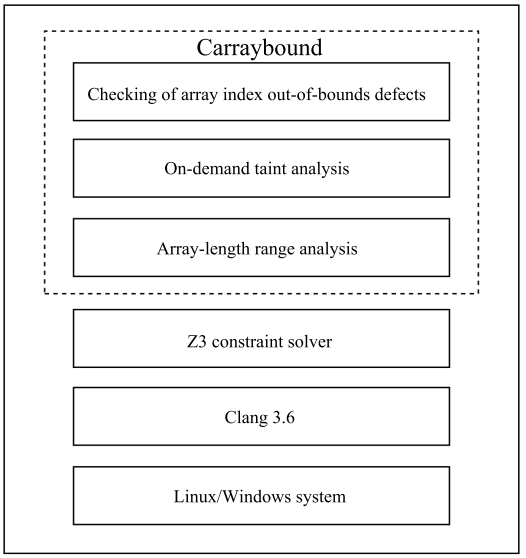


Figure 6 Architecture of Carraybound

Solving optimization: Constraint solving is time-consuming; especially frequently calling the constraint solver will seriously increase the analysis time and restrict the expandability of tools. However, our method needs to calculate fixed points, which will enlarge the demand for solving the same constraints. Accordingly, in light of the characteristics of Carraybound, we made special optimization when using it.

- Result caching: The result of whether the statements in the function imply array boundary checking will be saved as a list to reduce the calls of the constraint solver. The list is queried first. If no result is found, then the constraint solver is called, which can greatly reduce the number of calls of the constraint solver.
- Fast solving: When the encountered expression implying array boundary checking is judged through data flow analysis, constraints including $!(cond \rightarrow idx < size)$ are always solved by the constraint solver, and the constraint satisfies the condition that the object idx must be included in $cond$ to make the constraint UNSAT. Therefore, to assist the constraint solver in solving the problem faster, we complete the fast solving of the constraint ahead of time by comparing the mapping tables and filter out the constraints dissatisfying the condition that idx must be included in $cond$, greatly reducing the calls of the constraint solver.
- Time limitation: Bitwise operations and other operational statements may be observed in the program, and Z3 may take time to solve the constraint. Therefore, a configuration item, $timeout$, is provided for Z3.

3.2 Experimental evaluation

The experimental evaluation of Carraybound mainly tries to answer the following questions:

Q1: How effective is Carraybound?

Q2: How efficient is Carraybound?

Q3: What is the comparison result between Carraybound and existing methods/tools?

Q4: How is Carraybound's ability to discover common vulnerabilities and exposures (CVE)?

3.2.1 *Objects and tools*

To evaluate the effectiveness of Carraybound, we selected several common open-source projects as experimental objects, as shown in Table 2. As no available tools are found in the related literature, we compare it with the following well-known static analysis tools capable of checking array index out-of-bounds defects, including open-source tool Cppcheck[18] and commercial tools Checkmarx[19] and HP Fortify[20]. Cppcheck is an open-source static analysis tool for C/C++ language, which mainly checks program defects related to undefined behaviors, including security problems such as division by zero, integer overflow and out-of-bounds access[18]. Checkmarx is a static analysis tool based on source code. With regard to the tested program, the tool will first construct a logic graph according to the code elements and process information and then find suspected security vulnerabilities and business logic problems in the program by querying the graph. Lastly, HP Fortify is a rule-based static source code analysis tool, which supports vulnerability analysis of 25 programming languages. Statistics on the number of warnings of these tools are presented in Table 2. CAB-Simple indicates simple matching between assignment statements, while CAB-Z3 represents constraint solving. The maximum size of the program can reach over 2 million lines. We manually confirm the warnings of array index out-of-bounds reported by Carraybound in Table 2 by reviewing the source code of the program, and the confirmation process is a two-layer function call by default. Since the memory consumption of the other static analysis tools is hard to be counted, we only collect their time cost and add two large-scale programs to Table 2, as shown in Table 3. We do not know how many real array index out-of-bounds defects exist in the program in Table 2 and fail to confirm all warnings manually. As a result, to answer Q4, we read the report related to buffer overflow in CVE and find several programs with buffer overflow caused by array index out-of-bounds defects and their repaired versions, as shown in Table 4.

3.2.2 *Q1 effectiveness*

We count the false positives of CAB-simple and CAB-Z3 as 29.2% and 16.3%, respectively. The main reason for the false positives of CAB-Z3 is some calls of library functions cannot be handled. When calls of library functions are in conditional or assignment statements and guarantee the array boundary, false positives will be induced if a judgment cannot be made. Compared with CAB-Z3, CAB-simple has more false positives because it simply matches the statements with fixed formats and requires a specific position in the statement to be a constant; it fails to deal with many complex situations, resulting in false positives. In contrast, CAB-Z3 can enhance the judgment of conditions. Besides more linear constraints, it can even deal with nonlinear constraints.

3.2.3 *Q2 efficiency*

To evaluate the analysis efficiency of Carraybound, we counted the analysis time and memory consumption of the programs as shown in Table 3. CAB-Z3 consumes more time and memory than CAB-simple because of calling the constraint solver, but the time increased by 1.53% and the memory by 0.86% on average. As such, CAB-Z3 does not cause apparent increase in time and memory consumption. On the one hand, it is due to the storage and reuse of the results of constraint solving, avoiding redundant operations; in addition, special optimization is performed with regard to solving the $|expr\ 1| \rightarrow |expr\ 2|$ constraint, reducing the calls of the constraint solver. On the other hand, because CAB-Z3 can accurately judge whether the program statement has checked the array boundary, it can remove the array boundary checking that has

been satisfied as soon as possible, thus saving the overall overhead. As shown in Table 3, both CAB-simple and CAB-Z3 show a near linear increase in time and memory consumption with the expansion of program scale, indicating the good expandability of our method.

Table 2 Warnings of Carraybound and the compared tools

Program	Scale (KLOC)	CAB-simple		CAB-Z3		Cppcheck		Checkmarx		Fortify	
		W	T	W	T	W	T	W	T	W	T
libco-v1.0	6.0	3	2	3	2	0	0	10	0	0	0
libfreenect-0.5.7	34.7	10	10	10	10	0	0	10	0	–	0
vips-8.7.4	167.7	49	42	47	42	0	0	100	0	5	0
coreutils-8.30	206.8	12	5	10	5	0	0	305	1	35	2
curl-7.63.0	233.7	30	16	15	15	0	0	144	4	88	3
libxml2-2.9.9	2,302.4	9	5	6	5	0	0	96	0	–	0
Total	4,772.2	113	80	92	79	0	0	617	5	128	5

Note: "W" refers to the number of warnings reported by the corresponding tool. "T" represents the number of true warnings reported by the tool. "–" indicates unavailable data because an error occurs when Fortify is used to scan these programs.

Table 3 Time and memory consumption of Carraybound and the compared tools

Program	Scale (KLOC)	Number of AST files	CAB-simple time (s)	CAB-Z3 time (s)	Cppcheck time (s)	Checkmarx time (s)	Fortify time (s)	CAB-simple memory (MB)	CAB-Z3 memory (MB)
libco-v1.0	6.0	6	0.2	0.3	0.8	84	22	33	44
libfreenect-0.5.7	34.7	17	0.6	0.8	14.5	828	–	55	66
vips-8.7.4	167.7	411	110	117	4,633	1,364	504	3,866	3,874
coreutils-8.30	206.8	393	39	36	5,646	6,001	976	1,225	1,234
curl-7.63.0	233.7	179	11	12	466	1 191	429	556	565
vim-8.1.0818	838.6	81	142	140	434	1,483	–	1,785	1,797
espruino-2.01	1,141.6	97	5	9	539	1,801	3 403	336	349
libxml2-2.9.9	2,302.4	50	171	171	30	12,720	108	2,093	2,105

Note: "–" indicates unavailable data because an error occurs when Fortify is used to scan these programs.

Table 4 Results of checking programs with known out-of-bound CVEs and the repaired programs

Program	CVE	Repaired	CAB-Z3		Cppcheck		Checkmarx		Fortify	
			Faulty	Repaired	Faulty	Repaired	Faulty	Repaired	Faulty	Repaired
file(1)(9611f3)	CVE-2017-1000249	(35c94d)	Yes	No	No	No	No	No	No	No
openjpeg-1.5.0	CVE-2012-3535	1.5.1	Yes	No	No	No	No	No	Yes	Yes
sendmail-8.12.7	CVE-2002-1337	8.12.8	Yes	No	No	No	No	No	Yes	No

Note: "(9611f3)" and "(35c94d)" indicate the serial number of submission by the program git.

3.2.4 *Q3 comparison with existing methods*

As shown in Tables 2 and 3, some tools cannot be designated to only checking array index out-of-bounds defects, leading to higher time cost. We will not compare the efficiency of Carraybound with them, and they are listed here only for reference.

- Cppcheck: As shown in Figure 3, Cppcheck does not report any warnings related to array index out-of-bounds. However, experiments demonstrate simple array index out-of-bounds similar to "$char a[5]; a[5] = 0;$" can be reported by the tool. It indicates that this tool may be subject to false negatives. Additionally, the tool does not issue warnings related to array index out-of-bounds for the tested programs listed in Table 2. As indicated in Table 3, because the tool cannot be designated to only checking array index out-of-bounds, it will take a long time to check the corresponding defect types.
- Checkmarx: As shown in Figure 3, Checkmarx did not report any warnings about array index out-of-bounds. The tested programs listed in Table 2 are reported with a total of 617 warnings related to array index out-of-bounds. Among them, there are 25 high-risk warnings, one of which is manually confirmed to be a suspected defect of array index

out-of-bounds; the rest are low-risk ones, four of which are manually confirmed to be suspected array index out-of-bounds defects. When the report is confirmed manually, it is found that the tool cannot deal with the access problem of array index related to loop. Even if the array index is a loop variable, when the upper bound of the loop is that of the array, the array index out-of-bounds will still be falsely reported. As indicated in Table 3, the tool will still consume a long time when only a few defect types related to array index out-of-bounds are selected for checking. We find that it takes a long time to parse the source code to generate an intermediate representation of the tool, such as a logic graph, and then queries on the graph to check defects. Although only the defect types related to array index out-of-bounds are specified, the process of producing the logic graph is aimed at all types of defects, consuming a long time.

- HP Fortify: From Figure 3, Fortify only reports that $p.noisy[n]$ in Line 11 of test.c is the warning point of buffer overflow, which is actually a false positive; it also has a false negative about that $arr[i]$ in Line 15 of test.c will lead to array out-of-bounds/buffer overflow due to the loop in Line 12. With regard to the tested programs in Table 3, the tool reported massive warnings of buffer overflow, and those related to array index out-of-bounds are manually screened out. Manual confirmation reveals most of them are false positives, and in most cases, it cannot deal with the access problems of array index related to loops.

3.2.5 *Q4 comparison between reports on known CVEs*

As shown in Table 4, Carraybound can report the corresponding warnings of array index out-of-bounds in the faulty programs, but will not deliver the report in the repaired programs. Cppcheck and Checkmarx do not report the corresponding warnings of array index out-of-bounds for programs before and after repair. Moreover, HP Fortify gives the correct report in the Sendmail program before and after repair; however, it does not report on the file program before and after repair and fails to check that the defect has been repaired in the openjpeg program.

3.3 Discussion

The above experimental results demonstrate the existing open source and commercial static analysis tools are not specially designed for checking array index out-of-bounds, and they can help programmers find various types of defects in programs. However, these tools do not carry out accurate data flow analysis and constraint solving, resulting in massive false positives and false negatives for array index out-of-bounds.

Our tool performs array-length range analysis, on-demand taint analysis, accurate data flow analysis and constraint solving, with fewer false positives and false negatives. However, when manually confirming the warnings reported by these static analysis tools, we also find some shortcomings in the implementation of Carraybound, mainly involving expandability and accuracy.

Expandability: Because constraint solving is time-consuming, especially for some complex constraints, such as bitwise operations, the constraint solver fails to give the solution in a short time, limiting the expandability of our tool. Then in the experiment, timeout time is required to skip some complicated constraints, but it may lead to false positives and false negatives.

Accuracy: The main problems affecting the accuracy of Carraybound include the following:

(1) Type conversion: C programs are often observed with type conversion. Our current tool implementation fails to deal with this problem well, which may lead to false positives and false negatives.

(2) Complex loop out-of-bounds: In some cases, it is difficult to analyze the relationship between the array index and the upper bound of the loop, resulting in false positives and false negatives from the tool.
(3) Library function: In light of static analysis, we cannot judge the functions of these library functions without obtaining the source code implementation of them, but it is possible that these library functions check and guarantee the array boundary. As such, false positives may be induced in our tool.
(4) Complex array index: There are some cases in which complex expressions are taken as array indexes in programs, which will cause false positives. Especially for the simple matching method, it is easy to produce false positives because it cannot match the conditions of out-of-bounds checking. For example, with regard to the array index $2 \times i + j$, there may be range constraints on i and j respectively in the program, and the checking statements, such as $2 \times i + j < xx$, cannot be directly matched, leading to false positives. Considering expandability, existing tools set timeout time for solving constraints, so complex array indexes will also lead to false positives and false negatives of the constraint solving methods. For example, if the array index is an expression containing bitwise operations, the constraints in the decision rules will be more complex, which cannot be solved within the specified time, thereby inducing false positives.

4 Related Work

4.1 Taint analysis

Dynamic taint analysis is a popular method of software analysis, and substantial work tracks hidden vulnerabilities in software by dynamic taint analysis[10, 21–23]. Taint analysis takes the external input that may contain malicious data as the taint source, such as network packets; then, it tracks how the tainted data spreads in the whole execution process of programs; when sensitive data (such as the return address in the stack or setting of user's privilege) is affected by taint data, it will execute corresponding processing operations.

Compared with dynamic taint analysis, static taint analysis tracks the taint information in source code or binary files in a static manner. STILL[24] is a defense mechanism based on static taint and initialization, which can detect the malicious code embedded in data flow in various Internet services (such as a Web service). To reduce the overhead of taint analysis, TaintPipe[9] generates compact control flow information with the help of lightweight runtime logs and make use of multiple threads to perform symbolic taint analysis in parallel in a pipelined manner. Another problem confronting static taint analysis is manpower consumption. Most existing tools for static taint analysis will report errors in potentially vulnerable locations of programs, which will require developers to manually confirm them, resulting in a huge cost of labor. Ceara *et al*.[25] proposed a taint-dependent sequential operator, which was mainly based on fine-grained taint analysis of data flow and control flow, providing programmers with information relevant to the path to be analyzed.

4.2 Pointer analysis

Andersen's algorithm[26] and Steensgaard's algorithm[27] are the most representative ones for flow-insensitive pointer analysis. Andersen's pointer analysis[26] is a classical inclusion-based algorithm for pointer analysis in C programs, which is considered to be the most accurate algorithm for flow-insensitive and context-insensitive pointer analysis. The algorithm describes the direct points-to relation in the program as a set of constraints between variables and objects and then calculates the indirect points-to relation by solving the transitive closure of the constraint set, thus obtaining a complete set of point-to relations for all variables. This inclusion-based

idea is extensively applied to the subsequent pointer analysis[28]. Steensgaard's algorithm[27] is based on equivalence for pointer analysis, and its complexity is close to linear. However, flow-insensitive pointer analysis will affect the accuracy of subsequent static analysis. At the moment, there are also algorithms for flow-sensitive pointer analysis, which are usually based on data flow analysis[29], such as Emami algorithm[30], Lam algorithm[31] and Chase algorithm[32]. In this paper, the flow-sensitive and context-sensitive pointer analysis is on-demand; only the concerned array names are analyzed.

4.3 Checking of array index out-of-bounds

Xu *et al*.[33] proposed a method that can directly analyze untrusted machine code, which depends on the type state and linear constraints of the initial input of these programs. Detlefs *et al*.[34] proposed a static checker for common errors in programs, including array index out-of-bounds, null pointer dereference and concurrent errors in multithreaded programs. This method relies on linear constraints to automatically synthesize loop invariants for boundary checking. Leroy and Rouaix[35] put forward a theoretical model to systematically put type-based runtime checking into the interface program of host code. Kellogg *et al*.[36] proposed a lightweight verification method to check array index out-of-bounds at compile time, but this method requires developers to annotate relevant information in advance, such as program boundaries, to achieve linear verification time. In contrast, our method can identify the program boundaries. ABCD[37] is used to eliminate unnecessary array boundary checking on demand. It can delete 45% of instructions on dynamic boundary checking on average and sometimes witness near ideal optimization.

There are also numerous static tools for checking array index out-of-bounds[38–40]. Chimdyalwal[8] evaluated five static analysis tools for checking array index out-of-bounds, including commercial tools Polyspace and Coverity, academic tool ARCHER, and the other two open-source tools UNO and CBMC. Polyspace is the only tool without false negatives, but it cannot be extended to large-scale programs with the same high precision due to memory-intensive analysis. By contrast, Coverity can support the analysis of code with millions of lines, but massive false positives are encountered. UNO has both false positives and false negatives and cannot be applied to large-scale programs. ARCHER claims to run on code with millions of lines, but the analysis is not thorough enough. Finally, the CBMC model checker carries out accurate analysis, but cannot achieve the same accuracy in large-scale programs. Nguyen *et al*.[39] proposed a static checking method for array index out-of-bounds in Fortran language. Arnaud *et al*.[38] put forward a static analysis method for checking array index out-of-bounds in embedded programs. The size of programs handled by this method is over 200,000 lines, while our method can deal with the programs with millions of lines. The tool CGS provided in this literature is a closed-source customized tool on the basis of the NASA program, which takes the NASA closed-source program as the tested object. As such, the comparison with this method is not performed in the experiment.

4.4 Checking of buffer overflow for array index out-of-bounds

A large amount of work focuses on checking buffer overflow. Most of the work can check array index out-of-bounds while examining buffer overflow.

Tance[41] proposed a combinatorial approach in black-box testing to detect buffer overflow vulnerabilities. Dinakar *et al*.[42] proposed to reduce the running overhead for dynamic checking of array index out-of-bounds in C/C++ programs by fine-grained partitioning of memory. The methods based on instrumentation, such as Loginov[43] and rtcc[44], can detect whether buffer overflow occurs at runtime. However, these methods will bring extra runtime overhead, compromising testing efficiency. For instance, the extra overhead of Loginov is as high as 900%.

SafeC[45], Cyclone[46] and DangDone[47] apply extended pointer representation, containing the basic information and size of the legal target object of each pointer value. Using these pointers requires substantial modifications to the program to use external libraries, and the external library functions are usually packaged methods for converting pointers. In addition, writing such a package may be difficult to implement for indirect function calls and functions accessing global variables or other pointers in memory.

Prevention technology is a method to prevent array indexes from being used out of bounds. For example, StackGuard[2] may terminate the process after detecting that the return address on the stack is overwritten. The existing methods of runtime prevention have significant runtime overhead. In addition, these methods will take effect after potentially vulnerable programs are deployed. CFI[48] checks whether the control flow of the program is hijacked during execution. This is in contrast to our work which aims to discover array index out-of-bounds in programs before deployment.

4.5 Fuzzing test for array index out-of-bounds

Fuzzing test is one of the most used methods of black-box testing for security testing, which also plays an important role in detecting array index out-of-bounds or buffer overflow[49–57]. It mainly checks array index out-of-bounds through program crash. Fuzzing test usually starts with one or more legal inputs and then randomly changes these inputs to obtain new test inputs. Advanced fuzzing test[50] is a generation-based technique. To solve the problem of input generation of programs with complex input structure, it defines valid input by input reduction based on syntax. Godefroid *et al*.[51] proposed an alternative white-box fuzzing test method, which combines symbolic execution and dynamic test generation. Although the fuzzing test can detect array index out-of-bounds, a major limitation is low code coverage. In addition, some errors induced by array index out-of-bounds may be only reading out-of-bounds areas, so they will not cause crashes. Then the monitor in the fuzzing test may not detect this situation[52]. Our method is based on static analysis, which can achieve high code coverage and check different types of array index out-of-bounds.

5 Conclusion and Future Work

In this paper, a static checking method of array index out-of-bounds based on taint analysis is proposed, and an automatic static analysis tool, Carraybound, which can run on Windows and Linux systems, is implemented. If array index out-of-bounds are encountered in the program, we will report the corresponding array position and to-be-added array boundary conditions. We evaluate the Carraybound tool by scanning the source code of real programs. Experimental data demonstrates that Carraybound can quickly report array indexes without array boundary checking in the program, and the rate of false positives is about 16.3% when the constraint solving method is used. Although Carraybound has some false positives and false negatives, it can significantly reduce the manual review work of programmers. Our method can provide the to-be-added conditions and positions of array boundary checking and help programmers locate and confirm the reported warnings of array index out-of-bounds in a more convenient and quicker manner. It can also serve as a repair recommendation for programmers.

At the moment, Carraybound may cause false positives due to library functions and other reasons. When the library function has source code, function summary and other techniques can be employed to check array index out-of-bounds with higher accuracy. When the library function has no source code, dynamic testing can be combined for checking. In addition, the defect of array index out-of-bounds is a special type of buffer overflow, the checking of which can be extended to that of buffer overflow. For common APIs related to buffer overflow, such as

strcpy and memcpy, their overflow conditions can be defined and summarized to build a buffer overflow model. Then data flow analysis is adopted to detect whether there are corresponding statements for out-of-bounds checking in the program, thereby detecting buffer overflow defects.

References

[1] CWE. Improper validation of array index. https://cwe.mitre.org/data/definitions/129.html

[2] Cowan C, Pu C, Maier D, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q, Hinton H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. Proc. of the USENIX Security Symp. 1998, 98: 63–78.

[3] CVE. http://www.cvedetails.com/vulnerabilities-by-types.php

[4] Ye T, Zhang L, Wang L, Li X. An empirical study on detecting and fixing buffer overflow bugs. Proc. of the IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST). IEEE, 2016. 91–101.

[5] Gao F, Wang L, Li X. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2016. 786–791.

[6] Bao T, Gao F, Zhou Y, Li Y, Wang L, Li X. Automatically validating static buffer overflow warnings based on guided symbolic execution. Journal of Cyber Security, 2016, (2): 46–60.

[7] Wang L, Li F, Li L, Feng XB. Principle and practice of taint analysis. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 860–882. http://www.jos.org.cn/1000-9825/5190.htm [doi: 10.13328/j.cnki.jos.005190]

[8] Chimdyalwar B. Survey of array out of bound access checkers for C code. Proc. of the 5th India Software Engineering Conf. ACM, 2012. 45–48.

[9] Ming J, Wu D, Xiao G, Wang J, Liu P. TaintPipe: Pipelined symbolic taint analysis. Proc. of the 24th {USENIX} Security Symp. ({USENIX} Security 15). 2015. 65–80.

[10] Newsome J, Song DX. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Proc. of the Network and Distributed System Security Symp. (NDSS). 2005, 5: 3–4.

[11] Khedker U, Sanyal A, Sathe B. Data Flow Analysis: Theory and Practice. CRC Press, 2009.

[12] Kildall GA. A unified approach to global program optimization. Proc. of the 1st Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. ACM, 1973. 194–206.

[13] Galler SJ, Aichernig BK. Survey on test data generation tools. Int'l Journal on Software Tools for Technology Transfer, 2014, 16(6): 727–751.

[14] De Moura L, Bjørner N. Z3: An efficient SMT solver. Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer-Verlag, 2008. 337–340.

[15] Z3 theorem prover. https://z3.codeplex.com/

[16] Gao F, Chen T, Wang Y, Situ L, Wang L, Li X. Carraybound: Static array bounds checking in C programs based on taint analysis. Proc. of the 8th Asia-Pacific Symp. on Internetware. ACM, 2016. 81–90.

[17] Zhou Y. Extensible framework for static vulnerability detection based on taint analysis [Ph.D. Thesis]. Nanjing: Nanjing University, 2017.

[18] Cppcheck. http://cppcheck.net/

[19] Checkmarx. https://www.checkmarx.com/

[20] Fortify static code analyzer. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

[21] Costa M, Crowcroft J, Castro M, Rowstron A, Zhou L, Zhang L, Barham P. Vigilante: End-to-end containment of Internet worms. ACM SIGOPS Operating Systems Review, 2005, 39(5): 133–147.

[22] Crandall JR, Su Z, Wu SF, Chong FT. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: Proc. of the 12th ACM Conf. on Computer and Communications Security (CCS). ACM, 2005. 235–248.

[23] Suh GE, Lee JW, Zhang D, Devadas S. Secure program execution via dynamic information flow tracking. ACM SIGPLAN Notices, 2004, 39(11): 85–96.

[24] Wang X, Jhi YC, Zhu S, Liu P. Still: Exploit code detection via static taint and initialization analyses. In: Proc. of the 2008 Annual Computer Security Applications Conf. (ACSAC). IEEE, 2008. 289–298.

[25] Ceara D, Mounier L, Potet ML. Taint dependency sequences: A characterization of insecure execution paths based on input- sensitive cause sequences. Proc. of the 3rd Int'l Conf. on Software Testing, Verification, and Validation Workshops. IEEE, 2010. 371–380.

[26] Andersen LO. Program analysis and specialization for the C programming language [Ph.D. Thesis]. University of Cophenhagen, 1994.

[27] Steensgaard B. Points-to analysis in almost linear time. Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM, 1996. 32–41.

[28] Chen C, Huo W, Yu H, Feng X. A survey of optimization technology of inclusion-based pointer analysis. Jisuanji Xuebao/Chinese Journal of Computers, 2011, 34(7): 1224–1238.

[29] Pang L, Su X, Ma P, Zhao L. Research on flow sensitive demand driven alias analysis. Journal of Computer Research and Development, 2015, 52(7): 1620–1630.

[30] Emami M, Ghiya R, Hendren LJ. Context-sensitive interprocedural points-to analysis in the presence of function pointers. ACM SIGPLAN Notices, 1994, 29(6): 242–256.

[31] Wilson RP, Lam MS. Efficient context-sensitive pointer analysis for C programs. ACM SIGPLAN Notices, 1995, 30(6): 1–12.

[32] Chase DR, Wegman MN, Zadeck FK. Analysis of pointers and structures. ACM SIGPLAN Notices, 1990, 39(4): 343–359.

[33] Xu Z, Miller BP, Reps T. Safety checking of machine code. ACM SIGPLAN Notices, 2000, 35(5): 70–82.

[34] Detlefs DL, Leino KRM, Nelson G, Saxe JB. Extended Static Checking. 1998. [doi: 10.1007/978-0-387-35358-6_1]

[35] Leroy X, Rouaix F. Security properties of typed applets. Secure Internet Programming. Berlin, Heidelberg: Springer-Verlag, 1999. 147–182.

[36] Kellogg M, Dort V, Millstein S, Ernst MD. Lightweight verification of array indexing. Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2018. 3–14.

[37] Bodík R, Gupta R, Sarkar V. ABCD: Eliminating array bounds checks on demand. ACM SIGPLAN Notices, 2000, 35(5): 321–333.

[38] Venet A, Brat G. Precise and efficient static array bound checking for large embedded C programs. ACM SIGPLAN Notices, 2004, 39(6): 231–242.

[39] Nguyen TVN, Irigoin F. Efficient and effective array bound checking. ACM Trans. on Programming Languages and Systems (TOPLAS), 2005, 27(3): 527–570.

[40] Popeea C, Xu DN, Chin WN. A practical and precise inference and specializer for array bound checks elimination. Proc. of the 2008 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation. ACM, 2008. 177–187.

[41] Wang W, Lei Y, Liu D, Kung D, Csallner C, Zhang D, Kacker R, Kuhn R. A combinatorial approach to detecting buffer overflow vulnerabilities. Proc. of the 41st IEEE/IFIP Int'l Conf. on Dependable Systems & Networks (DSN). IEEE, 2011. 269–278.

[42] Dhurjati D, Adve V. Backwards-compatible array bounds checking for C with very low overhead. Proc. of the 28th Int'l Conf. on Software Engineering (ICSE). ACM, 2006. 162–171.

[43] Loginov A, Yong SH, Horwitz S, Reps T. Debugging via run-time type checking. Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering. Berlin, Heidelberg: Springer-Verlag, 2001. 217–232.

[44] Steffen JL. Adding run-time checking to the portable C compiler. Software: Practice and Experience, 1992, 22(4): 305–316.

[45] Austin TM, Breach SE, Sohi GS. Efficient detection of all pointer and array access errors. Proc. of the ACM SIGPLAN 1994 Conf. on Programming Language Design and Implementation (PLDI). ACM, 1994. 290–301.

[46] Hicks M, Morrisett G, Grossman D, Jim T. Experience with safe manual memory-management in cyclone. Proc. of the 4th Int'l Symp. on Memory Management. ACM, 2004. 73–84.

[47] Wang Y, Gao F, Situ L, Wang L, Chen B, Liu Y, Zhao J, Li X. DangDone: Eliminating dangling

pointers via intermediate pointers. Proc. of the 10th Asia-Pacific Symp. on Internetware. ACM, 2018. 6.

[48] Abadi M, Budiu M, Erlingsson Ú, Ligatti J. Control-flow integrity principles, implementations, and applications. ACM Trans. on Information and System Security (TISSEC), 2009, 13(1): 4.

[49] Sutton M, Greene A, Amini P. Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, 2007.

[50] Godefroid P, Kiezun A, Levin MY. Grammar-based whitebox fuzzing. ACM SIGPLAN Notices, 2008, 43(6): 206–215.

[51] Godefroid P, Levin MY, Molnar DA. Automated Whitebox fuzz testing. Proc. of the Network and Distributed System Security Symp. (NDSS). 2008, 8: 151–166.

[52] McNally R, Yiu K, Grove D, Gerhardy D. Fuzzing: The state of the art. Defence Science and Technology Organisation Edinburgh, 2012. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=15CF9A7FD272D62D76FB5ED26DA3808F?doi=10.1.1.461.4627&rep=rep1&type=pdf

[53] Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. EXE: Automatically generating inputs of death. ACM Trans. on Information and System Security (TISSEC), 2008, 12(2): 1–38.

[54] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. ACM SIGPLAN Notices, 2005, 40(6): 213–223.

[55] Xu RG, Godefroid P, Majumdar R. Testing for buffer overflows with length abstraction. Proc. of the 2008 Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2008. 27–38.

[56] Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G. Driller: Augmenting fuzzing through selective symbolic execution. Proc. of the Network and Distributed System Security Symp. (NDSS). 2016, 16(2016): 1–16.

[57] Pak BS. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution [Ph.D. Thesis]. School of Computer Science, Carnegie Mellon University, 2012.

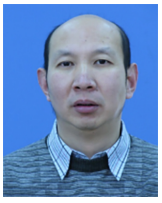
Fengjuan Gao, bachelor. Her research interests include the software engineering, program analysis, software testing and software security.

Lingyun Situ, Ph.D., assistant researcher, student member of CCF. His research interests include the software engineering, information security, static analysis and fuzzing test.

Yu Wang, bachelor. His research interests include the software engineering, program analysis, software testing and software security.

Linzhang Wang, Ph.D., professor, doctoral supervisor, distinguished member of CCF. His research interests include the software engineering, software testing and software security.

Tianjiao Chen, master. His research interest is software engineering.



Xuandong Li, doctoral supervisor, CCF Follow. His research interests include complex software modeling and analysis, software testing and verification.