DOI:10.7544/issn1000-1239.2016.********
卷(期):起止页,年

基于污点分析的数组越界检查的静态分析方法研究

高凤娟　王豫　王林章　李宣东

(计算机软件新技术国家重点实验室(南京大学),江苏 南京　210023)

通讯作者: 王林章, E-mail: lzwang@nju.edu.cn

ArrayBoundChecker: Static Array Boundary Checking in C Programs Based on Taint Analysis

Gao Fengjuan, Wang Yu, Wang Linzhang, and Li Xuandong

(State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract During the rapid development of programming languages, C/C++ language is still one of the most popular languages. However, it is prone to vulnerabilities because C programming language never performs automatic boundary checking in order to speed up execution. The situation is worse in service oriented programs, which automatically restart services when crashed. Therefore, boundary checking is absolutely necessary in any program. Because if a variable is out-of-bounds, some serious errors may occur during execution, such as endless loop or buffer overflows. When there are arrays used in a program, the index of an array must be within the boundary of the array. But programmers always miss the array boundary check or do not perform a correct array boundary check. In this paper, we perform static analysis based on taint analysis and data flow analysis to detect which tainted array indexes do not have correct array boundary checks in the program. To reduce false positives introduced by inaccurate match of the conditions of statements and the boundary conditions to be checked, we propose to check the satisfiability of the conditions by invoking a constraint solver. We have implemented an automatic static analysis tool, ArrayBoundChecker. And the experimental results show that ArrayBoundChecker can work effectively and efficiently.

Key words Array index out-of-bounds; Static analysis; Taint analysis; Data flow analysis

摘要　随着编程语言的不断发展，C/C++语言依旧是最受欢迎的编程语言之一。但是由于 C 语言为了提高程序运行效率，不会自动进行越界检测。这会让程序更容易出现漏洞，尤其是在面向服务的程序之中，程序崩溃后自动重启的特点使得更容易被攻击。以此可知，越界检测是在任何程序中都是有必要的。这是因为如果程序中有一个变量可以访问边界外的数据，那么可能在运行时导致严重的后果，例如为定义行为、死循环和缓冲区溢出等。当程序中使用了数组，访问该数组的索引必须在该数组的边界之内。但是开发者容易忽视边界检查或没有进行正确的边界检查。在本论文中，我们使用基于污点分析和数据流分析的静态分析方法检测程序中是否对由外部输入控制的数组下标索引进行了正确的边界检查。只通过简单匹配来判断程序语句是否进行了正确的

收稿日期：2016-03-16；**修回日期**：2016-04-26
基金项目：国家自然科学基金项目（xxxxxx）；
通信作者：王林章（lzwang@nju.edu.cn）
该论文前期版本发表于 Gao F, Chen T, Wang Y, et al. Carraybound: static array bounds checking in C programs based on taint analysis[C]//Proceedings of the 8th Asia-Pacific Symposium on Internetware. ACM, 2016: 81-90.

数组边界检查，会存在很多无法处理的情形，进一步会造成大量误报。因此，我们提出通过调用约束求解器来进行更加精确的判断，从而降低误报率。同时，我们实现了名为 ArrayBoundChecker 的自动静态分析工具，并通过实验展示了我们的方法的有效性。

关键词 数组越界;静态分析;程序分析;缓冲区溢出

中图法分类号 TP391

随着计算机技术的发展，人们对计算机以及软件技术的需求日益增加，软件的数目也在不断的快速增长。随着网络时代的来临，服务化、群智化和生态化已经成为当前软件开发的新趋势。近年来，研究面向服务的群智化生态化软件生态系统开发方法，增强软件生态系统的自适应演化能力已成为国内外学术界和工业界研究的前沿和重点，但如何构建健康可持续的软件生态系统仍是当前软件开发面临的重大挑战。C 语言是面向过程、抽象化的通用程序设计语言，广泛应用于底层软件生态系统的开发。软件生态系统中如果存在漏洞，可能会被恶意攻击利用，将会严重影响人们的生产生活，甚至威胁生命财产安全。软件安全已经成为软件企业不能回避的挑战。

C 语言被广泛应用于底层软件生态系统的开发是因为 C 语言编写的程序具有更高运行效率。但是为了提高程序运行效率，C 语言不会自动对边界进行有效性检查。但是越界检查是在任何程序中都是有必要的。当一个数组在程序中被使用时，访问该数组的下标索引必须在一定范围之内，即不小于 0 并且小于数组的大小，否则会造成数组下标越界。数组下标越界有两种模式，读越界和写越界。读越界会导致读取到随机的值，继而使用该值会导致未定义行为。相比之下，写越界会造成更加严重的后果，除了未定义行为，甚至是会造成控制流截取，使得攻击者可执行任意恶意代码。研究表明[34]，31%的缓冲区溢出是由数组越界造成的。缓冲区溢出漏洞是软件漏洞中常见的漏洞之一，它能为攻击者提供破坏数据、使系统崩溃甚至执行恶意代码。因此，为了提高软件生态系统的安全性，对由外部输入控制的数组下标进行边界检查是必须的。但是开发者可能会遗漏边界检查或者没有进行正确的边界检查。在此论文中，我们基于污点分析和数据流分析对污染的数组下标进行检查，以发现程序中可能存在的数组越界缺陷。

已经有学者提出静态分析方法和动态测试方法来进行数组越界检查。由于动态方法总是依赖于测试用例的完整性，这导致这些方法无法实现程序的高覆盖率。静态分析方法通过扫描分析程序源代码以检查程序中的缺陷。数据流分析通常用于静态代码分析，这是一种用于收集有关基本块的数据流信息的技术。

在本文中，我们提出了一个静态分析框架 ArrayBoundChecker，该框架基于静态的污点分析和数据流分析检查程序中是否存在潜在的数组越界缺陷。首先，ArrayBoundChecker 执行污点分析以确定所有变量的污染状态。然后，定位包含数组表达式的所有语句并构造数组边界信息。接下来，执行后向数据流分析遍历控制流图，以检查是否存在确保下标索引在数组边界内的任何语句。如果不存在这些语句，则报告数组越界警报，以及相应的数组和下标，提供待增加的数组边界检查条件，可以帮助程序员更加方便快速的定位和确认我们报告的数组越界警报，也可以作为修复推荐建议提供给程序员作为参考。

1 背景知识

1.1 控制流分析

控制流分析是一种静态分析技术，它被用来确定程序的控制流程。控制流程图（CFG）是一个有向图，其中的节点表示基本块，有向边表示控制流[3]。基本块是具有程序指令的线性序列，是一段只有一个入口和一个出口的代码块，控制图显示了事件在程序中的排序。控制流程图在静态分析技术中非常常用，控制流分析是数据流分析的基础。

1.2 调用图

要对给定的 C 程序执行静态分析，我们首先需要获取程序中程序之间的运行时关系，即程序的调用图。调用图是流程图，其中每个节点表示一个过程，每个边（f，g）表示过程 f 调用过程 g。调用图给出了程序的过程间视图。调用图是控制和数据流程序的有用数据表示，用于调查过程间通信[25]。

1.3 数据流分析

数据流分析通常用于静态代码分析，它是一种用于收集有关跨基本块的数据流信息的技术。数据流分析是基于控制流图进行的。对程序进行数据流分析的一种简单方法是为控制流图的每个节点建立数据流方程，并通过在每个节点上重复计算输入的输出来反

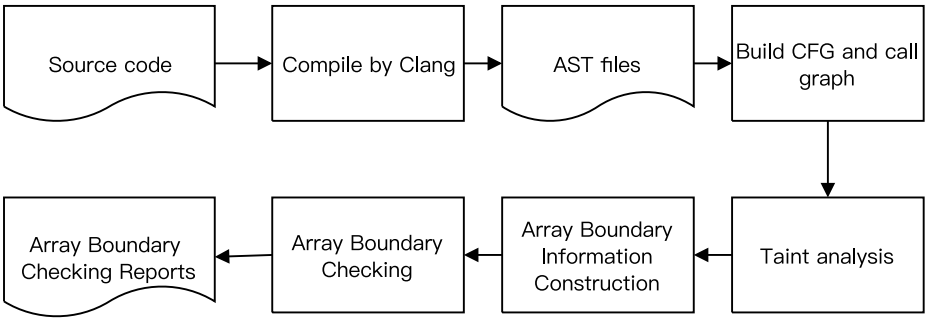

Fig. 1 The overall framework of our method.

图 1：方法框架

复求解，直到整个系统稳定为止，即到达一个固定点[18][19]。前向流分析和后向流分析是数据流分析的两种不同方法。前向数据流分析沿着正常的执行路径，从入口节点开始并在目标节点处结束。一个块的结束状态是这个块起始状态的一个函数，该函数实际上是块中语句对进入状态的影响的组合，块的进入状态则是其前驱退出状态的组合。与之相对地，后向数据流分析与控制流图中的有向边方向相反，从目标节点开始并在入口节点处结束。块的进入状态是块的退出状态的函数，该函数实际上是块中语句对退出状态的影响的组合，块的退出状态则是其后继进入状态的组合。

1.4 污点分析

污点分析是检测程序漏洞的常用技术。如果攻击者向程序输入一些恶意数据，程序可能不安全。这些受外部输入影响的数据在污染分析中标记为污染。外部输入包括用户或文件的输入、主函数的参数。污点分析尝试识别已被用户输入污染的变量，并将其追溯到易受攻击的函数。如果被污染的数据在未经净化的情况下传递到接收器，则会被标记为一个漏洞。污点分析分为静态污点分析和动态污点分析。其中动态污点分析需要执行程序，无法保证源代码的覆盖范围。静态污点分析依赖于程序抽象语法树的分析，不需要实际执行程序。因此静态污点分析可以实现比动态污点分析更高的源代码覆盖率。但静态污点分析可能因缺乏运行时信息而产生误报和漏报。

2 基于污点分析的数组越界检查的静态分析技术

我们的数组越界检查方法主要基于静态污点分析技术和数据流分析技术，它们基于程序的函数调用图和控制流程图。

我们的整个分析基于 Clang 的抽象语法树（AST）。首先，根据程序的 AST 构建函数调用图和控制流程图。我们的过程分析依赖于程序的调用图，从中获得程序中子过程之间的调用关系。然后，我们将对调用图执行深度优先搜索（DFS）分析，以生成一个没有任何递归的调用图。

如图 1 所示，首先，我们使用 Clang 编译给定程序源代码以获取其 AST 文件。然后，基于 AST 文件，构建 CFG 和函数调用图。接下来，基于 CFG 和函数调用图，我们将执行污点分析以获取所有表达式的污染状态。然后，我们将定位数组表达式并构造数组的边界信息。再然后，我们将通过后向数据流分析来执行数组越界检查。最后，我们得到完整的数组越界检查报告。

2.1 示例

在这一节中，我们将给出小例子来初步说明我们的方法。示例的控制流程图 CFG 如图 2 所示。冒号后面的数字是入口语句的行号。

```
1.  typedef unsigned int UINT32;
2.  typedef struct {
3.      UINT32 noisy[12];
4.      UINT32 arr[15];
5.  }myStruct;
6.  myStruct p;
7.  UINT32 arr[10];
8.
9.  void f(UINT32 m) {/*main->f: m<12*/
10.     UINT32 n=m;
11.     p.noisy[n]=0;
12.     for(UINT32 i=0;i<n;i++) {
13.         if(i>=15)
14.             break;
15.         arr[i]=0;
16.         p.arr[i]=0;
17.     }
18. }
19. int main(int argc, char** argv) {
20.     if(argc+2<15){
21.         f(argc-1); /*argc-1<12*/
```

```
22.     }
23.     return 0;
24. }
```

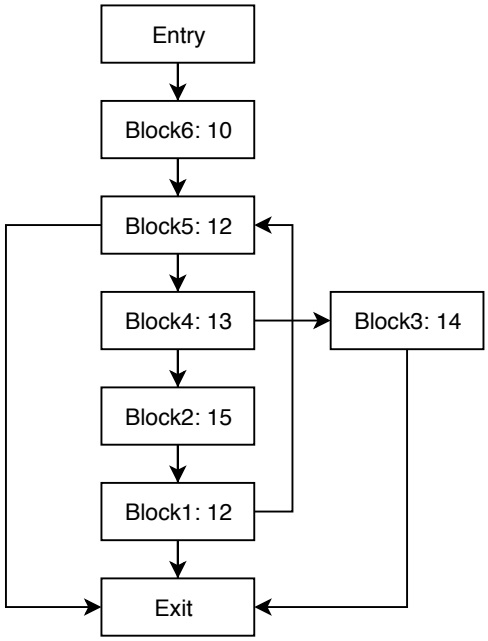

Fig. 2 CFG of the example

图 2 示例控制流程图

污点分析：通过对函数 f 进行污点分析，可以知道 f 中的变量 n 和 i 的污点值是和 f 的参数 m 一致的。然后对 main 函数进行污点分析，argc 和 argv 由外部输入，因此是污染的。由于 main 函数通过参数 argc-1 调用函数 f，导致函数 f 的形参 m 为污染的。进而，f 中的变量 n 和 i 也是污染的。

数组边界信息构造：通过遍历我们可以发现在函数 f 中，共有三个数组表达式的使用位置，即 15 行的 arr[i]、16 行的 p.arr[i]和 11 行的 p.noisy[n]。然后，我们分析可以知道三个数组表达式的数组下标为 i 和 n，其类型为 unsigned int。这时，我们将查询数组下标 i 的污染状态为污染的。接下来，我们可以分别获取到这三个数组对应的定义位置，即行号 7、4 和 3。同时我们可以分析出三个数组的大小，即 10、15 和 12。基于上述信息，我们构造可以为这三个数组表达式构造数组边界条件：15 行的 arr[i]中的 i 应满足 i<10；16 行的 p.arr[i]应满足 i<15；11 行的 p.noisy[n]应满足 n<12。经过数组边界信息构造，我们知道基本块 Block2 中有三个数组下标边界待检查。

数组越界检查：如图 2 所示的 CFG，我们从包含数组表达式的基本块 Block2 开始执行数组越界检查，也就是从源码中的第 15 行代码开始进行分析。首先，将遍历基本块 Block2，未发现对三个数组下标边界的检查。然后我们继续向上分析，得到 Block2 的前驱块 Block4。接下来，我们得到 Block4 的后继（Block2 和 Block3），从而得到 Block4 中待检查的数组信息，仍是 15 行的 arr[i]中 i<10、16 行的 p.arr[i] 中 i<15 和 11 行的 p.noisy[n] 中 n<12。由于 Block2 在 Block4 的 false 分支上，所以 if 条件为!(i>=15)。!(i>=15)可以推出 i<15，因此，找到了 16 行的 p.arr[i]应满足 i<15 的边界检查，第 16 行的条件 i<15 将从数组边界信息中移除。也就是说，Block4 的输出状态是 15 行的 arr[i]中 i<10 和 11 行的 p.noisy[n] 中 n<12。然后继续向上分析 Block5，在 Block5 中遇到 for 语句时，15 行的 arr[i] 中的数组边界信息将更新为 n<11。当遇到 Block6 中的赋值语句时，数组边界信息将更新为 15 行的 arr[i] 中 m<11 和 11 行的 p.noisy[n] 中 m<12。因此，当遇到函数 f 的入口时，数组边界信息不为空。如果配置的检测深度为 1，那么我们将报告数组越界警告。否则，将执行过程间的后向数据流分析。

数组越界检查报告：

当使用简单匹配方法对基本块中的语句进行分析时，将会报告：

test.c, line 11，p.noisy[n]，n<12;

test.c, line 12, arr[i], n<11;

当使用约束求解方法判断程序中的条件语句是否隐含数组边界检查条件时，将会报告：

test.c, line 12, arr[i], n<11;

通过分析可以发现由于 main 函数中对 argc 进行了检查，会使得调用函数 f 时的参数 argc-1 小于 12，也就是函数 f 的形参 m 小于 12。这样 f 中的 n 小于 12。则 11 行的 p.noisy[n]的数组边界检查条件 n<12 可以被满足，因此简单匹配方法中的警报“test.c, line 11，p.noisy[n]，n<12”为误报。但是 12 行的循环上界 n 应小于 11 才可以保证 15 行的 arr[i] 中 i<10，因此简单匹配方法和约束求解方法中的警报“test.c, line 12, arr[i], n<11”为真警报。

2.2 污点分析

在本文中，我们使用静态污点分析方法。我们的污点分析包括过程内污点分析和过程间污点分析。过程内污点分析对程序中的每个函数进行分析。过程间污点分析从入口函数（通常是主函数）开始分析整个程序。我们将外部输入（包括用户或文件的输入、以及主函数的参数）作为污点源。由于我们接下来需要执行数组越界检查，因此我们将包含数组使用的语句作为污点池。

过程内污点分析：过程内的污点分析是按照函数调用图自底向上的顺序进行的。首先，根据函数调用图的逆拓扑序，提取自下而上的函数序列。对于每个

函数，我们保留函数中每个变量的污点状态。每个变量的污点状态可能是被污染，无污染或与函数的参数相关。然后，我们使用前向数据流分析对每个函数执行污点分析。对于给定的函数，我们对函数中每个基本块中的每条语句执行污点分析。对于函数中的每个基本块，其 in-state 是其所有前驱基本块的 out-states 的组合，其 out-state 是其 in-state 的函数，是该基本块内所有语句作用后的结果。对于包含循环的函数，我们将迭代计算每个基本块的 in-state 和 out-state，直到该基本块的 in-state 和 out-state 的状态不再改变。函数的污点状态与函数出口基本块的 out-state 相同。这样，我们就可以得到函数内所有语句与相应函数参数之间的污点关系。并且，尽管每个函数可能会被其他函数调用多次，但是，每个函数只需要进行一次污点分析，后续只需要复用这次的函数内污点信息即可。对于 ArrayBoundChecker 遇到的每条语句，ArrayBoundChecker 将根据表 1 中列出的污点传播规则处理该语句。对于表达式，如果其中存在污染的变量，则我们认为整个表达式是污染的。对于数组，如果数组名或者数组下标有一个是污染的，则整个数组被视为污染的。结构体同理。

过程间污点分析：首先，通过对函数调用图进行深度优先搜索（DFS）遍历，去掉循环得到有向非循环图（DAG），即非递归调用图。DAG 具有拓扑排序，在该排序的顶点序列中，每一条边都直接连接前驱顶点和后继顶点。过程间污点分析从入口函数开始，根据广度优先搜索（BFS）顺序从上到下分析程序。入口函数的参数被标记为被污染的数据，每个表达式的

Table 1 Taint propagation rules

表 1 污点传播规则

Expression Name	Expression Type	Handling Method	Comments
Integer Constant	Stmt::IntegerLiteralClass	return TaintValue()	Untainted by default
Float Constant	Stmt::FloatingLiteralClass	return TaintValue()	Untainted by default
Type Transformation	CastExpr	visitStmt(subExpr)	Handle the subexpression recusively
Parenthesis Expression	ParenExpr	visitStmt(subExpr)	Handle the subexpression recusively
Declared Variable	DeclRefExpr a	return taintValue(a)	Return its taint value directly
Declaration statement	DeclStmt int a=b;	taintValue(a)=taintValue(b)	Similar to assignment statement
Unary Operator	UnaryOperator	visitStmt(subExpr)	Handle the subexpression recusively
BinaryOperator: assignment	BinaryOperator: x=y	taintValue(x) = taintValue(y)	Include +=, *=, etc
Binary Operator: others	BinaryOperator: x op y	return visitStmt(x)+visitStmt(y)	Return taint if any operand is taint
Conditional Expression	ConditionExpr	result=taintValue(b)+taintValue(c)	Similar to assignment statement
Array	ArraySubscriptExpr	Left: return visitStmt(E->getBase(),inLeft) Right: return TaintValue((*visitStmt(E->getBase(), inLeft)) +(*visitStmt(E->getIdx(), inLeft)))	Taint as a whole
Struct	MemberExpr	return visitStmt(E->getBase(),FD)	Taint as a whole
If statement	IfStmt	StmtValue=visit(IfStmt)	
While statement	WhileStmt	StmtValue=visit(WhileStmt)	
Function call	CallExpr	return vistCallExpr(callExpr,FD)	Taint from arguments to parameters
Return Expression	ReturnStmt	visitStmt(subExpr)(if not void)	return untainted when Void
Other statement		return untainted	Untainted by default

初始污点状态都设置为未被污染，每个变量的污点状态将被计算。我们根据非递归调用图上的拓扑顺序遍历每个函数，使污点数据从顶部函数的参数扩散到相关函数的参数。利用调用图中的每一条调用边，将调用者实参的污点数值传播到被调用者的参数中。如果有多个函数调用相同的函数，则被调用函数的参数污点值是其所有调用者实参的污点值的组合。这样，我们就可以获得每个函数参数的污点信息。

通过过程内污点分析，我们计算出了函数中表达式与该函数参数之间的污点关系。通过过程间污染分析，计算出了各函数参数与入口函数参数之间的污点关系。为了查询函数中表达式的污点状态，我们只需要将函数中表达式与函数参数之间的污点关系与函数参数的污点值相结合即可。

2.3 数组边界信息构造

为了检查程序是否遗漏了一些必要的数组边界检查，我们需要获取数组的使用位置以及应该检查的边界条件，以避免数组下标索引超出边界。为了构造所需的数组边界条件，我们基于程序的 AST 和上述污点分析的结果，进行了数组定位、数组下标索引提取、数组大小提取和数组边界检查条件构造。

数组定位：在程序的每个函数中都应该执行数组边界检查。给定一个函数，我们首先得到相应的控制流程图（CFG）。然后我们遍历 CFG 上每个基本块中的每条语句。如果语句类型是“ArraySubscriptExpr”，则找到一个数组的使用语句。数组基可以是结构数组，其下标也可以是数组。数组可以是结构变量的数组成员变量，也可以是多维的。

数组下标索引提取：一旦定位到一个数组，就需要构造它的边界检查条件。为此，我们首先需要提取数组的基和下标索引。在 Clang 中 API 的支持下，我们可以获取每次迭代后数组的最后一个索引，在最后一次迭代后提取数组基。数组基将用于提取数组大小，而数组索引将用来构造数组边界条件。

数组大小提取：基于 clang 的 AST，我们可以提取声明的数组大小。当数组是普通数组变量或结构数组变量时，我们可以提取该数组的声明大小。但当数组变量是外部变量时，我们无法提取数组的声明大小，因为 AST 不会保留外部变量的大小信息。一旦遇到外部数组变量，我们就会跳过它。值得一提的是，对于多维数组，我们将得到一个存放数组每一维大小的数字列表。

数组边界条件构造：基于特定数组的下标索引和大小，可以构造数组边界条件。每个下标索引应小于相应的数组大小，如 index <size。此外，如果 index 是有符号的变量，下标索引也应该不小于零，如 index≥0。在构造数组边界条件之前，我们会先查询下标索引是否被污染。 如果下标索引是常量，则将忽略该下标索引。如果下标索引被污染，我们使用下标索引和数组大小构建数组边界条件（如 index <size）。

为了支持数据流分析，我们保留了关于数组下标的更多信息，其中包括所在函数、基本块及位置。所有的这些信息将打包成一个自定义的结构，用作数据流分析的输入。

2.4 数组越界检查

本文首先从数组语句开始，对程序的控制流图进行后向数据流分析，寻找相应的数组边界检查，以确定数组语句前是否存在数组边界条件检查。如果存在相应的数组边界检查，则将从我们的目标中删除该数组边界条件。否则，将继续进行数据流分析，直到找到相应的数组边界检查，或达到入口函数，或达到用户指定的函数分析层数。

如上一节中所述，我们已经提取并构造了给定函数中每个数组下标索引的数组边界条件。接下来，我们根据程序的控制流程图进行后向数据流分析，以检查程序中是否包含了每一个数组边界条件。我们首先根据算法 1 和算法 2 对函数进行过程内数据流分析。如果没有找到数组边界检查，我们将根据算法 3，基于函数调用图进行过程间数据流分析。用户可以在配置文件中配置调用图中的遍历深度。

对于给定的函数，我们首先得到它的控制流程图 CFG。然后，我们遍历 CFG 以找到 CFG 的入口点。我们关心的是数组边界条件是否包含在任何基本块的任何语句中。每个块的 in-state 和 out-state 初始化为空集。对于每个数组边界条件，我们首先得到它对应的基本块和语句。依据算法 1，我们的过程内后向数据流分析从数组使用语句所在的基本块开始，向上遍历函数中的每个基本块，到数组边界条件所在的函数的入口点结束。如果在到达入口点之前在函数中找到了所有数组边界条件，则将终止后向数据流分析。最初分析的是数组使用语句所在的基本块，然后将分析其前驱基本块。分析完每个基本块，再继续将其前驱基本块加入到 blockSet 中。对于 blockSet 中的每个基本块，依照算法 2 进行后向数据流分析，以更新其 in-state 和 out-state。对于每个基本块，我们使用一个后序迭代算法来求解数据流方程。该步骤将重复执行，直到达到不动点，即当 in-state 和 out-state 不再改变。

依据算法 2，处理每个基本块时，我们对基本块中的语句进行后向数据流分析。对于遇到的每一条语句，我们将根据表 2 所示的规则进行处理。在这些语句的处理中，我们都提供了两种判断方法来检查数组边界检查是否被满足，即简单匹配方法和约束求解方

算法 1： 过程内后向数据流分析算法

函数：IntraBDFA(CFG, ABCSet)
输入：CFG，ABCSet
/*ABCSet 是函数内的数组信息集合，ArraryBoundaryConditionSet */
输出：ABCSet
for each block in CFG **do**
  if block->pred_begin()==block->pred_end() **then**
    Entry=block;
  end if
  InState[block]= ∅;
  OutState[block]= ∅:
end for
blockSet= ∅;
for each ABC in ABCSet**do**
  arraryBlock=ABC.getBlock();
  arrayStmt=ABC.getStmt();
  ABCSet=handleBlock(arraryBlock,ABCSet, arraryStmt);
  InState[arraryBlock]= ABCSet;
  for each pred of arraryBlock **do**
    blockSet.push(pred);
  end for
end for
while !blockSet.empty() **do**
  block= blockSet.top();
  blockSet.pop();
  for each succ of block do
    OutState[block]=Union(InState[succ]);
  end for
  ABCSet=handleBlock(arraryBlock,ABCSet, NULL);
  if ABCSet!= InState[block] **then**
    for each pred of block **do**
      blockSet.push(pred);
    end for
    InState[block]= ABCSet;
  end if
end while
return ABCSet;

算法 2： 基本块内数据流分析算法

函数：HandleBlock(block, ABCSet, arrayStmt)
输入：block，ABCSet，arrayStmt
输出：ABCSet
Stack s = ∅;
r = UNFOUND;
for each stmt in block **do**
  if stmt==arrayStmt **then**
    break;
  else
    s.push(stmt);
  end if
end for
while !s.empty() **do**
  stmt=s.top();
  s.pop();
  for each condition in ABCSet**do**
    r=HandleStmt(stmt,condition);
    if r==FOUND **then**
      ABCSet.remove(condition);
      if ABCSet.empty() **then**
        break;
      end if
    end if
  end for
end while
return ABCSet;

算法 3： 过程间后向数据流分析算法

函数：InterBDFA(CallGraph，CFG, ABCSet，f, Depth)
输入：CallGraph，CFG, ABCSet，f, Depth
输出：ABCSet
if Depth<=0 **then**
  return ABCSet;
end if
parents = CallGraph->getParent(f);
if parents.size()==0 **then**
  return ABCSet;

else
  for each parent in parents **do**
    pCFG=CFG(parent);
    set1=HandleParameters(ABCSet,f,parent);
    set2=IntraBDFA(pCFG, set1);
    ABCSet= InterBDFA(CallGraph，CFG, set2，parent, Depth-1);
  end for
end if
return ABCSet;

Table 2 Back-propagation data flow analysis rules

表 2 后向数据流分析处理规则

Statement Type	Statement Pattern	Handling Rules
Declare statement	int A = expr	replace A with expr, check and update array boundary information
Assignment	A=expr	replace A with expr, check and update array boundary information
Compound Assignment	A%=expr	check and update array boundary information
if statement	if(expr)	check whether expr implies idx<boundary or idx>=0
for statement	for(…;A<B;…)	check whether expr implies idx<boundary, if not, update loop boundary for arrays
while statement	while(expr)	check whether expr implies idx<boundary, if not, update loop boundary for arrays

法。在简单匹配处理方法中，主要处理一些表达式匹配中为常量到情形，如果常量值小于或等于相应的数组大小，我们则认为该语句完成了相应的数组边界检查。在约束求解处理方法中，我们将 clang 语句转换为约束求解器可以识别的一组约束$cond$。并将当前需要满足的数组边界条件同样转换为约束求解器可以识别的一组约束。为了将语句是否隐含数组边界条件$(idx < size)$转换为约束的可满足问题，我们将约束取反（即$!(cond \rightarrow idx < size)$交给约束求解器，若约束求解器的结果为 UNSAT（不可满足/无解），则表明原来的约束$(cond \rightarrow idx < size)$恒为真，也就是当前的条件语句将会完成对对应数组边界条件的检查。

赋值语句：如果语句是赋值语句，并且数组边界条件的下标索引正好是赋值运算符左侧的表达式，则数组边界条件的下标索引将被替换为赋值运算符右侧的表达式（我们在下面的文本中称为新索引）。并且，我们将检查该新索引是否满足相应的数组边界。*赋值语句简单匹配处理方法：*如果新索引是一个常量，则需判断该常量是否小于相应的数组大小，如果是，则将从集合中删除相应的数组边界条件；否则，我们将直接报告该数组下标需要添加数组边界检查。如果新的索引是一个包含带常数除数的模运算符的表达式，我们将检查该常数是否小于或等于相应的数组大小。如果常量小于或等于相应的数组大小，模运算将确保新索引不大于相应的数组边界，因此，我们将从待检查条件中删除相应的数组边界条件。如果新的索引是其他表达式，则会更新相应的数组边界条件。在之后的数据流分析中，我们将关注新的数组边界条件是否被满足。赋值语句约束求解处理方法：我们将赋值语句 idx=expr 和待满足的数组边界检查条件 idx<Size，构成约束$!(idx == expr \rightarrow idx < size)$交给约束求解器进行求解，如果结果为 UNSAT（不可满足/无解），则当前的赋值语句将会完成对对应数组边界条件的检查，我们将从集合中删除相应的数组边界条件。如果判断数组边界检查未被满足，数组边界条件的下标索引将被替换为赋值运算符右侧的表达式 expr。

复合赋值语句：*复合赋值语句简单匹配处理方法：*如果该语句是由带常数除数的模运算符的复合赋值操作，并且被除数正好是数组边界条件的索引，那么我们将检查该常量是否小于或等于相应的数组大小。如果该常量小于或等于相应的数组大小，则将从数组边界条件集中删除相应的数组边界条件。复合赋值语句约束求解处理方法：我们将复合赋值语句 idx op= expr 和待满足的数组边界检查条件 idx<Size，构成约束$!(idx == idx\ op\ expr \rightarrow idx < size)$交给约束求解器进行求解，如果结果为 UNSAT（不可满足/无解），则当前的赋值语句将会完成对对应数组边界条件的检查，我们将从集合中删除相应的数组边界条件。如果判断数组边界检查未被满足，数组边界条件的下标索引将被替换为表达式 idx op expr。

条件语句：如果该语句是“if”语句、“for”语句或“while”语句，我们提供了两种处理方法：

*条件语句简单匹配处理方法：*如果数组边界条件的索引正好是条件的操作数，另一个操作数是常量，并且该运算符与数组边界条件的运算符一致，则我们将检查该常量是否小于或等于相应的数组大小。如果常量小于或等于相应的数组大小，则将从数组边界条件集中删除相应的数组边界条件。

*条件语句约束求解处理方法：*当遇到“if”语句、

“for”语句或“while”语句时，我们将抽取出语句中的条件 cond 和待满足的数组边界检查条件 idx<Size，构成约束$!(cond \rightarrow idx < size)$交给约束求解器进行求解，如果结果为 UNSAT（不可满足/无解），则当前的赋值语句将会完成对对应数组边界条件的检查，我们将从集合中删除相应的数组边界条件。否则，若约束求解器的结果为 SAT，则表明当前的条件语句未完成对对应数组边界条件的检查，我们将保留这个数组边界条件，并在后续的后向数据流分析中继续对该数组边界条件进行检查。

如果在“if”或“while”条件中找不到检查，那么我们将检查数组下标是否是循环变量。如果“for”或“while”条件与模式 idx<var 匹配，则数组边界条件将更新为用 var 替换 idx。即，我们将在“for”或“while”语句之前检查 var<array size 是否存在。也就是我们将数组越界问题转换为循环上界问题继续检查。

如果在到达函数入口时未找到数组边界条件，我们将根据算法 3 继续进行过程间的后向数据流分析。首先，我们需要根据程序的函数调用图获取所有调用数组所在函数 f 的函数 parents。对于每个父函数 parent，我们遍历其 CFG 并找到调用数组函数 f 的调用语句。对于每个数组边界条件，如果其索引与函数 f 的形参之一相同，则将获取相应的实参来构造新的数组边界条件。然后在父函数 parent 中继续依照算法 1 和算法 2 进行过程内的后向数据流分析。过程间后向数据流分析过程将一只执行，直到找到所有数组的边界检查或达到配置的检查深度。

2.5 数组越界检查报告

通过后向数据流分析，在程序中存在一些赋值语句和条件语句已经保证了数组边界条件的数组将会被移除。我们需要报告程序漏掉的数组边界检查。为了报告程序中未检查的数组边界条件，我们首先需要用初始索引恢复目标集中的数组边界条件。这是因为一些数组边界条件的索引可能在数据流分析过程中，根据表 2 中列出的规则，通过赋值操作或者循环上界的替换进行了更改。数组越界检查报告主要包括关于每个数组索引的以下信息：文件、行号、函数和它所在的数组表达式，以及待检查的边界条件。这些信息比较详细，可以帮助程序员更加方便快速的定位和确认我们报告的数组越界警报，也可以作为修复推荐建议提供给程序员作为参考。

3 实现和评估

在这篇论文中，我们在原来的工具[40]的基础上，扩展实现了一个名为 ArrayBoundChecker 的工具，它的架构如图 3 所示。ArrayBoundChecker 能够在 Windows 系统和 Linux 操作系统上运行。并且，ArrayBoundChecker 基于 Clang 3.6 生成程序的抽象语法树（AST）文件。首先，ArrayBoundChecker 将程序的 AST 文件作为输入。然后，ArrayBoundChecker 基于抽象语法树构建控制流图和调用图。接下来，ArrayBoundChecker 基于控制流图和调用图执行污点分析和数据流分析。我们在原来的工具中增加了约束求解过程，使用到的约束求解器是 Z3 定理证明器[1]。最后，ArrayBoundChecker 输出数组越界检查报告。ArrayBoundChecker 是一个全自动工具，可以通过用户配置来设置执行过程间数据流分析的深度。

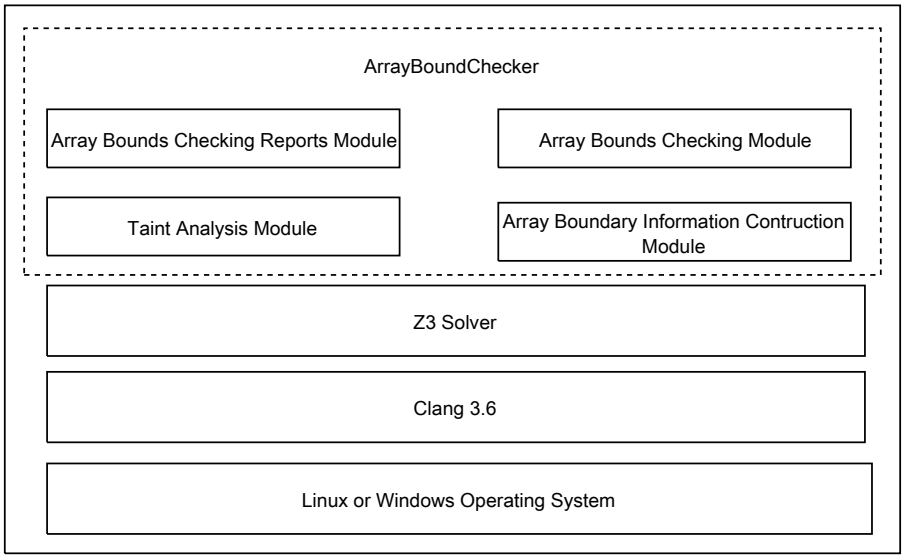

Fig. 3 Architecture of ArrayBoundChecker

图 3 ArrayBoundChecker 工具架构

Table 3 Warnings of ArrayboundChecker

表 3 数组越界检查警报统计

Program	Scale （LOC）	True Warnings	Warnings r Simplifier	FP Simplifier	Warnings Z3Solver	FP Z3Solver
libco-v1.0	5976	2	3	33.3%	3	33.3%
libfreenect-0.5.7	34664	10	10	0%	10	0%
vips-8.7.4	167730	42	49	14.3%	47	10.6%
coreutils-8.30	206751	5	12	58.3%	10	50%
curl-7.63.0	233722	16	30	46.7%	15	13.3%
libxml2-2.9.9	2302351	5	9	44.4%	6	28.6%
Total	4772197	80	113	29.2%	92	16.3%

Table 4 Time and Memory Consumption of ArrayboundChecker

表 4 数组越界检查时间和内存开销

Program	Scale （LOC）	AST Number	Normal Time (s)	Simplifier Memory(MB)	Z3 Time (s)	Z3 Memory (MB)
libco-v1.0	5976	6	0.18	32.9	0.32	44.3
libfreenect-0.5.7	34664	17	0.64	55.3	0.83	66.0
vips-8.7.4	167730	411	109.8	3866.2	116.6	3874
coreutils-8.30	206751	393	38.5	1225.4	36.1	1234.3
curl-7.63.0	233722	179	11.4	555. 6	12.3	565.0
vim-8.1.0818	838569	81	142.3	1785.1	140	1796.9
espruino-2.01	1141645	97	4.9	335.5	9.1	348.6
libxml2-2.9.9	2302351	50	171.0	2092.8	170.8	2105.1

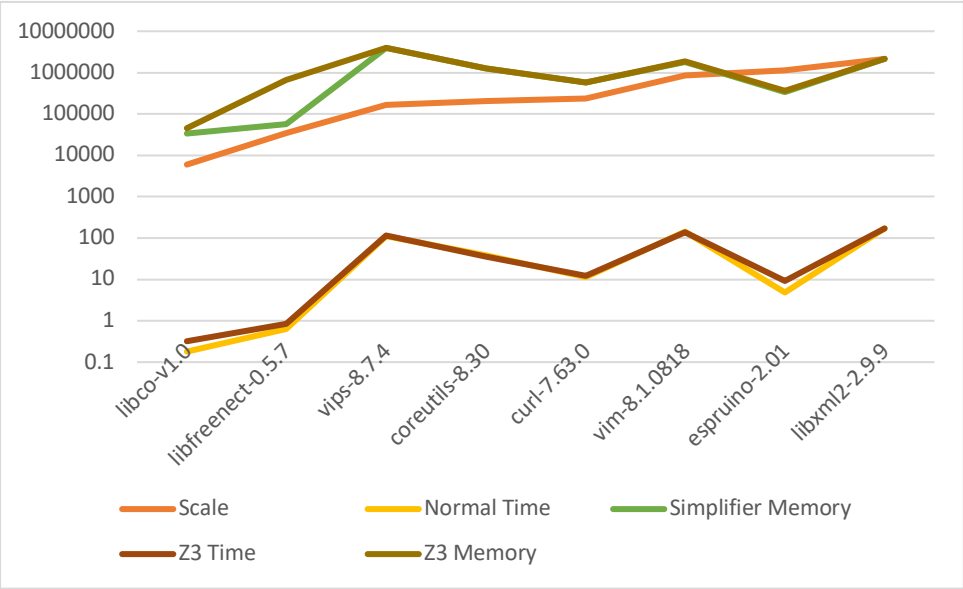

Fig. 4 Architecture of ArrayBoundChecker

图 4 数组越界检查时间和内存开销随代码规模趋势图

3.1 内存优化

大规模程序总是包含大量 AST 文件。如果一次性读入所有 AST 文件，包含 AST 文件的所有内容，将会消耗大量内存。这将会严重制约 ArrayBoundChecker 对大规模程序的支持。比如，PHP-5.6.16 包含 25 万行源代码和 211 个 AST 文件，当我们尝试一次性读入所有 AST 文件时，在 2 GB 内存的机器上将无法运行。为了支持在有限的内存资源下扫描十万行甚至一百万行代码的程序，我们在 ArrayBoundChecker 中实现了内存优化策略。内存优化策略的关键思想是使用一个 AST 队列来仅在内存保留最新使用的 AST 文件，例如只保留 200 AST 文件。AST 文件越少，内存消耗就越少。并且 AST 队列的最大容量可以由用户配置。用户可以根据需求和计算机容量配置 AST 队列的最大容量。在分析 AST 的内容时，ArrayBoundChecker 将首先检查相应的 AST 是否在内存中。如果 AST 在内存中，ArrayBoundChecker 会将 AST 移动到队列的末尾。如果 AST 不在内存中，ArrayBoundChecker 将从 AST 文件读入 AST 的内容。当 AST 队列达到其最大容量时，ArrayBoundChecker 将删除最先读入的 AST。值得注意的是，如果用户设置了一个较小的最大 AST 数，ArrayBoundChecker 将会更加频繁地读取 AST 文件。因此，如果有足够的内存，用户应选择更大的最大 AST 数，以减少频繁的读操作并提高 ArrayBoundChecker 的效率。

3.2 求解优化

约束求解是非常耗时的，尤其是频繁的调用约束求解器将严重增加分析时间，制约工具的可扩展性。而在我们的方法中，由于需要计算不动点，会增加对相同的约束求解的需求。为了减少对约束求解器的调用，我们保存了函数内语句是否隐含数组边界检查的结果列表，先查询这个列表，若未进行过相应的约束求解，再调用约束求解器，可以大大减少对约束求解器的调用。程序中可能存在一些按位操作及一些其他操作语句，对应到 Z3 约束求解时可能比较耗时。因此，我们为 Z3 提供了 timeout 的配置项。

3.3 实验评估

为了评估 ArrayBoundChecker 的有效性和效率，我们使用 ArrayBoundChecker 对几个开源程序的源码进行了扫描分析，结果显示在表 3 和表 4 中。程序的规模最大可以达到 200+万行。

我们通过人工审查程序源码，对表 3 中 ArrayBoundChecker 报告的数组越界检查警报进行了人工确认。我们分别统计了*赋值语句简单匹配处理方法*和*约束求解处理方法*这两种方法的误报情况，可以发现前者的平均误报率为 29.2%，后者的平均误报率为 16.3%。导致*约束求解匹配处理方法*误报的主要原因是我们无法处理一些库函数调用，如果在条件语句或者赋值语句中存在库函数调用并保证了数组边界，但是我们无法判断导致误报。而*赋值语句简单匹配处理方法*相比*约束求解匹配处理方法*误报更多的原因是前者是简单匹配一些固定格式的语句并要求语句中特定位置为常量，很多复杂情形无法处理导致误报；而通过约束求解我们可以增强条件判断的能力，除了能处理更多的线性约束，甚至可以处理非线性约束。

为了评估 ArrayBoundChecker 分析效率，我们统计了如表 4 所示的程序的分析时间和内存消耗。*约束求解匹配处理方法*由于调用约束求解器，总体会*赋值语句简单匹配处理方法*比消耗更多的时间和内存。但是时间平均增加了 1.53%，内存平均增加了 0.86%。使用约束求解方法并未造成明显的时间和内存消耗增加，这是因为，一方面，我们存储和复用了约束求解的结果，避免了冗余的约束求解；一方面，我们针对求解 expr implies expr 约束进行了专门的优化，减少了对约束求解器的调用；另一方面，由于约束求解可以精确的判断程序语句是否对数组边界进行了检查，可以尽早的移除已经被满足的数组边界检查，从而从总体上能过节约开销。如图 4 所示，我们可以发现*赋值语句简单匹配处理方法*和*约束求解匹配处理方法*在时间和内存消耗上都随着程序规模的增加，呈现接近线性趋势的增长，因此我们的方法具有很好的可扩展性。

3.4 实验讨论

从上述实验评估中可以看到我们的方法可以快速有效地报告出程序中遗漏的数组边界检查，但是我们的方法还存在一些不足。

库函数：目前的方法由于采用静态分析方法，在不能获得库函数的源码实现的情况下，我们将无法判断这些库函数的功能和作用，但是可能这些库函数实现了对数组边界的检查和保证，因此会导致我们的工具产生误报。

复杂数组下标：程序中存在一些使用复杂表达式作为数组下标的情形，目前的工具实现还未处理，因此会导致我们的工具存在漏报。

准确性和可扩展性：由于约束求解比较耗时，尤其是求解一些复杂约束，比如一些按位运算时，约束求解器将无法在较短的时间内给出求解结果。这将会限制我们工具的可扩展性。我们在实验只能够通过设置 timeout 时间来跳过了一些复杂的约束求解，但是这又可能会导致我们的工具的误报和漏报。

4 相关工作

4.1 污点分析

动态污点分析是一种当前流行的分析软件的方法。多种不同的技术被提出来通过进行动态污点分析用于追踪侦查软件中的隐藏漏洞[8,10,24,27]。污点分析会将潜在的恶性数据源视为污染源，如网络数据包；随后，监控这些受污染数据如何在整个程序执行过程中传播；当敏感数据（如堆栈中的返回地址或用户特权设置）被污染的数据污染时执行对应的处理操作。与动态污点分析相比，静态污点分析以静态方式跟踪污染源或者二进制文件中的信息。STILL [31] 是一个基于静态污染和初始化分析的防御机制，可以在各种互联网服务中（例如 Web 服）检测嵌入在数据流中的恶意代码。

为了减少污点分析的开销，TaintPipe[23]借助轻量级的运行时日志记录来生成紧凑的控制流信息，并产生多个线程，以流水线的方式并行地执行符号化污点分析。静态污点分析面对另一个问题是耗费人力。大多数现有的静态污点分析工具会在潜在的易受攻击的程序位置报错。这会导致需要开发者人工确认报告，为此需要耗费时间。Ceara 等人[35]呈现一种基于细粒度数据的控制污点分析的污点依赖序列算子，通过提供一些关于需要被分析的路径集合的信息帮助开发者们。

4.2 数组越界检查

Xu 等人[33]提出了一种直接在不受信任的机器上运行的方法。但是需要对这些不受信任的程序的初始输入要通过注释提供使用类型状态信息和线性约束。Detlefs 等人 [11] 提出了一种针对常见程序错误的静态检查器，这些错误有数组下标越界，空指针解引用和并发类错误（多线程程序中）。他们的分析利用线性约束，自动合成循环不变量用于边界检查，并且根据策略规范参数化。他们的安全检查分析适用于有源码的程序并可以利用既不健全又不完备的分析方法。

Leroy 和 Rouaix[20] 提出了一个理论模型，来系统的把基于类型的运行时检查放入主机代码的接口例程中。他们的技术与我们的在下面几个方面不同：它是动态的，它检查主机而非代码，并且它要求主机 API 的源码是可用的。此外，它的安全归约是在特定位置通过不变式定义，然而，我们的安全策略模型更为通用。

ABCD[5]是一种轻量级算法，用于按需消除无用的数组检查。它的设计强调简单和高效。实质上。ABCD 工作原理是通过对图执行一次简单的遍历来向 SSA 值的图增加一些边。 ABCD 平均能够删除 45%的动态边界检查指令，有时可以实现接近理想化的优化。

[37]提出一个在编译时检测数组越界的轻量级验证方法，但是为了达到线性验证时间，该方法需要开发者预先注释相关信息，例如程序边界信息。相比之下，我们的方法能够分析出程序边界。

当前也存在许多基于静态分析技术的数组越界检测工具。Chimdyalwar[7]对五种用于侦测这一漏洞的静态分析工具进行了评估。在它们之中，Polyspace 和 Coverity 是商业工具；一个是学术工具 ARCHER，其他两个是开源工具 UNO 和 CBMC。Polyspace 是唯一无漏报的工具，但由于它是内存密集型分析，不能以同样的高精度扩展大型软件上。相比之下，Coverity 应用于百万行级别的代码上，但它的分析并不可靠。我们发现，UNO 同时有误报和漏报，并且不能应用在大型软件上。ARCHER 宣称可以运行在百万行级别的代码上，但是分析并不完善。CBMC 模型检查器进行了精确的分析，无法在大型代码上达到相同精度。

4.3 缓冲区溢出检测

Tance[30] 为了检测缓存溢出漏洞提出了一个黑箱组合测试方法。Dinakar et al.[12] 提出了一种具有吸引力的、全自动的低开销方法，这种方法旨在对 C 和 C++程序中的数组和字符串进行运行时的越界检查。这些技术本质上是基于一种对内存的细粒度分区，在一组 benchmark，运行时额外开销可至 12%。

有一组用于调试但需要源码的方法，例如 Loginov[21]和 rtcc[26]。由于它们全都专注于调试，因此，它们运行时的额外开销会不会被重视。例如，Loginov 的工作被报道出的开销高达 900%。一些工具，包括 SafeC[4]， Cyclone[17] 和 DangDone[36]使用扩展的指针表示 ，包含每个指针值的合法目标对象的对象基础信息和大小。使用这些指针要对程序进行大量修改以使用外部库，这些外部库函数通常是被包装好的用于转换指针的方法。此外，编写这样的封装对于间接函数调用以及访问全局变量或存储器中的其他指针的函数来说可能是难以实现的。

预防技术是一种可以防止数组下标越界的影响的方法。例如，StackGuard[9]可能在检测到堆栈上的返回地址被覆盖后终止进程。运行时预防的现有方法具有显著的运行时开销。除此之外，这些方法在可能有漏洞的程序部署完成后生效。CFI[2]检查程序的控制流程是否在执行期间被劫持。这与我们的工作形成对比，我们的工作目的是在部署之前开发和释放没有数组下标越界错误的程序。

4.4 模糊测试与符号执行

Fuzzing[28]是安全测试中使用最广泛的黑盒测

试方法之一。模糊测试通常从一个或多个合法输入开始，然后随机改变这些输入以获得新的测试输入。高级模糊测试技术[13]是基于生成的模糊测试技术，它为了解决拥有复杂输入结构的程序的输入生成问题，通过基于语法的输入归约定义有效输入。[15]提出了一种替代的白盒模糊测试方法，结合了符号执行和动态测试生成。虽然模糊测试可以检测到数组越界错误，但一个主要限制是代码覆盖率低。此外，一些数组越界的错误可能只读取越界的区域，因此不会导致崩溃，这样模糊测试中的监视器可能无法检测到这种情况[22]。我们的方法基于静态方法，可以实现高代码覆盖，并且可以检测不同类型的数组下标越界。

模糊测试方法也会存在一些无法完成代码的高覆盖的问题。比如一些窄约束通过变异输入很难覆盖到。而常用的软件分析技术符号执行也面临着路径爆炸和约束求解等问题。尤其是在复杂的大规模程序可能导致路径爆炸，因此无法求解这些约束。最近，人们越来越关注将模糊测试与符号执行等技术相结合的方法[6,14,32,38,39]。这些技术中，符号执行主要用于收集路径条件，并通过约束求解生成测试用例。然后对分支条件取反以得到新的测试输入，这些测试输入将帮助模糊测试在执行时探索新的路径。

5 结论和未来工作

在这篇论文中，我们提出了一种基于污点分析和后向数据流分析的数组越界检查的静态分析方法，并实现了一个可以在 Windows 和 Linux 操作系统上运行的自动化静态分析工具——ArrayBoundChecker，来自动扫描 C 程序的源代码，检查程序中是否包含正确的语句，如赋值语句和条件语句，以确保数组的下标索引在数组边界内。如果程序中不存在正确的数组边界检查，我们将报告相应的数组位置和待添加的数组边界条件。我们通过扫描一些真实程序的源代码来评估 ArrayBoundChecker 工具。实验数据表明，ArrayBoundChecker 可以快速报告在程序中没有进行数组边界检查的数组下标，使用约束求解方法时，误报率大约为 16.3%。尽管 ArrayBoundChecker 有一些误报和漏报，但 ArrayBoundChecker 可以有效减少的程序员人工审查工作。我们的方法可以提供待增加的数组边界检查条件和位置，可以帮助程序员更加方便快速的定位和确认我们报告的数组越界警报，也可以作为修复推荐建议提供给程序员作为参考。

目前，ArrayBoundChecker 无法处理库函数调用、复杂数组下标等情形，在约束求解方法也存在复杂约束求解较慢，从而限制了工具的可扩展性，因此我们需要在未来工作中进一步完善和优化我们的方法和工具，比如通过优化约束求解器，以提高我们方法的可扩展性，通过函数摘要提前计算和存储库函数的功能信息，完善工具的实现等，进一步减少我们方法等误报和漏报。

参 考 文 献

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

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

[3] Allen F E. Control flow analysis[C]//ACM Sigplan Notices. ACM, 1970, 5(7): 1-19.

[4] Breach T M A S E, Sohi G S. Efficient Detection of All Pointer and Array Access Errors[J]. 1993.

[5] Bodík R, Gupta R, Sarkar V. ABCD: eliminating array bounds checks on demand[C]//ACM SIGPLAN Notices. ACM, 2000, 35(5): 321-333.

[6] Cadar C, Ganesh V, Pawlowski P M, et al. EXE: automatically generating inputs of death[J]. ACM Transactions on Information and System Security (TISSEC), 2008, 12(2): 10.

[7] Chimdyalwar B. Survey of array out of bound access checkers for C code[C]//Proceedings of the 5th India Software Engineering Conference. ACM, 2012: 45-48.

[8] Costa M, Crowcroft J, Castro M, et al. Vigilante: End-to-end containment of internet worms[C]//ACM SIGOPS Operating Systems Review. ACM, 2005, 39(5): 133-147.

[9] Cowan C, Pu C, Maier D, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks[C]//USENIX Security Symposium. 1998, 98: 63-78.

[10] Crandall J R, Su Z, Wu S F, et al. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits[C]//Proceedings of the 12th ACM conference on Computer and communications security. ACM, 2005: 235-248.

[11] Detlefs D L, Leino K R M, Nelson G, et al. Extended static checking [J]. 1998.

[12] Dhurjati D, Adve V. Backwards-compatible array bounds checking for C with very low overhead[C]//Proceedings of the 28th international conference on Software engineering. ACM, 2006: 162-171.

[13] Godefroid P, Kiezun A, Levin M Y. Grammar-based whitebox fuzzing [C]//ACM Sigplan Notices. ACM, 2008, 43(6): 206-215.

[14] Godefroid P, Klarlund N, Sen K. DART: directed automated random testing[C]//ACM Sigplan Notices. ACM, 2005, 40(6): 213-223.

[15] Godefroid P, Levin M Y, Molnar D A. Automated Whitebox Fuzz Testing[C]//NDSS. 2008, 8: 151-166.

[16] Handbook of graph theory[M]. CRC press, 2004.

[17] Hicks M, Morrisett G, Grossman D, et al. Experience with safe manual memory-management in cyclone[C]//Proceedings of the 4th internation al symposium on Memory management. ACM, 2004: 73-84.

[18] Khedker U, Sanyal A, Sathe B. Data flow analysis: theory and practice [M]. CRC Press, 2009.

[19] Kildall G A. A unified approach to global program optimization[C]//Pro ceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Pri nciples of programming languages. ACM, 1973: 194-206.

[20] Leroy X, Rouaix F. Security properties of typed applets[M]//Secure Inte rnet Programming. Springer, Berlin, Heidelberg, 1999: 147-182.

[21] Loginov A, Yong S H, Horwitz S, et al. Debugging via run-time type c hecking[C]//International Conference on Fundamental Approaches to So ftware Engineering. Springer, Berlin, Heidelberg, 2001: 217-232.

[22] McNally R, Yiu K, Grove D, et al. Fuzzing: the state of the art[R]. DE FENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBUR GH (AUSTRALIA), 2012.

[23] Ming J, Wu D, Xiao G, et al. TaintPipe: pipelined symbolic taint analy sis[C]//24th {USENIX} Security Symposium ({USENIX} Security 15). 2015: 65-80.

[24] Newsome J, Song D X. Dynamic Taint Analysis for Automatic Detecti on, Analysis, and SignatureGeneration of Exploits on Commodity Softw are[C]//NDSS. 2005, 5: 3-4.

[25] Ryder B G. Constructing the call graph of a program[J]. IEEE Transact ions on Software Engineering, 1979 (3): 216-226.

[26] Steffen J L. Adding run-time checking to the portable C compiler[J]. S oftware: Practice and Experience, 1992, 22(4): 305-316.

[27] Suh G E, Lee J W, Zhang D, et al. Secure program execution via dyna mic information flow tracking[C]//ACM Sigplan Notices. ACM, 2004, 39(11): 85-96.

[28] Sutton M, Greene A, Amini P. Fuzzing: brute force vulnerability discov ery[M]. Pearson Education, 2007.

[29] Thulasiraman K, Swamy M N S. Graphs: theory and algorithms[M]. Ne w York: Wiley, 1992.

[30] Wang W, Lei Y, Liu D, et al. A combinatorial approach to detecting bu ffer overflow vulnerabilities[C]//2011 IEEE/IFIP 41st International Conf erence on Dependable Systems & Networks (DSN). IEEE, 2011: 269-2 78.

[31] Wang X, Jhi Y C, Zhu S, et al. Still: Exploit code detection via static t aint and initialization analyses[C]//2008 Annual Computer Security App lications Conference (ACSAC). IEEE, 2008: 289-298.

[32] Xu R G, Godefroid P, Majumdar R. Testing for buffer overflows with l ength abstraction[C]//Proceedings of the 2008 international symposium on Software testing and analysis. ACM, 2008: 27-38.

[33] Xu Z, Miller B P, Reps T. Safety checking of machine code[C]//ACM SIGPLAN Notices. ACM, 2000, 35(5): 70-82.

[34] Ye T, Zhang L, Wang L, et al. An empirical study on detecting and fix ing buffer overflow bugs[C]//2016 IEEE International Conference on So ftware Testing, Verification and Validation (ICST). IEEE, 2016: 91-101.

[35] Ceara D, Mounier L, Potet M L. Taint dependency sequences: A charac terization of insecure execution paths based on input-sensitive cause seq uences[C]//2010 Third International Conference on Software Testing, Ve rification, and Validation Workshops. IEEE, 2010: 371-380.

[36] Wang Y, Gao F, Situ L, et al. DangDone: Eliminating Dangling Pointer s via Intermediate Pointers[C]//Proceedings of the Tenth Asia-Pacific Sy mposium on Internetware. ACM, 2018: 6.

[37] Kellogg M, Dort V, Millstein S, et al. Lightweight verification of array indexing[C]//Proceedings of the 27th ACM SIGSOFT International Sym posium on Software Testing and Analysis. ACM, 2018: 3-14.

[38] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing Thro ugh Selective Symbolic Execution[C]//NDSS. 2016, 16(2016): 1-16.

[39] Pak B S. Hybrid fuzz testing: Discovering software bugs via fuzzing an d symbolic execution[J]. School of Computer Science Carnegie Mellon University, 2012.

[40] Gao F, Chen T, Wang Y, et al. Carraybound: static array bounds checki ng in C programs based on taint analysis[C]//Proceedings of the 8th As ia-Pacific Symposium on Internetware. ACM, 2016: 81-90.

Fengjuan Gao, born in 1991. PhD student. Her main research interests include software analysis, symbolic execution.

Yu Wang, born in 1991. PhD student. His main research interests include software analysis, concurrent programs and machine learning.

Linzhang Wang is a Professor with the State Key Laboratory of Novel Software Technology at Nanjing University, China. His research interests include software engineering and software security.

Xuandong Li received the BS, MS, and PhD degrees in computer science from Nanjing University in 1985, 1991 and 1994, respectively. Since 1994 he has been in the Department of Computer Science and Technology, Nanjing University where he is currently a professor. His research interests include formal support for design and analysis of reactive, disturbed, real-time, and hybrid systems, software testing and verification, and model driven software development.