计算机科学
COMPUTER SCIENCE

结合模糊测试和动态分析的内存安全漏洞检测

马莺姿, 陈哲, 殷家乐, 毛瑞琪

引用本文

马莺姿, 陈哲, 殷家乐, 毛瑞琪. 结合模糊测试和动态分析的内存安全漏洞检测[J]. 计算机科学, 2024, 51(2): 352-358.

MA Yingzi, CHEN Zhe, YIN Jiale, MAO Ruiqi. Memory Security Vulnerability Detection Combining Fuzzy Testing and Dynamic Analysis [J]. Computer Science, 2024, 51(2): 352-358.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.221200136

结合模糊测试和动态分析的内存安全漏洞检测

马莺姿　陈　哲　殷家乐　毛瑞琪
南京航空航天大学计算机科学与技术学院　南京 211100
(18864820270@163.com)

摘　要　C语言因其在运行速度及内存控制方面的优势而被广泛应用于系统软件和嵌入式软件的开发。指针的强大功能使得它可以直接对内存进行操作,然而C语言并未提供对内存安全性的检测,这就使得指针的使用会导致内存泄露、缓冲区溢出、多次释放等内存错误,有时这些错误还会造成系统崩溃或内部数据破坏等的致命伤害。当前已存在多种能够对C程序进行内存安全漏洞检测的技术。其中动态分析技术通过插桩源代码来实现对C程序的运行时内存安全检测,但是只有当程序执行到错误所在路径时才能发现错误,因此它依赖于程序的输入;而模糊测试是一种通过向程序提供输入并监视程序运行结果来发现软件漏洞的方法,但是无法检测出没有导致程序崩溃的内存安全性错误,也无法提供错误所在位置等详细信息。除此之外,由于C语言的语法比较复杂,在对一些大型复杂项目进行分析时,动态分析工具经常无法正确处理一些不常见的特定结构,导致插桩失败或者插桩后的程序无法被正确编译。针对上述问题,通过将动态分析技术与模糊测试技术结合,并对已有方法进行改进后,提出了一种能够对包含特定结构的C程序进行内存安全检测的方法。文中进行了可靠性和性能的实验,结果表明,在增加对C语言中特定结构的处理方法之后,能对包含C语言中特定结构的程序进行内存安全检测,并且结合模糊测试技术后具有更强的漏洞检测能力。

关键词:内存安全;源代码插桩;动态分析;模糊测试;内存错误
中图分类号　TP311

Memory Security Vulnerability Detection Combining Fuzzy Testing and Dynamic Analysis

MA Yingzi,CHEN Zhe,YIN Jiale and MAO Ruiqi
College of Computer Science and Technology,Nanjing University of Aeronautics and Astronautics,Nanjing 211100,China

Abstract　C language is widely used in the development of system software and embedded software due to its high speed and precise control of memory through pointers,and is one of the most popular programming languages. The power of pointers makes it possible to operate directly on memory. However,C does not provide detection of memory security,which makes the use of pointers can lead to memory errors like memory leaks,buffer overflows,multiple releases,and sometimes these errors can cause fatal damage such as system crashes or internal data corruption. At present,there are some techniques that can detect memory security vulnerabilities in C programs. Among them,dynamic analysis technique can detect memory safety of C programs at runtime by staking the source code,but it can only find the error when the program executes to the path where the error is located,so it relies on the program's input. While fuzzy testing is a method to find software vulnerabilities by providing input to the program and monitoring the program's operation results,but it cannot detect memory safety errors that do not cause the program to crash,nor can it provide detailed information such as the location of the error. It also does not provide detailed information such as the location of the error. In addition,due to the complex grammar of the C language,dynamic analysis tools often fail to correctly handle some uncommon specific structures when analyzing large and complex projects,resulting in stubbing failures or stubbed programs not being compiled correctly. To address these problems, this paper proposes a method that can detect the memory safety of C programs containing specific structures by combining dynamic analysis techniques with fuzzy testing techniques and improving existing methods. The reliability and performance experiments show that with the addition of C-specific structures,the memory safety of programs containing C-specific structures can be detected,and the combination of the fuzzy testing technique can have stronger vulnerability detection capability.

Keywords　Memory safety,Source-level instrumentation,Dynamic analysis,Fuzzing,Memory errors

到稿日期:2022-12-23　返修日期:2023-04-21
基金项目:国家自然科学基金(62172217);国家自然科学基金委员会-中国民航局民航联合研究基金(U1533130);中央高校基本科研业务费人工智能+专项(NZ2020019)
This work was supported by the National Natural Science Foundation of China(62172277),Joint Research Funds of National Natural Science-Foundation of China and Civil Aviation Administration of China(U1533130) and Fundamental Research Funds of AI for the Central Universities of Ministry of Education of China(NZ2020019).
通信作者:陈哲(zhechen@nuaa.edu.cn)

1 引言

C 语言能够对内存进行精确的控制,并具有快速的运算能力,因此在系统软件和嵌入式软件中得到了广泛的应用。但是 C 语言没有提供内存安全检测机制,因此指针的使用可能会引起内存溢出、多次释放等问题,从而导致系统的崩溃和内部数据的丢失[1]。内存错误分为两类:空间内存错误和时间内存错误。空间内存错误包括缓冲区溢出、数组越界等;时间内存错误包括悬挂栈指针、悬挂堆指针、内存泄漏等[2]。

目前已有很多内存安全性动态分析工具,主要采用的方法包括内存对象技术和扩展指针技术。内存对象技术通过记录所有有效内存对象的地址范围,在通过指针访问内存时,检测当前访问是否在有效的地址范围内。这种技术可以检测出除子对象越界外的大部分内存错误,但插桩后的程序执行效率低下[3]。扩展指针技术对指针类型进行扩展,扩展后的指针类型除了包含内存地址外,还包括该指针指向内存块的基地址以及内存空间的大小等信息。但原有的指针类型和内存布局发生了改变,这就使得插桩后的程序存在兼容性问题,而且在检测过程中还需要对每个指针进行类型转换,导致插桩后的程序执行效率低[4]。

动态分析工具通常需要对程序的抽象语法树进行遍历,分别在对象创建、对象赋值、函数调用、对象释放、内存读写时把实现指针元数据更新和检测内存漏洞的语句插入源代码中指定的位置,生成插桩后的程序,最后编译运行插桩后的程序,并在运行时对源代码中的内存错误进行检测[5-6]。但是由于 C 语言的语法结构比较复杂,在采用源代码插桩技术对大型复杂项目进行分析时,无法正确处理一些不常见的结构,导致插桩失败或者插桩后的程序无法被正确编译。

本文介绍了对 C 程序中一些特定结构进行内存安全动态分析的方法。第 2 章介绍了与本文相关的基础知识,首先介绍了几种常见的动态分析工具和当前最先进的基于运行时验证的动态分析工具 Movec,然后对模糊测试工具 AFL 进行了详细介绍,最后介绍了实验中用到的测试集 CVE;第 3 章提出了能够对包含特定结构的 C 程序进行内存安全动态分析的方法,首先对已有方法进行了介绍,然后介绍了对 C 语言中特定结构进行动态分析的方法;第 4 章在增加了上述方法之后,利用当前最先进的动态分析工具 Movec,结合模糊测试工具 AFL 在 CVE 测试集上进行实验和分析,经过实验证明,改进的动态分析工具 Movec 与模糊测试工具 AFL 相结合能够更有效地检测程序中的漏洞;最后总结全文并展望未来。

本文的贡献包括:通过分析现实世界中的大量 CVE 程序,改进了现有的 C 程序内存安全动态分析方法,扩充了插桩框架能处理的语言结构;提出将模糊测试与动态分析相结合,并通过在这些 CVE 程序上的实验,证明了这种结合方式具有更强的漏洞检测能力。

2 主要技术

2.1 模糊测试

模糊测试是一种能够向程序提供随机输入的软件测试技术,它在发现现实世界程序中的漏洞方面非常有效。目前,已经使用模糊测试在各种软件中发现了数千个安全漏洞,证明了模糊测试使用简单并且有效[7-9]。

而 AFL 是一种基于代码覆盖率的模糊测试工具,它是应用最广泛且最先进的灰盒模糊测试器之一,也是首次使用源代码插桩和 QEMU 模式来实现通过代码覆盖引导进行模糊测试的方法,可以静态或动态地检测程序[10]。AFL 通过随机对提供的数据进行变异或者通过随机产生的种子来创建输入,目标源程序经过插桩编译后生成插桩后的可运行的二进制文件,并将初始种子文件提供给编译器进行变异后再提供给目标程序,以寻找程序中的漏洞[11]。因此,本文选择 AFL 作为模糊测试的工具来与动态分析工具结合,以提高对 C 程序进行内存安全动态分析的有效性。

AFL 进行模糊测试的流程如图 1 所示。

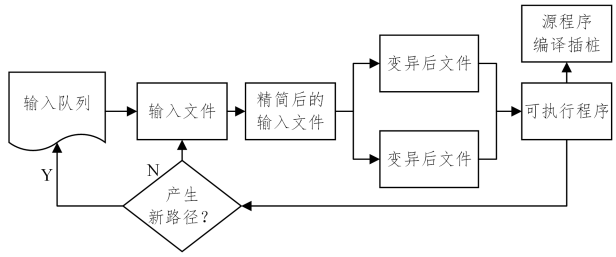

图 1 模糊测试流程图

Fig. 1 Fuzzy test flow chart

2.2 动态分析技术

ASan 是由谷歌研发的一种动态检测工具,主要是以影子内存技术为核心,所使用的是中间代码插桩方式。算法的思想是:在内存对象之间插入红色区域(Redzone),以检测缓冲区溢出;并将内存的状态信息记录到影子内存中,这样在对内存进行操作时可以通过影子内存判断该内存当前的状态。ASan 可以检测出释放后使用、多次释放、内存泄漏以及缓冲区溢出,但无法有效地检测子对象越界问题。此外,由于存储红色区域和影子内存表,并改变了内存分配器,执行了访问检测,使得整个程序的运行速度平均下降了 73%[12]。

SCet 由 SoftBound 和 CETS 两部分组成。在 SoftBound 中,通过记录每个指针指向内存的边界,可以检测空间内存错误[13]。CETS 无法检测出所有内存错误,它对对象访问内存的检测是通过为其维护的唯一的 ID 信息来完成的[14]。SCet 不能发现子对象越界等问题。

Valgrind 可以用来完成对程序的性能分析、检测其内存泄漏等操作,它是一个可以用来进行内存安全检测的重量级框架,可以对大多数的内存安全错误进行检测。它包括很多工具,而 MemCheck 是使用最广泛的内存分析工具,当读取和写入内存空间时,它可以检测程序中是否存在多次释放以及访问空指针的问题,但不能处理缓冲区溢出以及子对象越界问题[15]。

Movec 是一个在运行时可以自动检测内存错误的动态分析工具,它利用智能状态的监控算法,并在源代码级别执行插桩操作来实现对内存错误的检测[16]。Movec 的检测方法关注的主体是程序中指针的变化,它的实现是通过为程序中所有的指针变量构造一个指针元数据(pmd),其中存储的是

指针指向对象的上界(base)、下界(bound)和存储状态(snda),snda 中存储的是指针当前所指对象的状态(stat)和共享相同状态的指针个数(count)。所有指向同一对象的指针在其 pmd 空间中共享相同状态节点的地址,状态节点可以在其引用数目为 0 时自动删除。Movec 寻找漏洞算法的能力很全面,在访问超出追踪的边界时会检测到空间错误;当状态节点中的状态无效时会捕捉到时间错误;段混淆错误发生在使用追踪的方式与被跟踪的段类型不一致时;当被追踪的有效堆对象的引用计数为 0 时,会检测到内存泄露[17-18]。

由于动态分析技术通过插桩源代码实现对 C 程序的运行时内存安全检测,但是只有当程序执行到错误所在路径时才能发现错误,因此它依赖于程序的输入;而模糊测试是一种通过向程序提供输入并监视程序运行结果来发现软件漏洞的方法,但无法检测出没有导致程序崩溃的内存安全性错误,而且也无法提供错误所在位置等详细信息。因此,本文将动态分析技术与模糊测试技术结合,提高了对内存漏洞检测的有效性。

且经过实验证明,Movec 不存在已有动态分析工具面临的效率低下、优化敏感以及平台依赖的问题,而且它能够发现各种类型的内存错误,是当前最先进的动态分析工具。因此,本文选用 Movec 作为实验工具,来测试增加了处理 C 程序中特定结构后的内存安全动态分析方法的漏洞检测能力。

2.3 CVE

由 MITRE 创建并维护的通用漏洞披露(CVE)数据库全称为 Common Vulnerabilities and Exposures,是最大的公开可用的漏洞信息来源之一[19]。它收集了各种信息安全中的漏洞并为其赋予一个独一无二的编号。一旦被赋予编号,漏洞就会被列入公开的 MITRE 列表中。CVE 的作用是使得漏洞信息标准化,并统一安全专业人士之间的交流。目前,安全公告、漏洞数据库和错误追踪器都采用了该标准。

一旦报告了一个漏洞,CVE 编号机构(CNA)就会从它所持有的独特的 CVE 标识符块中为其分配一个号码。然后,CNA 向 MITRE 报告带有指定编号的漏洞。当一个 CVE 漏洞被公开时,会列出它的 ID、问题的简要描述,以及任何包含额外信息或报告的参考文献。随着新的参考资料出现,这些信息将被添加到该条目中[20]。

3 对包含特定结构的 C 程序的动态分析

当前已有的对 C 程序进行内存安全动态分析的方法主要利用的是源代码插桩框架。源代码插桩技术的实现是通过抽象语法树 AST 的访问接口来完成对 AST 的遍历,在遍历期间,根据读取节点类型的不同执行不同的插桩操作,并将插桩后的代码存储到新的文件中。插入的代码可以实现对 C 程序的动态分析,包括对指针元数据的创建与维护。

利用源代码插桩技术实现对 C 程序的内存安全动态分析,其插桩过程主要由以下 3 个步骤组成:1)插桩变量定义语句对指针元数据进行初始化,插桩变量赋值语句可以更新指针元数据,需要插桩的变量包括指针、包含指针成员的结构体、指针数组或包含指针的结构体数组等;2)在解引用指针变量时,内存安全动态分析通过读取指针变量的元数据完成,其中解引用可以是一元解引用,如 * p,p[i],p->m 或(* p).m 等,这里的指针通常包括变量、常量或函数调用;3)定义插桩函数,以初始化指针参数的元数据,然后计算并存储返回值的元数据;接着生成函数定义、函数声明的包装函数的声明和定义,包装函数可以传递实参的指针元数据,并更新返回值的元数据;检测的完成是通过重命名函数并插入新的函数后将函数调用重定向到为它生成的包装函数来实现的。

源代码插桩不同于中间代码以及二进制代码插桩,在对 C 程序的处理过程中,由于 C 语言语法结构复杂,对一些不常用但在大规模 C 程序中会出现的语法结构的处理容易被忽略,从而导致无法对程序进行插桩以及进行内存安全动态分析。在通过已有方法对大量 CVE 程序进行处理的过程中,发现了一些当前方法无法处理的 C 程序中的特定结构。

下文对已有方法无法处理的 C 程序中的特定结构进行了介绍,并提出了解决方法。

3.1 注释提前结束问题

在对 C 程序插桩之前,需要先预处理源程序,而预处理过程除了引入头文件以外,还包括宏的使用。当宏所表示的某个表达式可能是指针或者类指针的表达式时,需要对其插桩以完成对指针的检测。但由于宏所表示的词法单元在文件中不存在具体的位置,无法直接改写它的内容,因此,动态分析工具必须在预处理阶段进行宏展开操作,即将宏调用替换为宏定义中的内容,并注释掉宏定义和原来的宏调用,然后插桩程序。

在通过动态分析工具插桩 CVE-2019-14275 时会出现注释提前结束的问题,可通过如下程序复现出该问题。

```
#define func(a,b) strcmp(a,b)
void main() {
    if(func("a","abc*/def")== 0)
        return;
}
```

上述程序包含一个宏调用语句"func("a","abc*/def")",该宏调用中带有字符串参数,且字符串中包含注释结束符"*/"。对宏调用语句执行宏展开操作后的语句如下:

```
strcmp("a","abc*/def")/*func("a","abc*/def")*/
```

这里会出现注释提前结束的原因是,预处理过程中替换宏的内容后,需要注释掉替换前的宏内容,但当宏中包含有完整字符串,而字符串中又包含字符"*"与字符"/"相连的情况时,编译器会将上述两个字符识别为注释结束符,这就会导致对宏的注释被提前结束,从而使得插桩过程出现错误。

根据上文对"注释提前结束"问题的描述,在对源代码预处理的过程中,如果原来的宏调用带有字符串参数,且字符串中包含注释结束符"*/",则在"*"与"/"之间插入空格字符,以破坏其注释结束符的结构。上述改进后重新进行插桩操作时,不再出现注释提前结束的问题。

3.2 宏扩展问题

动态分析工具预处理过程中用到的宏替换算法借助的是编译器自带的词法分析器和一个原始词法分析器同时处理文件来实现的,它们的区别在于是否读取 include 指令或 define 指令。通过原始词法分析器和预处理词法分析器分别对源文件进行词法分析。在处理过程中,当两个词法分析器获取到

的词法单元处于同一文件时,会获取两个词法单元的偏移量,若获取到的偏移量相等且词法单元也相等时,则会判断当前没有使用宏,因此两个词法分析器都会继续向下读取,以获取新的词法单元,不会在此处进行宏扩展。

在插桩 CVE-2022-27044 时会出现注释失败的问题,原因是在宏替换算法中,没有考虑到当源程序中存在一个被定义为同名函数的宏时,两个词法解析器获取到的词法单元与词法单元的偏移量都相等。根据宏替换算法,会判断实际需要被替换的宏调用是无须被替换的,从而导致在插桩时对该宏的注释出现问题。例如,有如下语句:

```
#define test(x,y) test(x)
……
test("a","b");
```

在上面的代码中,宏定义中是同名的函数只是参数不同,根据宏替换算法,由于此时两个词法分析器获取到的词法单元偏移量与词法单元都相等,因此不会在此处进行宏扩展。

根据上面对"宏扩展"问题的描述,除了判断两个词法分析器的偏移量相等且词法单元也相等外,还需增加判断条件以确定此处是否存在宏的调用。当存在宏时,即使词法单元偏移量和词法单元都相等也要执行宏扩展操作,修改后再重新进行插桩操作,不再出现宏扩展问题。

3.3 对二元条件运算符的处理

动态分析工具只能插桩完整的三元条件表达式,例如"f()? f():y"。但对于条件表达式和真值表达式相同的三元条件表达式可以在 C 语言中被简写为二元条件操作符,例如"f()?:y"。

在抽象语法树中,三元条件表达式与二元条件操作符具有不同的节点类型,导致动态分析工具无法插桩二元条件操作符。例如,在插桩包含如下语句的 C 程序时,会出现上述问题。

```
void fun(char * x)
{f()?:x;}
```

根据对上述问题的分析,需要在已有算法的基础上增加对二元条件操作符类型的处理,由于其在抽象语法树中的类型为 BinaryConditionalOperator,且三元条件表达式在抽象语法树中的类型为 ConditionalOperator,而 AbstractConditionalOperator 是上述两个类型的父类,因此可以通过在已有算法中增加对 AbstractConditionalOperator 节点类型的处理,来同时实现插桩过程中对二元条件操作符和三元条件表达式的处理。

3.4 goto 语句跳过可变长度的数组声明问题

静态程序分析指在不执行程序的条件下,通过分析和检查程序源代码的执行路径、变量使用以及函数算法等,来发现程序中可能存在的缺陷。即当对源代码进行分析后,若其存在已有算法不能处理的特定结构时,就会在插桩过程中在终端给出修改提示,只有根据提示对源代码进行修改后,才能成功对其插桩。

在对程序的插桩过程中,为了防止 return,break,continue等语句跳过插入到源程序中的对指针元数据的清除操作,已有方法对上述语句进行了替换,其中就存在 goto 语句,用来执行跳转到指针元数据的删除语句,以实现对指针元数据的删除操作。此时若 goto 语句与其 label 之间存在其他语句,则运行插桩后的程序时会直接跳过这些语句。在插桩 CVE-2022-24793 时发现,若 goto 语句与 label 之间存在数组声明语句且数组长度是可变大小时,会导致插桩后的代码编译出现问题,如下所示:

```
for(int i=0;i < 3;++i)
{ ++ps_len;}
unsigned char ps_string[ps_len + 1];
```

由于上述代码中存在 for 语句,因此会进行替换,导致替换后的 goto 语句在运行时跳过了可变长度数组的声明语句"unsigned char ps_string[ps_len + 1];",从而在对其进行插桩操作时会出现错误,因此需要对源代码进行静态分析。

对所述情况进行静态分析的基本思想为:首先对源程序函数体中的语句进行分析,判断函数体中是否包含有复合语句、返回语句、条件语句、循环语句:如果有,则判断此时函数体中会被插入 goto 语句,直接跳转到函数体的结束位置,并用一个变量记录是否插入了 goto 语句;然后对函数体中的变量声明进行判断,只有当检测到当前声明语句是数组的声明且 goto 语句已经被插桩到此时的函数体中时才会进行后续判断;在满足上述条件的前提下,还需判断当前数组是否是可变大小的数组。当源代码满足上述所有要求时,就会在插桩期间向用户输出提示信息,并需要在修改后重新插桩源程序。

3.5 函数名与方法中的关键字冲突

由于插桩 C 程序源文件时,为了在运行时为指针变量创建指针元数据、检查指针访问是否存在内存错误,会向源代码中插入一些数据结构定义和接口函数定义。这些定义中包含以自定义的前缀 PREFIX 开头的标识符,例如类型 PREFIX-status 表示指针的存储状态;另一方面,为了传递实参指针元数据以及更新返回值指针元数据,需要包装程序中所有参数或返回值中含有指针成员的函数,这些包装函数的名字也是以自定义的前缀 PREFIX 开头的标识符,例如源代码中的 status 函数定义包装函数 PREFIXstatus。因此,当插入代码中的标识符与包装函数名相同时,标识符重定义会导致插桩后的程序无法正确编译。为了解决这个问题,插入的标识符与包装函数名需要使用不一样的前缀。

根据上述分析,在插桩源程序时,包装函数名都使用自定义的前缀 PREFIX,而数据结构定义和接口函数定义使用前缀_PREFIX,即在自定义的前缀 PREFIX 之前再添加一个下划线,从而避免标识符和包装函数名发生同名冲突。

3.6 sscanf 包装函数对参数 *format* 的处理

由于 *int sscanf*(*const char * str*,*const char * format*,…)的功能是根据参数 *format* 来将参数 str 中的内容读入到剩余参数中,因此在生成 sscanf 的包装函数时,为了检测对字符串的读入是否会导致内存错误,需要在包装函数中声明一个 *is_string* 数组,用于存储可变参数是否被当成字符数组使用。如果格式字符串 *format* 中的某个转换规约是"%s",则意味着该转换规约对应的可变参数被当成字符数组使用,因此将 *is_string* 数组的相应元素设置为 1;如果格式字符串 *format* 中的某个转换规约不是"%s",而是"%d"等,那么将 *is_string* 数组的相应元素设置为 0。但是动态分析工具通常忽略了一些不常用的转换规约,例如"%[]",将其 *is_string* 数组的相应元素也设置为 0,从而跳过对其内存访问的检测,然而该

转换规约对应的可变参数也被当成字符数组使用，这就可能导致漏报内存中的漏洞。为了解决这个问题，在插桩函数sscanf过程中，如果获取到的转换规约为“%[]”时，需要将其*is_string*数组中的对应元素也设置为1。

3.7 函数返回值被const修饰

在插桩代码的过程中，我们会通过向源代码中插入一些用C语言编写的代码片段来重写源代码。在这个过程中，为了实现对源代码中内存漏洞的检测，会在返回值不为空的函数以及其包装函数中插入一个变量*PREFIXret_val*，用于表示函数的返回值，其类型即为函数返回值的类型，并在后续的插桩过程中对其进行赋值操作。当函数类型被const修饰时，此时执行插桩过程后插入的*PREFIXret_val*的类型也会被const修饰，对其赋值会使得插桩后的代码编译错误，因为被const修饰的变量的值不可以被修改。因此需要修改变量*PREFIXret_val*的类型，在此通过在插桩包装函数的过程中将返回值变量的const修饰符删除，来实现在函数中对该变量的一次或多次修改。

4 实验与分析

由于CVE列表中都是现实世界中使用的软件，因此使用CVE来作为测试集更具有现实意义。但CVE种类繁多，且涵盖的代码语言范围广泛，因此需要对CVE进行筛选后再进行实验。这里采取的筛选标准主要有两个，由于本文关注的是C程序中的漏洞检测，因此第一个筛选标准是CVE程序必须是使用纯C语言编写的；且因为本文是对内存错误检测技术的研究，因此第二个标准是CVE的漏洞类型是内存安全漏洞。

经过筛选后，此处选择了16个CVE作为实验的测试集，它们分别隶属于fig2dev，ffjpeg，jhead，libsndfile，ytnef，dmg2img，ngiflib，epub2txt以及ezxml。

动态分析工具在对CVE程序进行漏洞检测的过程中，由于存在无法处理的C程序中的特定结构，因此无法完成内存检测过程。但本文提出的对C程序中特定结构进行内存安全动态分析的方法，将模糊测试工具AFL与当前最先进的动态分析工具Movec结合后，对Google的ASan，SCet和Valgrind在可用性、有效性和性能方面进行了评估。本文实验运行的硬件和软件环境为：处理器为Intel® Core™ i5-10210U CPU、八核、CPU主频1.60 GHz，内存为16 GB，操作系统为64位Ubuntu 22.04.1，编译器为gcc 11.3.0，脚本语言使用bash。

4.1 可用性

在ASan，Valgrind和SCet中，除了Valgrind可以直接插桩二进制文件外，另外两个工具都需要提前改写源程序的Makefile。其中，ASan需要在编译时增加编译选项-fsanitize=address来开启内存检测功能；SCet不仅需要增加编译选项-fsoftboundcets，还必须使用Clang编译器来编译程序，除此之外还需要链接库，而且它检测到错误时，给出的错误信息不够详细，只是指出出现了错误，但并未指明错误的类型；Valgrind则可以使用“valgrind --tool=tool_name program_name”命令来对程序进行内存安全检测。而Movec则支持文件夹形式的编译，它能将插桩后的程序直接输出到指定的文件夹，并直接使用原项目已有的Makefile来编译插桩后的程序。

综合上述，相比ASan，Valgrind以及SCet，Movec具有更强的可用性。

4.2 有效性

为了验证Movec对程序中存在的内存错误进行检测的有效性，本文使用改进后的Movec与ASan，SCet以及Valgrind在筛选后的CVE测试集上进行了实验，具体的实验结果如表1所列。

表1 有效性对比实验数据

Table 1 Validity comparison

CVE	改进前的Movec	Movec	SCet	Valgrind	ASan
2019-14275	YES	YES	YES	YES	YES
2019-19746	YES	YES	YES	YES	YES
2019-1010302	ERROR	YES	NO	YES	YES
2020-26208	ERROR	YES	NO	YES	YES
2021-3496	ERROR	YES	YES	NO	YES
2021-28277	ERROR	YES	NO	YES	YES
2021-34122	YES	YES	YES	YES	YES
2021-44956	YES	YES	YES	YES	YES
2021-44957	YES	YES	NO	NO	YES
2021-45385	YES	YES	YES	YES	YES
2021-3548	ERROR	YES	NO	NO	YES
2022-23850	ERROR	YES	NO	NO	YES
2022-30045	ERROR	YES	NO	YES	YES
2021-3246	ERROR	YES	NO	YES	YES
2021-36531	ERROR	YES	NO	NO	YES
2021-3404	ERROR	YES	YES	YES	YES

表1中，YES表示可以查找出CVE的错误，而NO则相反，ERROR则表示通过工具执行插桩操作后的代码无法正确编译。从表中可以看出，ASan和Movec可以检测出所有CVE中的错误，SCet和Valgrind则无法找出CVE中的全部错误。

利用改进前的对C程序进行内存安全动态分析的工具Movec测试CVE程序时，无法检测出所有CVE中的错误。

因此，对包含特定结构的C程序进行内存安全动态分析的方法不只比其他动态分析工具有效性强，也比改进前的方法具有更有效的漏洞检测能力。

4.3 性能

为了验证结合模糊测试的对包含特定结构的C程序进行内存安全动态分析的工具Movec的运行性能，这里使用ASan，SCet以及Valgrind与Movec进行内存错误检测时间的对比。考虑到存在误差，选用了5次实验的平均值作为结果，实验结果如表2所列，其中time表示时间，单位为s。由于本文中提到的4个工具都是通过插桩源代码来实现其内存安全漏洞检测功能，而编译插桩后的程序得到的可执行文件运行所需时间与编译源程序生成的可执行文件的运行时间不同，因此设置T.R.表示插桩后的程序与原程序的运行时间的比例，可以更加直观地观察到各个工具的运行性能，AVG T.R.表示每个工具对测试集中的CVE的平均拖慢时间比例。从表中可以看出，Movec的平均拖慢倍数是1.87；SCet，Valgrind以及ASan的平均拖慢倍数分别为2.86，43.15，5.20。其中SCet无法检测的CVE数量过多，因此其平均拖慢倍数的参考意义不大；而Valgrind的平均拖慢倍数远远

大于 Movec,表明 Movec 的运行性能远远优于 Valgrind;ASan 的平均拖慢倍数也比 Movec 的平均拖慢倍数大,表示结合模糊测试的对包含特定结构的 C 程序进行内存安全动态分析的工具 Movec 的性能优于现有的动态分析工具。

表 2 程序运行时间对比实验数据

Table 2 Program runtime comparison

CVE	origin	Movec		SCet		Valgrind		AddressSanituzer	
		time/s	T. R.	time/s	T. R.	time/s	T. R.	time/s	T. R.
2019-14275	0.40	0.44	1.10	0.49	1.22	4.63	11.58	0.49	1.23
2019-19746	0.40	0.40	1.00	0.43	1.25	3.90	9.75	0.49	1.23
2019-1010302	0.40	0.05	0.13	0.40	1.00	3.32	8.10	0.56	1.37
2020-26208	0.05	0.05	1.00	0.05	1.00	3.33	66.60	0.56	11.20
2021-3246	0.40	0.43	1.08	0.40	1.00	3.28	8.20	0.45	1.13
2021-3404	0.05	0.05	1.00	0.40	8.00	2.78	55.60	0.43	8.60
2021-3496	0.05	0.05	1.00	0.40	9.20	3.69	73.80	0.51	10.20
2021-3548	0.40	0.42	1.04	0.40	1.00	12.14	30.35	0.63	1.58
2021-28277	0.05	0.45	9.00	0.40	8.00	3.53	70.60	0.64	1.56
2021-34122	0.40	0.40	1.00	0.40	1.00	2.99	7.48	1.00	2.50
2021-36531	0.05	0.05	1.00	0.05	1.00	2.89	57.80	0.05	1.00
2021-44956	0.15	1.00	0.15	0.41	8.20	15.67	104.47	1.00	6.67
2021-44957	0.05	0.05	1.03	0.05	1.00	2.71	54.20	1.00	20.00
2021-45385	0.43	0.43	1.00	0.40	0.93	28.09	65.33	1.00	2.33
2022-23850	0.42	0.49	1.16	0.42	1.00	3.53	8.40	0.60	1.43
2022-30045	0.05	0.41	8.20	0.05	1.00	2.91	58.20	0.56	11.20
AVG T. R.	—	—	1.87	—	2.86	—	43.15	—	5.20

4.4 与 ASan 的进一步对比

由于在 ASan,Valgrind 以及 SCet 中,ASan 是综合表现最好的动态分析工具,因此在此处将结合模糊测试的改进后的 Movec 与 ASan 进行了更进一步的对比。由于 ASan 在检测到第一个错误会立即停止运行,因此需要首先对 CVE 中的漏洞进行修改后重新插桩,再根据 CVE 中给出的输入文件,通过对插桩后的程序利用 AFL 进行模糊测试操作来生成更多的种子,这些种子会引起经过 Movec 插桩后的程序报错,但不一定会使 ASan 报告出程序中的错误。

下面结合例子对 ASan 在内存安全漏洞检测上的缺陷进行介绍。

4.4.1 libsndfile 中的内存空间错误

在对 CVE-2021-3246 的插桩后代码进行模糊测试的过程中得到了多个种子文件,将它们作为输入运行 Movec 插桩后的程序会存在越界漏洞。但使用相同的种子文件作为 ASan 进行内存检测的输入时,并没有出现上述错误。此时,根据报错信息对其进行验证,该指针指向的地址值为 0x556d6acebc7e,且由于其类型为 unsigned int*,因此其大小为 4。但 Movec 在插桩过程中为其创建的 pmd 的 base 和 bound 的差值为 2,这就导致在访问指针时会超出其指向内存的范围,从而引起上述错误。在源代码中,对指针赋值的变量对应的类型为 unsigned short,其 size 为 2。因此 Movec 在为该指针变量创建 pmd 时没有出现错误,即该报错是程序中真实存在的问题。

而 ASan 无法检测到该错误是因为它无法检测到长距离溢出,而此类溢出可能会跳过插入在内存块周围的 red zone,并落入已分配的内存中。向报错指针变量传递值的变量是结构体中的一个 unsigned short 类型的成员,ASan 在使用 unsigned int 类型的指针对内存进行访问时,访问的位置已经超出正确的地址范围。但由于访问的地址仍然在为结构体分配的内存空间的范围内,这就使得此处的越界访问不会出现异常,因此在对长距离溢出的检测问题上,改进的内存安全动态分析工具 Movec 优于 ASan。

4.4.2 jhead 中的内存泄漏

在对 CVE-2019-1010302 进行插桩后检测的过程中,Movec 报出的错误比 ASAN 报出的错误多出了内存泄漏。

在经过 Movec 插桩后的代码中对该错误进行调试发现,是在对全局变量的 pmd 进行清除时检测到的内存泄漏,通过 gdb 对其 pmd 进行跟踪调试后发现其 pmd 中各参数的值在被初始化后直到程序运行结束都没有发生变化,其 pmd 也没有被删除,即为其分配的内存没有被释放,此时打印其值得到其地址值与发生内存泄漏的内存一致。因此,此处的内存泄漏是程序中真实存在的内存泄漏。

而由于 ASan 在检测过程中不会为全局对象周围建立 redzone,无法检测到 C 程序中全局对象的溢出以及泄漏现象,因此,在对 C 程序中全局对象的访问是否出现内存漏洞的问题上,Movec 是优于 ASan 的。

4.4.3 epub2txt 中的空间内存错误

在利用 Movec 对 CVE-2022-23850 进行插桩后检测的过程中存在越界漏洞。该越界由于存在指针指向地址为 0x7ffffffcdc4,且其大小为 1,但其 pmd 中存储的上界和下界分别是 0x7ffffffcdb0 和 0x7ffffffcdc4,因此在此处会出现空间内存错误。在源程序中,该指针指向一个大小为 20 的数组,并在程序运行过程中会递归地读取数组中的内容。此时,如果向数组传递长度超过 20 的字符串,则递归访问到第 21 个字符时会出现越界访问现象。

而 ASan 是将内存的状态记录在影子内存中,在对内存进行操作时通过影子内存来判断该内存的状态。但由于影子内存只可以防止内存访问溢出到相邻存储区,当溢出的区域超出相邻存储区时,无法对溢出进行检测。因此,在对该情况进行内存安全检测时,由于传递给数组的字符串所在内存区域已经超出为数组所分配内存空间的相邻存储区,ASan 无法检测出该内存溢出,因此在检测数组溢出长度过长的问题上,包含对 C 程序特定结构的内存安全动态工具 Movec 优于 ASan。

在 4.3 节中得到 Movec 和 ASan 在性能上大致相同,但

根据本节中的内容可以得到结合模糊测试的改进的 Movec 在内存漏洞检测能力上优于 ASan，且 Movec 可以检测错误的类型更广泛。

上述实验表明，对包含特定语法结构的 C 程序进行内存安全动态分析的工具 Movec 结合 AFL，可以更有效且高效地对 C 程序中存在的内存安全漏洞进行检测。

结束语 本文主要介绍了一种对包含特定语法结构的 C 程序进行内存安全动态分析的方法。通过对大量的 CVE 程序进行插桩操作后得到的无法处理的 C 程序中的语法结构，在已有方法的基础上增加了对这些结构的处理，从而能够检测包含特定结构的 C 程序中存在的内存安全漏洞，并将模糊测试技术与动态分析技术相结合，提高了内存漏洞的检测效率。

本文还采用理论证明和实验证明的方法验证了所提方法的有效性和正确性。但目前的工作仍有不足，后续可以进行的工作包括将对包含特定结构的 C 程序进行内存安全检测的方法与静态分析技术结合，来减少不必要的代码插桩；也可使用多线程并行技术来提升它的插桩效率。

参 考 文 献

[1] RITCHIE D M. The development of the C language[J]. ACM Sigplan Notices, 1993, 28(3): 201-208.

[2] LI Y, TAN W, LV Z, et al. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication [C]// Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. 2022: 1901-1915.

[3] GAO F, WANG Y, CHEN T, et al. Static Checking of Array Index Out-of-Bounds Defects in C Programs Based on Taint Analysis[J]. Journal of Software, 2021, 11(2): 121-147.

[4] XU S, HUANG W, LIE D. In-Fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection[C]// Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021: 224-240.

[5] BABATI B, PATAKI N. Comprehensive performance analysis of C++ smart pointers[J]. Pollack Periodica, 2017, 12(3): 157-166.

[6] CHEN Z, WANG C, YAN J, et al. Runtime detection of memory errors with smart status[C]// Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2021: 296-308.

[7] ZHU X, WEN S, CAMTEPE S, et al. Fuzzing: a survey for roadmap[J]. ACM Computing Surveys (CSUR), 2022, 54(11s): 1-36.

[8] LIANG H, PEI X, JIA X, et al. Fuzzing: State of the art[J]. IEEE Transactions on Reliability, 2018, 67(3): 1199-1218.

[9] CHEN C, CUI B, MA J, et al. A systematic review of fuzzing techniques[J]. Computers & Security, 2018, 75: 118-137.

[10] KLEES G, RUEF A, COOPER B, et al. Evaluating fuzz testing [C]// Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018: 2123-2138.

[11] ZALEWSKI M. Technical "whitepaper" for afl-fuzz[J/OL]. URl: http://lcamtuf. coredump. cx/afl/technical details. txt, 2014.

[12] WANG Y, CUI B. The Study and Realization of a Binary-Based Address Sanitizer Based on Code Injection[C]// International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing. Cham: Springer, 2020: 125-134.

[13] NAGARAKATTE S, ZHAO J, MARTIN M M K, et al. SoftBound: Highly compatible and complete spatial memory safety for C[C]// Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2009: 245-258.

[14] NAGARAKATTE S, ZHAO J, MARTIN M M K, et al. CETS: compiler enforced temporal safety for C[C]// Proceedings of the 2010 International Symposium on Memory Management. 2010: 31-40.

[15] ROBSON D, STRAZDINS P. Parallelisation of the valgrind dynamic binary instrumentation framework[C]// 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications. IEEE, 2008: 113-121.

[16] CHEN Z, WU J, ZHANG Q, et al. A dynamic analysis tool for memory safety based on smart status and source-level instrumentation[C]// Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. 2022: 6-10.

[17] CHEN Z, YAN J, KAN S, et al. Detecting memory errors at runtime with source-level instrumentation[C]// Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2019: 341-351.

[18] CHEN Z, YAN J, LI W, et al. Poster: Runtime Verification of Memory Safety via Source Transformation[C]// 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion). IEEE, 2018: 264-265.

[19] KRONSER A. Common vulnerabilities and exposures: Analyzing the development of computer security threats[D]. Helsinki, Finland: University of Helsinki, 2020.

[20] PHAM V, DANG T. Cvexplorer: Multidimensional visualization for common vulnerabilities and exposures[C]// 2018 IEEE International Conference on Big Data (Big Data). IEEE, 2018: 1296-1301.

MA Yingzi, born in 1996, postgraduate. Her main research interests include verification of software and model checking.

CHEN Zhe, born in 1981, associate professor, is a member of CCF (No. 22234M). His main research interests include verification of software, software engineering and network security.

(责任编辑：喻藜)