CBOC：一个 C 语言缓冲区溢出漏洞有效检测工具[1]

陈石坤[1]，李舟军[2]

[1]国防科技大学计算机学院，长沙 (410073)

[2]北京航空航天大学计算机学院，北京 (100191)

E-mail: nudtmao@126.com

摘 要：缓冲区溢出是 C 程序中很多安全问题的根源。本文给出一个 C 语言缓冲区溢出漏洞的有效检测工具 CBOC（C Buffer Overflow Checker）。该工具基于符号执行，并引入基址安全表达式的概念，优化了符号执行过程。CBOC 以 C 程序源代码作为输入，能够自动检测程序中包含的缓冲区溢出漏洞。使用 CBOC 对公开的基准程序测试包进行检测，并在相同条件下和主流的 ANSI-C 限界模型检验工具 CBMC 进行了比较。实验表明：对于大多数测试用例，CBOC 的性能优于 CBMC，且 CBOC 使用内存较少。在检测精度的测试实验中，CBOC 误报率约为 0.34%，漏报率约为 1.72%。

关键词：缓冲区溢出; 符号执行; 基址安全表达式

中图分类号:

1. 引言

缓冲区溢出发生在将数据D写入到缓冲区B，而D的长度比分配给缓冲区B的长度大的时候。在没有显式边界检查的编程语言中，缓冲区可能被不受限制地越界读写。C语言允许直接操作指针而不作边界检查，标准C库中包含很多函数，如果使用不当，它们就可能导致缓冲区溢出问题。这使得缓冲区溢出成为C语言程序中很多安全问题的根源。

缓冲区溢出漏洞主要被攻击者利用来执行任意的代码（比如，一个具有管理员权限的shell）。在C语言中，函数调用时，被调函数的参数、返回地址以及函数内部定义的局部变量将先后被压入栈中，栈由高地址向低地址增长。当函数中的一个局部缓冲区溢出时，它将会由低地址向高地址覆盖相邻的数据，这将有可能覆盖函数的返回地址。攻击者可以精心设计一段数据来覆盖栈空间，改写函数的返回地址，让返回地址指向一段打开shell的代码，从而获取管理员的权限。

缓冲区溢出漏洞检测方法一般分为动态[1,2]和静态[3-8]两类。动态检测方法，需要在程序中插入动态检测代码或断言，在程序的执行过程中及时发现缓冲区溢出漏洞。这类方法增加了程序的执行开销，也不能证明系统中不存在缓冲区溢出漏洞。而静态方法从分析程序的源代码入手，在软件发布和运行之前就能发现和确定软件的安全漏洞，可直接应用到软件开发阶段，以提高软件质量。

目前，主要的静态分析方法通常将缓冲区溢出问题转换为约束求解问题：例如，将缓冲区溢出问题转换为整数范围分析问题[3]。受限于整数范围分析的精度，这种方法可能产生大量的误报，同时，这种方法几乎完全没有考虑指针别名，这也会导致大量的误报或漏报；将缓冲区溢出问题转换为整数线性规划问题[4]。由于流敏感的指针分析计算复杂度较高，该方法采用了流不敏感的指针分析算法，并最终导致系统产生大量误报。在上述两种分析方法中，所发现的缓冲区溢出问题必须通过进一步的手工检测来验证其是否确实是一个真实的漏洞。

CSSV [5]是一个缓冲区溢出检测工具，利用它可以发现程序中包含的所有由字符串操作

[1]本课题得到高等学校博士学科点专项科研基金（项目编号：20070006055）的资助。

导致的缓冲区溢出漏洞。该工具对软件中的每个函数单独进行分析，需要程序员为每个函数提供前置条件、后置条件以及可能的副作用。该方法仅产生少量的误报。然而，它的分析过程需要手工为函数提供合同（前置、后置以及副作用申明），难以保证函数与它的合同的一致性，并且不能全自动地进行检测。

模型检验方法具有可靠性和完备性，理论上能保证没有误报和漏报，并且能全自动地实行。文献[6,7] 尝试使用模型检测工具ComFoRT来证明软件系统中不包含缓冲区溢出漏洞。然而，ComFoRT本身对指针提供的支持也很有限 [6]。

CBMC [8]是目前使用较为广泛的C语言限界模型检验工具。它支持ANSI-C的所有特性，包括指针操作，复杂的数据结构，存储空间动态分配，函数递归调用等。它能够对程序中的一些安全属性进行检测，包括数组越界访问，除0错误，空指针，算术溢出以及用户自定义的断言等。然而，对一些常用的C语言指针表达式（可能不符合ANSI-C标准，但对缓冲区溢出检测至关重要），CBMC没有提供充分的支持。例如：设x是一个类型为struct {int a; int b;}的结构体变量，CBMC最近版本（2.8版）将认为表达式*((int*)((char*)&x+sizeof(int)))是非法的。实际上该表达式常被用于访问x.b。

从以上分析可以看出，C程序中的指针和指针表达式给C语言程序分析工具带来很大挑战。实际上，即使对最常用的数组，常用的C语言模型检验工具SLAM[9]和BLAST[10]都不能提供很好的支持。

本文给出一个C语言缓冲区溢出检测工具CBOC（C Buffer Overflow Checker）。该工具使用符号执行[11]的方式检测C程序中的缓冲区溢出漏洞。使用经典的符号执行算法处理C语言程序时，同样会遇到一些与指针相关的棘手问题，包括：数组元素的混淆，复杂的数据结构，指针与引用的混淆等，也难以处理程序中出现的循环。CBOC引入基址安全表达式的概念，基址安全表达式的等价性可以方便的从形式上进行判断，依据这一特性CBOC对符号执行过程进行了优化，从而缓解了指针、数组等给程序分析带来的复杂度。使用公开的缓冲区溢出漏洞测试包对CBOC进行评测，并在相同条件下和CBMC进行比较。实验结果表明：对于大多数测试用例，CBOC的性能都优于CBMC，且CBOC使用内存较少。采用一个包含1164个代码片段的缓冲区溢出漏洞测试包对CBOC的误报率和漏报率进行评测，结果表明：CBOC的误报率和漏报率都很低。

本文以下章节安排如下：第二节简单介绍缓冲区溢出的形成机理；第三节介绍CBOC的系统结构；第四节介绍CBOC中使用的关键技术；第五节给出一些实验分析结果；最后一节总结全文。

2. 缓冲区溢出漏洞

缓冲区溢出通常可以分为栈溢出、堆溢出、格式化串溢出等多种类型，而按照溢出产生的原因往往还可以进一步分为整数宽度溢出、整数算术溢出、整数符号问题以及数组越界等多种类型。本节以栈溢出为例，介绍缓冲区溢出漏洞的形成机理。

栈溢出发生在函数调用过程中，通过覆盖函数的返回地址重定向程序的控制流。程序执行过程中，通常将局部数据和函数调用现场保存在栈中。栈由高地址向低地址增长（与字符串增长方向相反）。函数调用过程大致如下：(1) 函数入口参数逆序入栈；(2) 指令指针EIP入栈，作为函数的返回地址RET；(3) 当前栈顶指针EBP入栈，以便在函数调用返回时恢复栈状态；(4) 函数局部变量入栈；(5) 执行函数体；(6) 函数返回。图 1 给出一个典型的栈溢出例子：当函数func中的strcpy执行完后，str中的内容（128个字符）将覆盖由buf_dest开

始的128个字节，函数func的返回地址将被覆盖。通过精心设计str的内容，可将函数func的返回地址覆盖为一段恶意代码的地址。函数返回时控制流就会定向到恶意代码上。

```
func(char* buf_src) {
   char buf_dest[8];
   strcpy(buf_dest, buf_src);
}
void main() {
   int i; char str[128]
   for(i=0; i<127; i++) str[i] = 'c';
   str[127] = '\0';
   func(p);
}
```

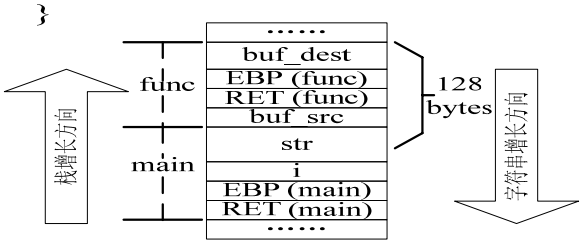

图 1　栈溢出

Fig.1　Stack overflow

C库中的一些函数，如果使用不当，它们将可能导致缓冲区溢出。这类函数包括gets，strcpy, strcat, sprintf，memcpy, strncpy等。

3. CBOC 的系统结构

CBOC使用Ocaml语言开发，运行在Window平台上（需要Cygwin支持），也可以编译为linux的可执行程序。CBOC提供命令行执行方式，同时以Eclipse插件的方式提供了一个图形界面。图形界面需要Eclipse(v3.42)和CDT(v5.02)支持。

CBOC包括如下六个主要的功能模块：

1）存储表达式简化模块Simplemem，其主要功能为：通过合理地引入临时变量对程序进行变换，在变换后的程序代码中，表达式中的每一个变量至多只会包含一重间接访问，并使用明确的常数来替换sizeof、alignof等系统相关的常量。

2）无关语句裁剪模块Astslicer和Slicer，其主要功能为：裁剪未被调用的函数及与缓冲区溢出无关的语句。

3）表达式规范模块Normalexp，用于将程序中的表达式化为规范型表达式。

4）文法转化模块Csg，它是一个辅助模块，使用一种简洁的表达式文法来存储和索引符号执行过程中的表达式。

5）顺序语句序列符号化执行模块Secsg，该模块是CBOC的重要模块之一，它根据变量及函数的定义建立缓冲区初始化信息，符号执行不包含分支的顺序路径，在符号执行过程中，判定路径是否可达，并判定路径中的断言是否成立。在它提供的主接口中，有2个接口是关键接口，分别是assign和assume。接口assign为当前正在执行的符号化路径增加一条待执行的赋值语句。接口assume为当前正在执行的符号化路径的路径条件增加一个新的断言；

6）程序执行路径动态扩展模块Symex，该模块是CBOC的另一个重要模块，它动态地进行循环展开和函数内嵌，扩展程序执行路径，并调用Secsg模块完成程序的符号化执行，根据Secsg的反馈来判断路径是否可达，当发现路径不可达时停止对当前路径进行扩展，转而执行一条新的符号化路径。当Secsg报告溢出漏洞时，记录Secsg反馈的错误路径。

图2给出CBOC进行漏洞检测的工作流程。CBOC首先读入待检测的C程序源文件列表，启动GCC对列表中的程序文件进行预处理，主要是进行宏扩展。随后CBOC读入预处理后得到的程序代码文件，进行必要的变量重命名，将这些文件合并成为一个程序文件，进行无关语句裁剪，表达式化简。最后，对程序进行符号执行，检测程序中的缓冲区溢出漏洞。

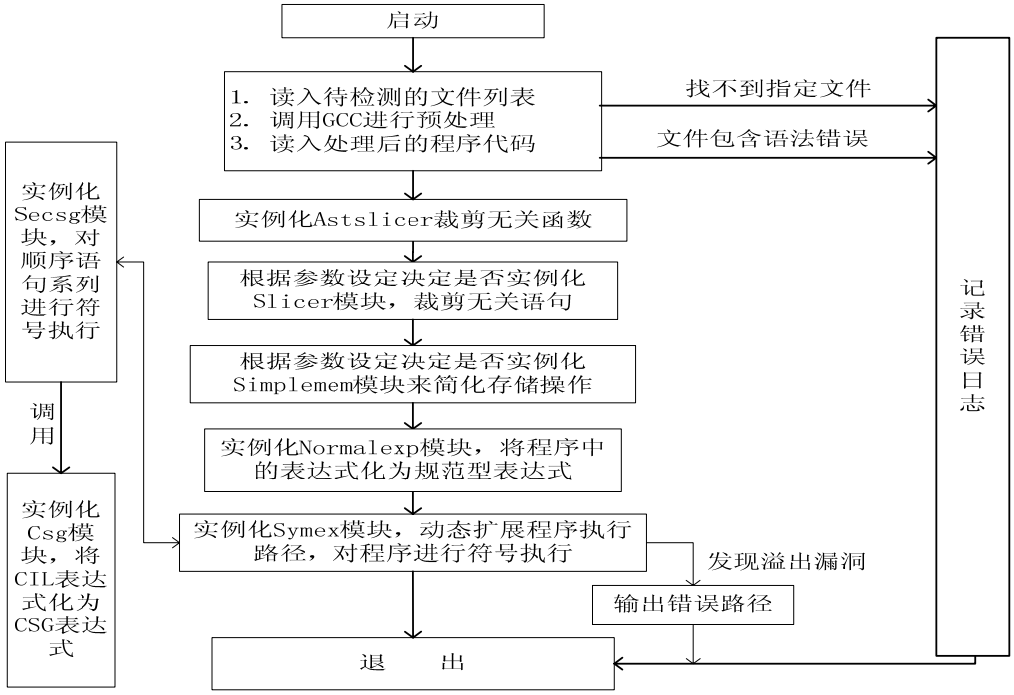

图 2 CBOC 工作流程

Fig.2 Workflow of CBOC

4. 关键技术

4.1 程序代码预处理技术

CBOC中使用的很多关键技术均假设程序代码已经做过必要的预处理。这里对CBOC中需要作的预处理加于说明。

C语言是一种非常灵活的编程语言，它可以方便地执行直接面向硬件的操作，这些特性同时也导致C语言难于理解和分析。将C语言转化为一种方便分析的中间语言，可以使程序分析工具从灵活而复杂的C语言语法中解脱出来。CIL(C Intermediate Language) [12]是berkeley大学开发的开源软件。CIL定义了C语言的一个简洁（clean）、规范的子集，支持ANSI-C的全部特性，并为GCC和MSVC提供部分扩展支持。CBOC采用CIL作为前端，对C程序进行解析，并作必要的预处理。首先CBOC调用编译器GCC或CL对程序进行宏扩展。程序中通常采用#define，#ifdef等指令定义一些宏，使用编译器提供的宏扩展功能（例如，-E参数可以让gcc提供宏扩展功能）可以将代码中的宏替换成其对应的常量或语句。随后，CBOC对程序中的控制流结构和表达式进行简化。CBOC借助CIL提供的库函数来实现这些功能。CIL提供了几个比较重要的功能，包括：消除表达式的副作用：将局部变量定义中的初始化转化为等价的赋值语句；将条件表达式 “?:”转化为等价的if…else结构；将各种循环(while，for以及do)统一转换为while循环；进行必要的变量重命名，将多个程序文件合并为一个；通过合理地引入临时变量对程序进行变换，消除程序中的多重指针解引用。消除多重解引用后，对任意的程序变量x，只允许x，&x和*x出现在程序中，而**x，***x…等都被等价的表达式

所替换。CBOC还借助CIL将Switch语句转化为if..else if…else..结构，将break转化为等价的goto语句，将continue转化为一个等价的goto语句。

4.2 循环处理技术

CBOC借鉴限界模型检验中的思想，即只检查有限长的路径上的状态来发现错误。因此，程序中的循环将被展开，并用if…else语句替换。对于while循环W，若W的循环的遍数是常量或常量表达式n，就将W的循环体复制n遍。否则，展开的遍数可以用事先设定的参数控制。例如，假设循环为while(cond) { Body; }，参数设定的展开次数为2，我们将循环体复制2遍。变换后代码为：if(cond) { Body; if(cond) { Body; assert(!cond); } } 。其中assert(!cond)称为展开断言，如果循环展开不充分（实际的循环遍数大于2），程序验证时，该展开断言将被违背。利用该断言，可以通过逐步调整参数，加大展开遍数，在更长的路径上查找错误。对于向后的goto语句（跳转目标在该goto语句之前），使用类似while循环的方式展开。向前的goto可以使用等价的if…else语句替换。CBOC在实现时并不静态地复制程序代码来展开循环，而是由symex模块动态产生符号化执行路径。

4.3 函数调用的处理技术

对于有源代码的函数调用，采用过程内嵌的方式进行处理，即，在函数调用的地方，复制函数体，经过必要的变量重命名，使用函数体本身来代替函数调用。若函数将返回值赋给一个左值表达式，则使用等价的赋值语句替换Return语句。递归函数可采用类似循环展开的方式进行扩展，但目前的CBOC版本尚不支持函数递归调用。没有源代码的函数将被分为两类：一类是strcpy，strcat，fgets等字符串操作函数，以及malloc和free等存储管理函数，为了检测缓冲区溢出漏洞，需要对此类函数提供特别的支持，我们将在后面的章节介绍对这些函数的处理。另一类是普通函数，这类函数被处理成非解释函数(uninterpreted functions)，即，对于函数F(x,y)，仅有如下假设成立：若x1=y1且x2=y2，则F(x1,x2)=F(y1,y2)。这里CBOC没有考虑这类函数的副作用。

4.4 基址安全表达式

C语言程序中频繁地使用到指针，指针和指针运算是缓冲区溢出漏洞中至关重要的因素。指针变量的解引用可以通过指针分析来进行处理。如果计算出指针变量的指向关系，则可在对指针进行解引用的地方进行指向集替换。例如，若已知p的指向集为{x,y}，则*p可以由&x，&y来替换，语句i=*p可以替换成 if(p==&x) i=x; else if(p==&y) i=y; else assert(0)。然而，指针分析问题是NP-完全问题，当允许动态内存分配、递归数据结构、循环和分支时，指针分析问题甚至是不可计算的。另外，C语言复杂的语义导致可靠的指针分析可能产生大量虚假的指向关系。为了减少误报，现有的很多软件验证工具常常使用不可靠的指针分析。例如：在广泛用于操作系统错误检查的Metal编译器和Intrinsa系统中，两个指针互为别名，当且仅当这两个指针被证明指向相同位置或对象。在缓冲区溢出检测中，很多检测工具也遵循了上述原则。然而，使用不可靠的指针分析导致缓冲器溢出检测工具不能保证检测的可靠性（产生误报）。

指针表达式解引用更加难以处理。指针的存在使得程序访问的空间不是平坦的，考虑到效率和可实现性等因素，很多程序分析工具都不支持或者仅有限支持指针和指针表达式。即使最常用的数组也会给程序分析工具（尤其是基于谓词抽象的工具，如：BLAST和SLAM）带来很大挑战。包括CBMC在内的很多程序分析工具将数组看作一个整体进行更新，更新数

组的一个元素时，需要对整个数组进行更新。例如，设A是一个包含两个元素的数组。语句A[x]=10将被替换成语句序列A[0]=(x==0)?10:A[0]; A[1]=(x==1)?10:A[1].来执行。

CBOC引入基址安全表达式的概念，基址安全表达式的等价性可以从形式上方便地进行判断，这就为符号执行过程提供了较大的优化空间。本节详细讨论基址安全表达式。

4.4.1 规范型左值表达式

C语言中，表达式分为左值表达式和右值表达式，左值表达式同时也是右值表达式。只有左值表达式对应到程序中实际的存储块，可以出现在赋值符号的左边。CBOC所采用的表达式规范以CIL为基础。图3给出了CIL中表达式的定义。其中，exp定义了表达式的全集，lval定义了左值表达式，constant定义了常量表达式，lhost定义了左值表达式引用的存储块的起始地址，offset定义了相对偏移，biop和unop分别定义了二元操作和一元操作，Var为变量集合，typ为变量类型的集合，fieldinfo为结构体的域，偏移包括0偏移NoOffset，域偏移（结构体）Field和索引偏移Index，CastE 为类型强制转换符。StartOf(lval)表示数组的起始位置，等同一个指针，AddrOf对应到C语言中的&操作符，Mem对应C语言中的*操作符，SizeOf，SizeOfE，AlignOf，AlignOfE分别表示不同类型的变量及表达式的大小和对齐方式，SizeOfStr表示常量字符串的长度。CIL还引入StartOf来简化类型转换规则。

```
exp = Const of constant | Lval of lval | SizeOf of typ
    | SizeOfE of exp | SizeOfStr of string | AlignOf of typ
    | AlignOfE of exp | UnOp of unop*exp*typ | BinOp of binop*exp*exp*typ
    | CastE of typ*exp | AddrOf of lval | StartOf of lval
constant = CInt64 of int64*ikind*string option | CStr of string
    | CWStr of int64 list | CReal of float*fkind*string option
    | CEnum of exp*string*enuminfo | CChr of char
lval = lhost*offset
lhost = Var of varinfo | Mem of exp
offset = NoOffset | Field of fieldinfo*offset | Index of exp*offset
binop::=PlusA | PlusPI | IndexPI | MinusA | MinusPI | MinusPP | Mult
    | Div | Mod | Shiftlt | Shiftrt | Lt | Gt | Le | Ge | Eq | Ne | BAnd | Bxor
    | BOr | LAnd | LOr
unop::= Neg | BNot | LNot
```

图 3　CIL 中表达式的定义

Fig.3　definition of expressions in **CIL**

C语言中的表达式都可以写成CIL的文法形式。例如，左值表达式*((int*)100)可以写成(Mem CastE(intType, (integer 100)))，其中intType是TInt(IInt,[])的缩写，integer 100是Const(CInt64(Int64.of_int 100, IInt, None))的缩写，x.b可以写成(x,Field(b,NoOffset))。为了叙述方便，在不引起混淆时，本文使用原始的C表达式作为CIL文法形式的表达式的简写。

定义 1：不含域偏移（Filed）和索引偏移（Index）的 CIL 表达式称为规范型表达式。也就是说，规范型表达式中只包含 0 偏移 NoOffset。

借助指针运算，很多表达式都可以变换为等价的规范型表达式（这里讨论等价时，不考虑编译过程由于对齐方式导致的差异，这种差异不影响溢出检测结论。因此，不要求表达式在编译后仍然等价，只需要在源码中表达式语义相同）。例如，假设x为struct { int a[3]; int b[2];}类型的变量，则表达式x.a[1]等价于*((int*)((unsigned)&x+sizeOf(int)))，表达式x.b[1]等价于*((int*)((unsigned)&x+4* sizeOf(int)))。但是，结构体中的bit域（“bit field”）相对结构体起始地址的偏移不是整数个字节，没有规范型表达式与之等价。bit域是结构体（或联合体）中的特殊的成员，它占用的空间以位（而不是字节）为单位。幸运的是，利用C语言中固有的位操作符可以消除结构体中bit域在程序中的出现。具体方法是：为每一个bit域b关联一个掩模字段b_m，假设b有n位，是b_h的成员，且相对于b_h的偏移为o位。对给定的结构体变量b_h和成

员b，o可以在编译时计算出来。不失一般性，设sizeOf(unsigned)等于32，且n+o小于32。掩模b_m是一个无符号整数“$b_{31}b_{30}\ldots\ldots b_0$”，其中，当$o \le i < n+o$时，$b_i$=0，其它情况下$b_i$=1。在需要对b进行读操作时，使用“(*(unsigned*)(&b_h) & (～b_m)) >> o”代替b，在需要将w写入到b中时，使用“*(unsigned*)(&b_h) = ((*(unsigned*)(&b_h) & b_m) | (～b_m & (w << o)))”来替换这次写操作。其中“～”是按位取非操作，二元操作符“&”是按位与操作，“|”是按位或操作，“<<”和“>>”分别是按位左移和按位右移操作。图4给出一个消除bit域的例子。

```
struct mybf {int a:4; int b:5;}test; int a; /*变量定义*/
void main() {test.b= 10; a = test.b;} /*原始程序*/
void main (){   /*消除bit域之后的程序*/
      unsigned mb = 0xFE0F;
      *(unsigned*)(&test) = (*(unsigned*)&test & mb) | ((～mb) & (10 << 4));
      a = (*(unsigned*)(&test) & (～mb)) >> 4;
}
```

图 4　消除 bit 域的出现

Fig.4　eliminate “bit field”

定理 1：任意一个不包含 bit 域的 CIL 表达式，一定存在一个规范型表达式与之等价。

上述定理可以用结构归纳法加以证明。对应于该结构归纳证明，任意一个不含bit域的左值表达式，都可以使用一个递归算法将其化为等价的规范型表达式。右值表达式中包含的左值表达式都规范化后就得到该右值表达式对应的规范表达式。限于篇幅，本文仅给出将不含bit域的左值表达式化为规范型的算法，如图5所示。其中，函数offsetToInt递归地计算表达式中包含的偏移量，它以一个类型t和一个偏移o作为输入，返回偏移量的符号表达式。例如，给定类型struct { int a[3]; char b[2];}和偏移Field(b,Index(1, Nooffset))，它将返回3×sizeOf(int)+sizeOf(char)。函数normalLval调用offsetToInt将表达式化为规范型。这里的算法假设表达式已经进行了必要的化简，将对指针变量的索引操作化为指针运算。例如，设x定义为int* x，尽管C语言中允许使用x[2]来代替*(x+2)，但CBOC预先将x[2]化为*(x+2)的的形式。

```
function offsetToInt(t, o) = begin
  switch (o)
    case  NoOffset ： return 0
    case  Field(fi, off) ：
          /* t一定是结构体类型，且fi是该结构体的一个成员。
           因为v中不含bit域，所以f相对于结构体起始位置的
           偏移一定是整数个字节，且可以在编译器确定，
           设为c。设fi的类型为t1 */
          return  c + offsetToInt(t1, off)
    case  Index(ei, off) ：
          /* t一定是数组类型，设其元素的类型为t1 */
          return  ei × sizeOf(t1) + offsetToInt(t1, off)
end
```

```
function normalLval (lv) = begin
  switch (lv)
    case  (Var v, NoOffset) ： return lv
    case  (Var v, off) ：
          /* 设左值表达式lv的类型为t1，变量v的类型为t2 */
          c = offsetToInt (t2, off) ;
          return  *((t1*)((unsigned)&v+c))
    case  (Mem a, off) ： return lv
          /* 设左值表达式lv的类型为t1；
            左值表达式(Mem a, NoOffset)的类型为t2  */
          c = offsetToInt (t2, off) ;
          return  *((t1*)((unsigned)a+c))
end
```

图5　将表达式化为规范型

Fig.5　transform expression to its normalized form

任意一个表达式，可以消除表达式中的bit域，并使用normalLval将其规范化，表达式最终都会被化为*e的形式。例如，设a和b分别是类型为 struct {int a[10];int b[5];}和struct {int* a;int* b;}的结构体变量，并设sizeOf(int)等于32，a.b[2]和b.b[2]将分别被化为*((int *)((unsigned)(&a) + 48)) 和*(*((int **)((unsigned)(&b) + 4)) + 2)。本文后面的章节将假设程序中的所有左值表达式都具有*e的形式。

4.4.2 基址安全表达式

C语言中的很多表达式，即使能通过编译检查，在使用的过程中也可能导致不可意料的后果。例如，arr[(int)&x]，*((int*)100)。这些左值表达式试图访问不确定的或受限制的地址空间。在程序语句中使用这些表达式，如：arr[(int)&x]=10，*((int*)100)=10将会导致运行时错误。本文认为这类表达式是不合法的，不应该在程序中使用。这里我们不考虑静态规划好的数据区，例如，在嵌入式程序设计中，通常划定一片空间为数据区，可以使用形如*((int*)100) 的表达式对这些数据区进行访问，此时，必须由程序员来保证访问不超出数据区，程序分析工具无法从源代码中进行检查。

事实上，程序中所有可以合法访问的存储块包括三类：一类通过全局变量定义来静态分配空间，另一类通过局部变量定义动态地在执行栈里分配空间。还有一类通过malloc，free等存储管理函数来动态地在堆空间中分配和释放（为方便叙述，本文使用malloc来统一表示所有内存分配函数，使用free来统一表示所有内存释放函数）。若g是全局变量，可以用&g来标记分配给g的存储块的起始地址，并作为该存储块的ID。对函数进行内嵌处理后，分配给局部变量l的存储块的起始地址及ID可以用&l来标记。在进行循环展开和函数内嵌后，动态分配的存储块的个数也可以通过计算代码中malloc出现的次数得到。为每个动态分配的存储块b引入一个虚拟变量vb，并用&vb表示存储块b的起始位置并作为b的ID。那么，所有可以合法访问的存储块都以&x开始，所有可以访问的存储地址都可以用一个形如*(&x+O)的规范型表达式来表示，其中x为程序中定义的某个变量或虚拟变量，O是一个表达式，表示该存储地址相对宿主存储块的整数字节偏移。当O为0时，*(&x)就是x。因此，合法的存储地址都可以通过形如*(&x+O)的表达式进行访问。前一节我们已经讨论过，每一个C语言左值表达式都可以化为*e的规范形式。这里，我们引入基址安全（site-safe）表达式的概念来进一步对表达式加以限制。

定义2：对程序位置L处的左值表达式lv，从程序开始位置对程序进行符号执行，当符号执行到达程序位置L时，若lv可以化简成为*(&x+z)的形式，其中，x是程序变量或虚拟变量，z为整数，则称lv是基址安全（site-safe）的左值表达式。表达式是基址安全的，当且仅当它所包含的所有左值表达式都是基址安全的。

上面的定义基于这样的考虑：若程序可能访问的所有地址组成的集合与用户的输入有关，则用户可以采用精心构造的输入来让程序访问非法的地址空间。定义 2 要求左值表达式访问的存储块有一个合法的起始位置，但没有对偏移z的大小进行限制，因此，称这样的表达式为基址安全表达式。

需要注意的是：判定表达式是否是基址安全的是在符号执行的过程中进行的，是一种动态的行为，而不是依据表达式在程序代码中的形式来静态地进行判断，也就是说基址安全是相对于一条符号化执行路径而言的。例如，对如下的程序片段：

int *p; L1: p=malloc(100); if(y){ L2: p=p+x-x+2;} else {L3: p=p+x;} L4: *p=10;

其中x，y是输入变量，程序位置L4处的指针表达式*p相对于执行路径L1;L2;L4来说是基址安全的；而相对于执行路径L1;L3;L4来说，则不是基址安全的。

对于不同变量x1和x2，它们的地址&x1和&x2在编译时确定，且必然不相等，它们之间也不存在确定的线性关系。因此，有如下推论：

推论1：基址安全的表达式*(&x1+z1)和*(&x2+z2)相等，当且仅当x1和x2是同一个变量，且z1=z2。

这一推论说明：任意两个基址安全的左值表达式是否相等可以方便地从形式上进行判断。正是基址安全表达式的这一优点，使得CBOC能够极大的简化缓冲区溢出漏洞的检测过程，并提高检测效率。

基址安全表达式要求符号执行过程中左值表达式中的偏移部分是整数，而不是可以包含输入符号的普通表达式。这使得该定义对表达式的限制太强，难以处理实际程序中一类典型的情况：程序员可能会使用用户输入作为偏移量来索引程序中的数组，并且采用条件判断语句对用户输入进行检查，保证程序不会越界访问。这类情况需要结合路径条件进行处理。但是，基址安全表达式保证了左值表达式的偏移部分是整数，使得左值表达式的等价性判断非常容易，这就允许我们采用优化的符号执行过程来进行缓冲区溢出漏洞检测。本文第五节的实验结果也显示：很多包含缓冲区溢出的实际应用程序，其表达式都是基址安全的，优化的符号执行算法能高效地发现这些代码中包含的缓冲区溢出。因此，相对于基址安全表达式给检测算法在性能上所带来的优化和提升，基址安全表达式的负面效果是可以接受的。

4.5 优化的符号执行过程

符号执行的主要思想是使用符号值而不是实际的数据来代表程序输入，并在执行过程中产生关于输入符号的代数表达式。符号执行每一步的状态包括三个内容：路径条件（Path Condition, PC)、程序计数器与程序变量的符号值。PC是基于输入符号的布尔型公式，它积累程序执行的每一步输入变量必须满足的约束。当PC表达的约束可满足时，说明对应该PC的路径是可达的。否则，对应该PC的路径不可达。程序计数器定义了下一个要执行的语句，通常用程序位置表示。

CBOC利用符号执行技术检测C程序中的缓冲区溢出漏洞。使用经典的符号执行算法，难以解决数组元素的混淆，复杂的数据结构，指针与引用的混淆等问题。基址安全表达式可以方便的进行等价性判断，这为CBOC的符号执行过程提供了优化的空间。优化主要体现在指针的解引用上，两个指针是否指向相同的存储位置可方便地通过表达式的等价判断得出。因此，CBOC可以使用哈希表来维护基址安全的左值表达式（而不是变量）和其符号值的对应关系。每个基址安全的左值表达式对应一个存储地址，在哈希表中，每个表达式只会有一个对应的符号值。假如哈希表中的左值表达式不是基址安全的，很难判断两个表达式是否等价（互为别名），这样可能导致哈希表中同一存储地址有多个符号值，且不能同步更新。

4.5.1 赋值语句

图6给出了CBOC中符号执行赋值语句lv=e的算法，其中lv是左值表达式，e是普通表达式。函数assign以lv和e作为输入，将赋值语句执行后的程序状态更新到state中。state是一个全局的哈希表，它将基址安全的表达式映射到表达式当前的符号化值。显然，基址安全表达式高效的等价性判断是建立哈希表state的前提。函数symbVal以表达式e为参数，计算e的符号化值，计算过程中可能会引入虚拟变量（遇到用户输入变量时），并将虚拟变量的符号化值更新到state中。算法假设程序已经经过预处理：所有的表达式都是规范型表达式，变量初始化已经转化为等价的赋值语句。算法的主要思想是：从程序入口点开始符号执行，将对左值表达式的修改更新到state中，在对未赋值的左值表达式lv进行读操作时，需要引入一个新的输入符号，表示lv的符号值并将其对应关系更新到state中。对已经赋值的左值变量进行读操作时，从state中获得变量的当前值。可以看出，该算法仅需要维护被使用过的变量的当前值，而无需像CBMC一样，当更新数组中的一个元素时，需要同时更新整个数组的问题。

```
01 function symbVal(e) = begin
02        for each sub expression e_i of e do symbVal (e_i)
03        do constant folding on e
04        if e is a lvalue, do
05              if e is not site-safe, raise an exception
06              otherwise, do
07                    if a symbolic value S_e corresponds to e is found in state, then, return S_e
08                    otherwise, do
09                          get a new symbol S_n for e
10                          add a binding of e to S_n in state
11                          return S_n
12 end
01 function assign(lv, e) = begin
02        lv is a normalized lvalue, so lv has the form *b
03        let l = symbVal (b) and r = symbVal (e) in
04        if *l is not site-safe, raise an exception
05        otherwise, do
06              add a binding of *l to r in state
07 end
```

图 6　符号执行赋值语句

Fig.6　Symbolic execute an assignment

4.5.2 存储空间动态分配

为了检测缓冲区溢出漏洞，我们使用哈希表aloc维护存储块的相关信息，aloc将存储块的ID映射到该存储块的长度。根据变量的定义信息对aloc进行初始化。例如，对程序中定义的每一个全局变量（或每一个局部变量的实例）v，需要在aloc中添加一条记录，将&v映射到存储块的长度sizeOf(typeOf(v))。动态分配的存储块的信息同样维护在aloc中。图7给出处理动态存储空间分配的算法。函数doMalloc和函数doFree分别对应存储空间的分配和释放。算法的主要思想是：当符号执行到动态存储分配语句lv=malloc(e)时，引入一个新的虚拟变量x作为该存储块的ID，在aloc中添加一条记录，将x映射到对应存储块的长度e。当符号执行到存储空间释放语句free(e)时，symbVal (e)必须是某个存储块的ID，将该ID对应的存储块的长度更新为0即可。

```
01 function doMalloc(lv, e) = begin
02        lv is a normalized lvalue, so it has the form *b
03        let l = symbVal (b) and r = symbVal (e) in
04        if *l is not site-safe, raise an exception
05        otherwise, do
06              get a new symbol x to identify the allocated block
07              add a binding of x to r in aloc
08              add a binding of *l to &x in state
09 end
01 function doFree(e) = begin
02        let S_e = symbVal (e) in
03        if S_e is not a site-safe lvalue, raise an exception
04        otherwise, S_e has the form *(&x + z). do
05        if z is not constant 0, raise an exception
06              otherwise, do
07                    if the length l of the block corresponds to x is found in aloc, do
08                          if l is constant 0, raise an exception
09                          otherwise, replace l by 0 in aloc
10                    otherwise, raise an exception
11 end
```

图 7　符号执行动态内存分配语句

Fig.7　Symbolic execution of dynamic allocation

4.5.3 越界访问检查

有了aloc，越界访问的检查相当简单。左值表达式*(&x+z)将引用ID为&x的存储块中偏

移量为z的存储地址。使用&x在aloc中查询到存储块的长度，记为len，若z>len可满足（使用SMT求解器[13]对表达式的可满足性进行判断），则说明*(&x+z)引用的存储地址可能超过了存储块&x的边界。

4.5.4 字符串操作函数导致的溢出检查

C库中的一些函数，如果使用不当，它们将可能导致缓冲区溢出。这类函数包括gets，strcpy, strcat, sprintf，memcpy, strncpy等。这些函数的源代码往往不公布，基于源代码的漏洞检测工具不能直接处理这类函数。CBOC采用一种直观的方式来处理这类函数：为这些函数提供简单的源代码来刻画这些函数的操作过程，然后将这些函数当作用户定义的普通函数来处理，这样字符串操作函数导致的缓冲区溢出问题就被转化为越界访问问题。作为例子，图8给出了函数strlen和strcpy的简单代码。

```
char*  strlen(char* s){
        int i = 0;
        while (s[i] != 0) i++;
        return i;
}
char*  strcpy (char*  d, char*  s){
        int i;
        for (i = 0; ; i++) {
                d[i] = s[i];
                if (s[i] == 0) break;
        }
        return d;
}
```

图 8 strlen 和 strcpy

Fig.8 strlen and strcpy

5. 实验结果

与CBMC一样，CBOC使用一个循环展开参数（-unwind）来控制循环展开的遍数。CBMC报告两种类型的错误：overflow和non-site-safe。当报告overflow时，说明在程序中发现了一个真实的缓冲区溢出漏洞，当报告non-site-safe时，说明在程序中存在某些表达式不是site-safe的。当程序中的某些循环展开不充分时，CBOC还会发出警告：unwinding assertion violation。此时，需要增大循环的展开遍数来查找更长的路径中包含的缓冲区溢出漏洞.。

表 1 实验数据

Table 1 Experiment results

软件名称	漏洞编号	循环展开遍数	CBOC		CBMC	
			时间 (1/10s)	空间 (MB)	时间 (1/10s)	空间 (MB)
SpamAssassin	BID-6679	10	**6**	**5.4**	124	132.3
bind	CA-1999-14	10	**10**	**5.4**	13	11.8
sendmail	CVE-1999-0047	5	**7**	**5.4**	341	244.5
sendmail	CVE-1999-0206	5	**5**	**5.4**	22	30.9
wu-ftpd	CVE-1999-0368	10	7	**3.9**	**6**	4.6
bind	CVE-2001-0011	15	**5**	**5.4**	12	11
sendmail	CVE-2001-0653	15	**5**	**5.4**	32	16.3
sendmail	CVE-2002-0906	5	**5**	**5.4**	7	7.3
sendmail	CVE-2002-1337	15	**5**	**5.4**	12	12.8
sendmail	CVE-2003-0161	5	5	**5.4**	5	5.6
wu-ftpd	CVE-2003-0466	10	11	**5.4**	**9**	8.2
sendmail	CVE-2003-0681	5	**5**	**5.4**	11	11.1
apache	CVE-2004-0940	4	162	**5.4**	**12**	12.4

apache	CVE-2006-3747	15	**10**	**5.4**	17	18.4
MADWiFi	CVE-2006-6332	5	5	5.4	**4**	**3.4**
NetBSD-libc	CVE-2006-6652	5	**5**	5.4	>50000	>900
OpenSER	CVE-2006-6749	5	**7**	**5.4**	35	14.1
OpenSER	CVE-2006-6876	10	**5**	**5.4**	7	6.4
edbrowse	CVE-2006-6909	4	**64**	**5.4**	178	166.4
gxine	CVE-2007-0406	5	5	5.4	**4**	**3.4**
samba	CVE-2007-0453	5	**5**	5.4	11	**3.6**
libgd	CVE-2007-0455	5	**5**	**5.4**	>878	>900

使用Kelvin Ku等人[14]提供的测试包对CBOC进行了验证。该测试包收集了来自apache，bind，sendmail，NetBSD-libc，gxine等12个著名的开源软件的298个真实代码片段，包含了22个CVE[15]发布的缓冲区溢出漏洞。测试使用的硬件环境为：1.66GHz Intel Core2 处理器，1G内存。我们使用工具run[16]对工具的运行时间和内存使用情况进行统计，并设置工具运行的时间上限为50000秒，内存占用上限为900MB。在相同的条件，我们将CBOC和CBMC进行比较。表1给出详细的比较结果。可以看到，在大多数测试中，CBOC的性能都优于CBMC。特别地，CBOC使用内存很少，因为CBOC执行过程中只维护栈顶的一个程序状态，而像CBMC这样的模型检验工具通常同时保存访问过的所有状态。

在测试过程中，CBOC准确地定位了测试包中除CVE-2002-0906之外的21个缓冲区溢出漏洞，而漏洞CVE-2002-0906被报告为一个non-site-safe错误。可见，实际应用中的很多包含缓冲区溢出程序，其表达式都是基址安全的，CBOC优化的符号执行算法能高效地发现这些代码中包含的缓冲区溢出。因此，相对于基址安全表达式给检测算法和工具在性能上所带来的优化和提升，基址安全表达式所带来的负面效果是可以接受的。

在测试过程中，CBOC还发现了因测试包作者的疏忽所导致的两个问题。一个问题出现在CVE-2006-6652的测试代码中，文件glob3_int_bad.c，第69行。指针表达式*pathlim试图访问整数数组A中偏移量为44字节的存储空间。但A中只包含3个元素（12字节）。图9是从文件glob3_int_bad.c中抽取的一段代码。在图9中第10行，sizeof(A)的值为12，因此，变量bound的值将为A+11。但pathlim是一个整数指针，它将引用从A开试偏移11个整数（也就是44字节）的地方。这个漏洞并不是开源软件中真实存在的，而是由于测试包的作者在编写测试程序过程中的疏忽导致的。

另一个问题出现在CVE-2006-6876的测试代码中，文件loops_bad.c被标记为包含了缓冲区溢出漏洞。使用CBOC进行检查时没有发现漏洞。通过人工检查，该文件确实不包含缓冲区溢出漏洞。

```
01 typedef int Char;
02 static int glob3(Char *pathbuf, Char *pathend,
03                   Char *pathlim, Char *pattern) { ...
04           if (pathend + i > pathlim && *pathlim != 0)
05           {...} ...
06}
07 int main () {
08           Char *buf, *pattern, *bound, A[3], B[7];
09           buf = A; pattern = B;
10           bound = A + sizeof(A) - 1;
11           glob3(buf, buf, bound, pattern);
12           return 0;
13 }
```

图 9　测试包中的一个错误

Fig.9　a mistake in the benchmark

CBOC通过遍历程序的所有有限长的路径来查找程序错误。理论上不存在误报，也能发现有限长的路径中包含的所有缓冲区溢出漏洞。然而在具体实现的过程中，可能会存在误报和漏报，原因包括：由于C语言的语义极为复杂，为兼顾效率，设计过程中可能会作一些近似；某些API调用（如pthread_create，fork，setitimer等）导致的多线程和控制流变化；shmget等API函数导致的内存共享；以及工具原型尚未刻画的函数，如：fgets，getcwd等。

我们使用Kendra Kratkiewicz[17]提供的一个测试包对CBOC的漏报率和误报率进行了详细考察。该测试包收集了1164个精心编制的代码片段，覆盖了多种类型的缓冲区溢出漏洞。其中，291个片段没有溢出漏洞，其余的873个代码片段中包含漏洞。使用CBOC的检测结果显示：工具正确区分出290个没有漏洞的代码片段，误报1个，误报率约为0.34%，准确定位了858个溢出漏洞，漏报15个，漏报率约为1.72%。

6. 总结

本文给出一个基于符号执行的C语言缓冲区溢出漏洞检测工具CBOC。CBOC通过引入基址安全表达式的概念，对CBOC的符号执行过程进行了优化，从而缓解了指针、数组等给程序分析带来的复杂度，提高了检测效率。实验表明：相对于基址安全表达式给检测工具带来的巨大优化空间，基址安全表达式所带来的负面效果是可以接受的。这说明基址安全表达式的引入对大概率出现的事件给予了更多优化，符合“大概率事件优先”的原则。在大多数测试中，CBOC的性能都优于著名的C语言限界模型检验工具CBMC，且CBOC使用内存较少，误报率和漏报率都很低。CBOC可用于发现缓冲区溢出漏洞，给出错误路径，帮助开发人员理解和分析漏洞。

CBOC采用符号执行技术，存在如下固有的局限性：

1) 不能克服分支和循环等导致的路径数目指数级增长的问题，难以对大规模软件系统进行检测；

2) 循环展开次数不易确定。如果程序中包含不终止的循环，参数调整的过程可能不收敛。

参考文献

[1] Bodik R, Gupta R, Sarkar V. ABCD: Eliminating array-bounds checks on demand [A]. Programming Language Design and Implementation (PLDI), Vancouver: ACM, 2000: 321-333.
[2] Condit J, Harren M, McPeak S, et al. CCured in the real world [A]. Programming Language Design and Implementation (PLDI), San Diego: ACM, 2003: 232-244.
[3] D. Wagner, J. Foster. A first step towards automated detection of buffer overrun vulnerabilities [A]. Network and Distributed System Security Symposium (NDSS), San Diego: CA, 2000: 3-17.
[4] Ganapathy V, Jha S, Chandler D, et al. Buffer overrun detection using linear programming and static analysis [A]. Computer and Communications Security (CCS), Washington: ACM, 2003: 345-354.
[5] Dor N, Rodeh M, Sagiv S. CSSV: towards a realistic tool for statically detecting all buffer overflows in C [A]. Programming Language Design and Implementation (PLDI), San Diego: ACM, 2003: 155-167
[6] Chaki S, Hissam S. Precise Buffer Overflow Detection via Model Checking [R]. SEI, CMU, 2005.
[7] Chaki S, Hissam S. Certifying the Absence of Buffer Overflows [R]. CMU/SEI-2006-TN-030, SEI, CMU, 2006.
[8] Clarke E, Kroening D, Lerda F. A Tool for Checking ANSI-C Programs [A]. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Barcelona: Paris: Springer, 2004: 168-176.
[9] Thomas Ball, Sriram K. Rajamani. The SLAM Toolkit [A]. Computer Aided Verification (CAV), LNCS, 2001:260–264.
[10] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, et al. Lazy Abstraction [A]. SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Portland: ACM, 2002:58–70.
[11] James C. King. Symbolic Execution and Program Testing [J].Commun. ACM 19(7): 385-394, 1976
[12] Necula G, McPeak S, Rahul S, et al. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs [A]. Compilier Construction (CC), Grenoble: Springer, 2002: 213-228.

[13]Moura M, Dutertre B, Shankar N. A Tutorial on Satisfiability Modulo Theories [A]. Computer Aided Verification (CAV), Trento: Springer, 1999 2007: 20-36
[14]Ku K, Hart T, Chechik M, et al. A buffer overflow benchmark for software model checkers [A]. Automated Software Engineering (ASE), Atlanta: IEEE/ACM, 2007: 389-392
[15]http://cve.mitre.org/. National Cyber Security Division of the U.S. Department of Homeland Security, 2009.
[16]J. K. U. Armin Biere, ETH Zurich. http://fmv.jku.at/run/, 2008.
[17]Kratkiewicz K. Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code [D]. USA:Harvard University 2005.

CBOC : An efficient tool to detect buffer overflows in C source code

Chen Shi-Kun[1], Li Zhou-Jun[2]
1(School of Computer Science, National University of Defense Technology, Changsha 410073)
2(Computer Science and Engineering School, Beihang University, Beijing 100191)

Abstract

Buffer overflows are the source of a vast majority of security issues in C program. This paper presents a tool called CBOC (C Buffer Overflow Checker) to detect buffer overflows in C source code. It bases on symbolic execution and introduces the notion of site-safe expression to optimize the process of symbolic execution. CBOC takes as input source code, and can identify buffer overflows in the source code automatically. CBOC was applied to a publicly-available benchmark and compared against state-of-the-art ANSI-C bounded model checker CBMC. Experiments show that CBOC exceeds CBMC in most of the test cases and the memory usage of CBOC is very low. In the accuracy test experiments, the false alarm rate of CBOC is about 0.34% and omission rate is about 2.08%.

Keywords: buffer overflow; symbolic execution; site-safe expression

作者简介：

陈石坤(1980－),男,博士研究生,主要研究领域为信息安全,软件验证;

李舟军(1963－),男,博士,教授,主要研究领域为高可信软件技术,网络与信息安全技术,智能信息处理技术