计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.241100060

一种结合静态分析的轻量化内存安全运行时检测方法

毛瑞琪[1]　陈　哲[1,2]
1 南京航空航天大学计算机科学与技术学院　南京 211106
2 软件新技术与产业化协同创新中心　南京 211106
(maoruiqi714@qq.com)

摘　要　缓冲区溢出等内存安全问题长久困扰C语言开发者。运行时检测是解决C语言内存安全问题的可靠方法,但也引入了较高的运行时开销。现存的内存安全运行时检测开销削减方法或不兼容已有代码、依赖人工标注,或在削减开销的同时引入漏报和误报,或无法保证非法访存和检测报错的时序性。对此,提出了一种结合静态分析的针对栈上内存区域的轻量化运行时检测方法,将部分运行时元数据查询替换为编译时元数据查询,将大部分高开销的检测函数调用替换为轻量化的内联布尔条件判断,并使用跨过程的按需别名分析将方法扩展到跨过程分析、全程序检测。基于C语言抽象语法树进行静态分析和检测代码插桩,实现了原型工具LISA(Lightweight Inline Safety Assertion)。实验结果表明,LISA降低运行时检测的时间开销平均达36%,仅引入约0.5%额外的空间开销。此外,LISA还解决了现存方法不兼容已有代码、运行时检测有效性低、无法实时保证内存安全的问题。

关键词:内存安全;运行时检测;静态分析;源代码插桩;别名分析
中图分类号　TP309

Lightweight Memory Safety Runtime Detection Method Combined with Static Analysis

MAO Ruiqi[1] and CHEN Zhe[1,2]
1 College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China
2 Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 211106, China

Abstract　Memory safety issues, such as buffer overflow, have long troubled C language developers. Runtime detection is a reliable solution to C language memory safety problems, but it introduces significant runtime overhead. Existing methods to reduce runtime overhead for memory safety detection may be incompatible with existing code, depend on manual annotations, introduce false negatives and positives, or fail to ensure timing consistency between illegal memory access and error reporting. This paper proposes a lightweight runtime detection method for stack memory regions, which combines static analysis to replace certain runtime metadata lookups with compile-time metadata checks, and replaces most high-overhead detection function calls with inline Boolean condition checks. The method also uses on-demand interprocedural alias analysis to extend detection to interprocedural and whole-program analysis. A prototype tool, LISA (Lightweight Inline Safety Assertion), was implemented with static analysis and detection code instrumentation based on the C language abstract syntax tree. Experiments show that LISA reduces runtime detection overhead by an average of 36%, with only about 0.5% additional space overhead. Furthermore, LISA addresses compatibility with existing code, enhances runtime detection effectiveness, and ensures real-time memory safety, overcoming limitations of previous methods.

Keywords　Memory safety, Runtime verification, Static analysis, Source level instrumentation, Alias analysis

1　引言

C语言以其对底层内存的直接高效控制而广泛运用于各类嵌入式系统与性能敏感软件。然而,其对底层内存的控制容易引入内存安全问题。其中,由数组越界引发的缓冲区溢出是内存安全的高频错误之一,且往往引发严重后果[1]。

内存安全运行时检测技术是解决C程序内存安全问题的主要方法之一[2-5]。现存的内存安全运行时检测技术将额外的代码插入到原始程序中,以记录每个内存区域的元数据,并在访问内存前根据元数据进行安全性检查。密集的运行时检测逻辑调用导致了高时间开销。当前各类内存安全运行时检测工具都会拖慢程序运行速度,最高可达数十倍[5],这严重削弱了此类工具检查内存安全问题的效率和实用性。结合运行时检测技术结合适当的静态分析和代码变换,可以有效降低运行时开销。

自动静态分析或手动标注和运行时检测技术结合的方法[1,6-10]能在编译时提取有效语义信息,并在运行时削减内存检测开销。CCured[6]和CheckedC[7]有法需要改变已有的代码,但不能实现自动化分析,需要手动标注代码,效率较低;ASAP[8]进行适当工作负载的分析,计算每个检查的时间成本,实现在给定的时间限制下进行尽可能多的检查,然而此方法会漏报内存错误;SanRaZor[9]使用静态分析的方法提取互

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)
This work was supported by the National Natural Science Foundation of China(62172277) and Joint Research Funds of National Natural Science Foundation of China and Civil Aviation Administration of China(U1533130).
通信作者:陈哲(zhechen@nuaa.edu.cn)

相依赖的检查,减少被插桩的程序的检查次数,然而此方法可能会移除危险访存检查,漏报内存错误;Catamanran[10]将检查线程和程序的主线程分离,使用静态分析调整检测线程工作负载,以最大化并行效率,然而此方法不能保证内存报错发生在内存访问之前,无法实时保证内存安全。

现有的C语言内存安全运行时检测工具在检查内存错误时,会调用工具的内存检查函数,内存检查函数会传递运行时刻内存信息,查询内存元数据进行检测,这个过程频繁发生且开销高。而安全的内存访问占程序访存行为的绝大部分,危险的内存访问仅占极少数。本文基于此特点,提出一种轻量化内联的运行时内存安全检测算法,主要贡献如下:1)提出一种静态分析方法,在编译时构造内存区域的元数据表,采用跨过程别名分析,得到编译时元数据别名表,并通过插入影子栈在运行时传播指针元数据;2)提出一种插桩方法,根据编译时元数据表及其别名表,插入轻量化的内联布尔表达式作为调用内存错误检查函数的前置条件,避免对安全访存进行高开销的内存安全检测;3)在现存检错能力最强的内存安全运行时检测工具 Movec[2,11-14]功能基础上,扩展并实现原型工具 LISA。在 SPEC、MiBench[15]和常用算法测试集上的实验结果表明,LISA 能在保持运行时检测工具有效性前提下,显著削减运行时的时间开销,且几乎不引入额外内存开销。

2 相关技术

2.1 内存安全运行时检测技术

现存的C语言内存安全运行时检测工具常常在二进制代码级别[5]、编译器中间代码级别[3-4]或高级语言源代码级别[12-13]插入用于内存安全检测的代码片段。由于工具的实现原理不同,C程序的内存安全运行时检测的时间开销通常为数倍到数十倍,内存开销可达数倍。此类检测工具通常在运行时维护内存区域元数据。任何指向内存区域的指针都有一个对应的内存区域元数据,元数据可以表示成三元组:

$$\langle base, bound, status \rangle \tag{1}$$

其中,*base* 为内存基址;*bound* 为内存上界,用来检测内存区域的空间安全性,违反空间安全性会造成缓冲区溢出等错误;status 为内存状态,用来表示内存访问是否符合时间安全性,违反时间安全性可能会造成释放后使用、多次释放等错误。内存区域元数据通常会存放在查找表中,其查找键为指针的地址,值为元数据三元组。

基于这个特征,本文运用静态分析构建编译时元数据查找表,插桩轻量化内联布尔表达式。在运行时,内联布尔表达式将作为调用检测函数的前置条件,以减少检测函数调用和运行期元数据查询次数。

2.2 静态分析技术

静态分析技术包括模型检测、数据流分析、抽象释义等。针对C语言的成熟静态分析框架有 FaceBook Infer[16],Clang Static Analyzer[17],SVF Saber[18]等。FaceBook Infer 和 Clang Static Analyzer 无法进行跨编译单元分析;SVF Saber 虽然可以进行全程序分析,但受限于程序复杂度,分析精度受到极大影响。

别名分析是一项经典的静态分析任务,可分为流敏感和流不敏感两类。流敏感分析关注程序中的控制流,即程序的不同执行路径上、不同时间点的别名状态,但时间和空间复杂度很高,不适用于大型程序;流不敏感分析忽略程序执行顺序,只关注变量的定义,能快速给出大致别名关系,适合大型程序分析。按需别名分析是一种优化的别名分析方法,只对特定位置和特定目标做别名关系分析,避免了计算所有变量的别名信息,能够显著提升分析速度,但往往只能聚焦在局部,对涉及多个函数的跨过程情况效果有限。

本文基于抽象语法树实现了流不敏感按需别名分析,并扩展算法到跨过程,将其与内存安全运行时检测结合,实现了高效的全程序分析插桩。

3 LISA 整体框架与优化原理

3.1 LISA 整体框架

LISA 首先使用编译器前端将C文件编译为其对应的抽象语法树;LISA 分析模块基于抽象语法树进行元数据收集、过程内别名分析、过程间别名分析,产出编译时元数据表和元数据别名表;LISA 插桩模块根据分析结果进行影子栈插桩和解引用插桩;Movec 进行剩余的内存安全检测代码插桩并生成插桩后的C文件;C文件编译链接后生成可执行文件。其整体框架如图1所示。

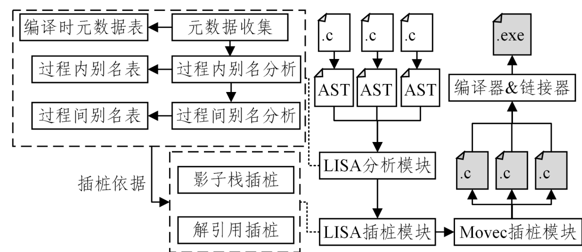

图1 LISA 整体框架

Fig. 1 LISA general framework

LISA 分析模块分为3个阶段:1)元数据收集阶段遍历抽象语法树,识别栈上定义的定长和变长数组变量,并提取其元数据信息生成编译时元数据表;2)过程内别名分析阶段以收集的栈上数组变量作为分析目标变量,提取赋值表达式左右操作数核心指针的定义,分析每一个栈上数组变量可能的别名,生成过程内元数据别名表;3)过程间别名分析阶段以函数参数作为按需别名分析的目标变量,对目标变量进行过程内别名分析产生增量别名信息,对增量别名信息重复分析步骤,LISA 使用一种工作列表算法迭代求解直至抵达算法不动点,生成跨过程的别名表。

LISA 插桩模块根据元数据表和别名表进行影子栈插桩和解引用插桩。影子栈插桩的目标是解决栈上数组及其别名的跨过程传播问题,其在函数调用处插桩元数据入栈代码,在函数体第一条代码前插桩元数据出栈代码。解引用插桩部分的目标是根据元数据构建布尔表达式,并将其插入到解引用代码的合适位置。

Movec 插桩模块继承了 Movec 已有的工具功能,在抽象语法树上进行剩余内存安全运行时检测算法的插桩流程,其任务是保证 LISA 布尔约束为假或优化未覆盖时的兜底安全检测。

经以上流程后,LISA 工具会生成插桩后的C程序源代码,使用任意C语言编译器和链接器即可编译链接,生成目标文件或可执行文件。

3.2 LISA 优化原理

图2展示了现存的内存安全运行时检测方法和 LISA 的检测流程,其中实线代表运行时流程,虚线代表编译时流程。可见现存方法检测函数开销较大的原因在于:1)检测函数非内联,产生参数传递的开销;2)检测函数在运行时查询元数据表,产生运行时查询开销。以 Movec 为例,分析其检测函数的汇编代码发现,无需查询元数据表的简单情况需要执行76条 x86 汇编指令;在需要查询元数据表的情况下达到100条以上。但并非所有检测都需要依赖运行时元数据信息,可以

根据编译时元数据表及其别名表中的信息进行轻量化内联检测。对于绝大部分安全访存，轻量化检测即可保证内存安全；对于危险访存，轻量化检测失效，会落入检测函数逻辑，仅引入极小的额外开销。特别地，栈帧的单向增长与解退天然保证了栈上变量内存访问的时间安全性。对于栈上变量，轻量化检测的布尔表达式可以表示为：

$$0 \leqslant i \land i < len \tag{2}$$

其中，i 为访存下标，len 为内存区域长度。在不同的代码语义下，该布尔表达式的计算仅需 5～10 条汇编指令，显著减少了运行时开销。

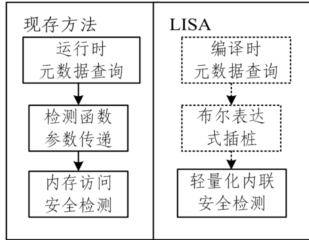

图 2　现存方法与 LISA 的检测流程

Fig. 2　Detection process of existing methods and LISA

本文使用 Movec 插桩 SPEC 部分测试集统计了危险访存和安全访存的数量，如表 1 所列。对于真实世界中的 C 程序，安全内存访问占总内存访问数的绝大部分，危险内存访问占比极小。因此，对于绝大部分安全的内存访问，只需检查布尔表达式是否为真即可；对于极少数危险的内存访问，其布尔表达式为假，则调用检测函数以确保检测工具的有效性。

表 1　危险访存比例数据

Table 1　Hazardous memory access ratios

测试集	内存检查次数	危险访存次数	危险访存占比/%
lbm	5 007 119 212	0	0.0000
xz	2 858 024 214	33	0.0001
nab	925 549 123	1 015	0.0001
x264	128 898 076 911	3 712 628	0.0029
总和	137 688 769 460	3 713 676	0.0027

4　LISA 算法设计与实现

4.1　栈上内存区域元数据定义

栈上内存区域元数据是一个二元组⟨Len，$Type$⟩。其中 Len 为数组变量的元素长度，$Type$ 是元素类型。LISA 基于该元数据实现内存安全保证，包括以下两类。

1）空间安全保证。在变量定义、调用函数实参传播、内存访问 3 类代码位置上，根据指针类型和元数据中的元素类型等比例缩放元数据中的元素长度。例如，在对图 3 代码的函数调用和解引用进行插桩时，缩放元素长度。

```
void foo(double * arr);
// …
int arr[N];
PRFaddr_stack_push(arr);
PRFlen_stack_push(N * sizeof(int) / sizeof(double));
foo(arr);
double * alias=arr;
for(int i=0; i < N; ++i)
  alias[0<=i&&i<N * sizeof(int)/sizeof(double)? i:PRFcheck(arr,arr+
  i,…)]=i;
```

图 3　空间安全保证示例代码

Fig. 3　Sample code for spatial safety assurance

2）时间安全保证。对于没有别名的栈上变量，由于栈帧的单向增长和解退，栈上变量天然具有时间安全性；由于此前使用过近似的别名分析，对于通过别名分析得出的错误结果，影子栈可能会压入错误的元数据信息，最终导致工具漏报内存错误。LISA 采用保守化的插桩策略，不插桩可能具有多个别名的指针的解引用。例如，图 4 中的代码片段，指针 alias 可能为多个数组的别名，因此在过程内 LISA 不进行插桩；在过程间，影子栈入栈时压入地址值 NULL，长度值 0。

```
void foo(int * arr);
// …
int arr[N]; int arr2[M]; int * alias = arr;alias = arr2;
PRFaddr_stack_push(NULL); PRFlen_stack_push(0);
foo(alias);
for(int i = 0; i < M; ++i)
alias[PRFcheck(alias,alias+1,×)] = i;
```

图 4　时间安全保证示例代码

Fig. 4　Sample code for temporal safety assurance

4.2　栈上内存区域元数据收集

LISA 通过编译器前端将 C 文件编译为抽象语法树，然后递归地访问抽象语法树节点，收集栈上变量的元数据信息，LISA 判断数组变量定义是否为变长或定长数组，若是，则获取数组长度及其数组元素的类型，将其加入元数据表 M_{meta}，输出元数据表 $M_{meta}=\{VD \mapsto \langle Len, Type\rangle\}$，其中 VD 为变量定义节点，⟨Len，Type⟩为内存区域长度、类型的二元组（其中内存区域长度可能是编译时常量，如 100；也有可能是运行时才能决定其具体值的变量，如 int 类型变量 N）。

例如，图 5 中元数据表含有两个元素 arr1 和 arr2，其中 arr1 的长度在运行时确定，其元数据为⟨N，int⟩；arr2 的长度在编译期就能确定，其元数据为⟨100，int⟩。图 5 中的例子会在后文多次出现，用于阐述算法过程。

```
int* bar(int* arr, int n) {
  void* PRFaddr_bar_0 = PRFaddr_stack_pop();
  unsigned long PRFlen_bar_0 = PRFlen_stack_pop();
  //...
}
void foo(int* arr, int n) {
  void* PRFaddr_foo_0 = PRFaddr_stack_pop();
  unsigned long PRFlen_foo_0 = PRFlen_stack_pop();
  //...
  PRFaddr_stack_push(PRFaddr_foo_0);
  PRFlen_stack_push(PRFlen_foo_0);
  bar(arr, n); //case (3)
}
int main() {
  int arr1[N], arr2[100];
  int* alias1 = arr1;
  int* alias2 = arr2;
  //...
  alias1 = arr2;
  PRFaddr_stack_push(NULL);
  PRFlen_stack_push(0);
  foo(alias1, N); //case (4)
  PRFaddr_stack_push(arr1);
  PRFlen_stack_push(N);
  foo(alias2, N); //case (2)
  PRFaddr_stack_push(arr2);
  PRFlen_stack_push(100);
  foo(arr2, 100); //case (1)
  //...
}

Mmeta
main::arr1 -> <N, int>
main::arr2 -> <100, int>

Malias
main::arr1 -> {main::arr1, main::alias1,
               main::alias2}
main::arr2 -> {main::arr2, main::alias2}

Mref
main::alias1 -> {main::arr1, main::arr2}
main::alias2 -> {main::arr1}

Midx
{foo -> {0}, bar -> {0}}
```

图 5　元数据相关算法示例代码

Fig. 5　Sample code for metadata algorithms

4.3　基于抽象语法树的元数据过程内别名分析

元数据过程内别名分析在抽象语法树上实现了一种按需

的流不敏感别名分析，计算元数据表中每一个变量的别名集合。分析分为 3 个阶段：1)赋值运算符收集，遍历抽象语法树收集所有的赋值运算符；2)提取核心指针变量，提取赋值运算符左右操作数中的核心指针变量；3)按需流不敏感别名分析，以核心指针算法为辅助，以元数据表中每个指针作为目标变量进行别名分析。

4.3.1 赋值运算符收集

赋值运算符收集算法遍历函数的子抽象语法树，将赋值相关的抽象语法树节点收集到统一的集合中，其中赋值相关的抽象语法树节点包括带初始赋值操作的变量定义节点和二元赋值运算符节点。算法生成用于别名分析阶段的赋值运算符集合$S_{AO}=\{AO_1, AO_2, \cdots, AO_n\}$。在图 5 的例子中，$S_{AO}$中含有 alias1＝arr1，alias2＝arr2，alias1＝arr2 这 3 句代码的抽象语法树节点。

4.3.2 核心指针变量提取

核心指针变量提取的目标是将复杂的复合表达式递归地简化为最小指针表达式，即字符串、数组和指针变量标识符，以便后续别名分析。LISA 关注的指针表达式的语法如下：

$$\begin{aligned}&E := ptr \mid uop\ E \mid E\ bop\ E \mid E.E\\&ptr := arr \mid str \mid id\\&uop := (type) \mid \& \mid ++ \mid -- \mid \cdots \# \#\\&bop := + \mid - \mid = \mid [\,] \mid \cdots\end{aligned} \tag{3}$$

其中，*arr* 为数组变量，*str* 为字符串，*id* 为指针标识符，*uop* 为单目运算符，*bop* 为双目运算符，*E*. *E* 为结构体访问运算符。核心指针变量提取算法的语义操作如表 2 所列。

表 2 核心指针算法语义及其对应操作

Table 2 KPE Algorithm and its semantic operation

语义操作形式	对应操作
KPE(E)	如果 E 为指针类型，则返回 E；否则返回空
KPE(uop E)	如果 uop 为类型保持运算符，则递归调用 KPE(E)；否则返回空
KPE(E_1 bop E_2)	如果 bop 为类型保持运算符，则分递归调用 KPE(E_1)和 KPE(E_2)，返回其中非空结果；否则返回空
KPE(&E)	如果 E＝$*E_1$或者 E＝E_1[i]，则递归调用 KPE(E_1)；否则返回 &E
KPE((type)E)	如果 E 为指针类型，则递归调用 KPE(E)；否则返回 E
KPE(E_1. E_2)	$M_{me}[E_2]=E_1$，递归调用 KPE(E_2)

表中指针类型包括了指针和数组；类型保持运算符在运时会保持操作符的类型，如＋＋，－－，＋，－等；M_{me}为父子节点映射关系表，用于处理抽象语法树上成员访问节点的父子关系。

例如，对于复杂表达式 &a. b. c. arr[0]＋1＊0，其算法过程为：1)处理二目运算符“＋”，“＋”为类型保持运算符，递归调用 KPE(&a. b. c. arr[0])和 KPE(1＊0)，容易判断 KPE(1＊0)结果为空，则结果在另一个表达式中；2)处理 KPE(&a. b. c. arr[0])，得到 KPE(a. b. c. arr)，其中 a. b 执行操作M_{me}[b]＝a，b. c 执行M_{me}[c]＝b，c. arr 执行M_{me}[arr]＝c；3)处理 KPE(&arr[0])，由于满足 E＝E1[i]形式，因此返回 arr，此时算法返回的该复杂表达式的核心指针为 arr，且获得父子节点关系映射表$M_{me}=\{\langle arr,c\rangle, \langle c,b\rangle, \langle b,a\rangle\}$。在接下来的别名分析阶段，元数据表和别名表的键为指针的 VarDecl 节点，而对于结构体，其引用的 VarDecl 节点为最外层的结构体，需要根据节点关系表将最外层结构体转换为指针变量。例如，a. b. c. arr 的抽象语法树结构为：

```
-MemberExpr 'int *' .arr
 `-MemberExpr 'C':'struct C' .c
  `-MemberExpr 'B':'struct B' .b
   `-DeclRefExpr Var 'a' 'A':'struct A'
```

在后续的分析阶段，根据核心指针 arr 查询其父结构体分别为 c，b，a，获得最外层结构体 a 的 VarDecl 节点，最终取得其元数据和别名信息，后文中的算法描述将省略这个转换过程，以保证算法描述简洁可读。

4.3.3 过程内别名分析

过程内别名分析算法根据赋值运算符左右操作数的核心指针，以 4.2 节中输出的元数据表M_{meta}的键作为按需别名分析的目标变量，对每一个函数体的抽象语法树的子树进行过程内别名分析，输出对应函数内变量的别名表。

算法 1 过程内别名分析算法

输入：函数体对应子抽象语法树AST_{FD}，内存区域元数据表M_{meta}，赋值运算符集合S_{AO}

输出：过程内别名表$M_{alias}=\{VD_1 \mapsto \{VD_1, \cdots\}, \cdots, VD_i \mapsto \{VD_i, \cdots\}\}$，指针指向表$M_{ref}=\{VD_2 \mapsto \{VD_1\}, \cdots, VD_j \mapsto \{VD_i, \cdots\}\}$

```
1. M_alias={}
2. for VD in M_meta.keys//遍历元数据表的键
3.    if(该变量在子抽象语法树AST_FD下)
4.       M_alias[VD]={VD}//别名初始化
5.    end if
6. end for
7. do
8.    reachedFixPoint=true
9.    for AO in S_AO//遍历每一个二元操作符
      //获取左右操作数核心指针的变量定义
10.   lvd=GetVarDecl(KPE(AO.lhs))
11.   rvd=GetVarDecl(KPE(AO.rhs))
12.     for ⟨VD,S_vd-alias⟩ in M_alias
13.       if(rvd in S_vd-alias)//右操作符在集合内
          //加入 VD 的别名集合
14.       S_vd-alias=S_vd-alias ∪ {lvd}
          //构造左操作符元数据辅助查找表
15.       M_ref[lvd]=M_ref[lvd] ∪ {VD}
16.       reachedFixPoint=false
17.       end if
18.     end for
19.   end for
20. while(!reachedFixPoint)
```

M_{alias}的键在M_{meta}中具有元数据，VD_{alias}为其可能的别名集合；M_{ref}仅记录M_{meta}中不存在的键，用于在插桩阶段使用该表时降低指针变量查找含元数据别名的时间复杂度，从而不必遍历M_{alias}内所有元素，且用于判断一个指针变量是否可能指向多个栈上变量，具有多个指向则不进行解引用插桩。

在图 5 的例子中，过程内别名分析算法会遍历赋值运算符集合S_{AO}中的 alias1＝arr1，alias2＝arr2 和 alias1＝arr2 这 3 个赋值节点，获取其核心指针变量，通过集合并操作，分别将 alias1 加入 arr1 和 arr2 的别名集，将 alias2 加入 arr2 的别名集，并在此过程中维护指针指向表M_{ref}，得到 alias1 可能指向 arr1 和 arr2，alias2 可能指向 arr2。

4.4 基于抽象语法树的元数据过程间别名分析

过程间别名分析根据过程内别名分析的结果，遍历抽象

语法树上函数调用节点中的形参，判断函数参数是否引用到算法 1 中M_{alias}的元素，将此类参数作为按需别名分析目标变量，在对应函数内调用过程内别名分析算法，产生增量别名信息，对增量别名信息重复上述分析步骤，实现过程间分析。如算法 2 所示。

算法 2　过程间别名分析算法

输入：过程内别名表M_{alias}，指针指向表M_{ref}，所有函数的子抽象语法树集合S_{FD}

输出：函数参数序号表$M_{idx}=\{FD \mapsto \{paramIdx_1,\cdots,paramIdx_i\}\}$，过程间别名分析过程中更新后的指针指向表$M_{ref}$

```
1. Midx={}
2. workQ=[]//工作队列元素为〈函数，参数序号〉
3. visited={}//已访问的〈函数，参数序号〉
4. for FD in SFD//初始化工作队列
5.     for CE in FD //遍历 FD 下的所有调用表达式
6.         if(CE 是一个库函数 ‖ CE 没有调用的函数名)
7.             continue
8.         end if
9.         for arg in CE.args//遍历 CE 下所有参数
10.                if(arg 引用的 VD 在Mmeta或Mref内)
11.                    workQ.enque(〈GetFD(CE),arg.idx〉)
12.            end if
13.        end for
14.    end for
15. end for
16. while(! workQ.empty())
17.    〈FD,idx〉=workQ.deque()
18.    if(〈FD,idx〉在 visited 集合中)
19.        continue
20.    visited=visited∪{〈FD,idx〉}
       //对 FD 第 idx 个参数在ASTFD过程内进行别名分析，过程内别名分析结果会加入Mref
21.    IntraAnalysis for 〈FD,idx〉
22.    Midx[FD]=Midx[FD]∪{idx}
23.    for CE in FD//遍历 FD 下的 CE
24.        forarg in CE.args//遍历 CE 下所有参数
25.            if(arg 引用的 VD 在Mmeta、Mref内或为Midx内实参)
26.                workQ.enque(〈GetFD(CE),arg.idx〉)
27.            end if
28.        end for
29.    end for
30. end while
```

在进行过程间别名分析的过程中，会对函数实参调用过程内分析，同时更新其指针指向表M_{ref}。在图 5 的例子中，初始化阶段首先遍历 main 函数下的所有调用表达式，当访问到 foo(alias1，N)时，遍历其所有实参，发现第一个实参中 alias1 在M_{ref}内，则将〈foo，0〉加入工作队列，在M_{idx}内加入〈foo－>{0}〉；在迭代分析阶段，从工作队列中取出〈foo，0〉，以 foo 第一个形参作为目标变量，在 foo 函数内进行过程内别名分析，更新第一个形参的别名表，并遍历 foo 函数下所有调用表达式，当访问到 bar(arr，n)时，遍历其所有实参，发现 arr 为M_{idx}内下标为 0 的参数，则将〈bar，0〉加入工作队列，在M_{idx}内加入〈bar－>{0}〉。重复上述步骤直至抵达不动点。

4.5　元数据跨过程传播

由于同一个函数可能在不同的上下文中被调用，不同上下文中作为函数实参的内存区域的元数据信息不同。图 5 的代码中，foo 函数多次调用的实参不同，因此，LISA 引入影子栈机制，在运行时传递内存区域的长度和地址，以保证算法跨过程可靠性。现存的影子栈技术往往在中间代码上实现，因为静态单赋值形式容易保证影子栈入栈、出栈的顺序性和原子性。而在源代码上需要分析函数调用所处的上下文环境，本文使用影子栈插入顺序分析保证影子栈执行与函数调用的原子性；使用影子栈插入点分析保证多参数和嵌套函数调用的顺序性；针对函数可能通过指针调用情况，使用分析算法排除此类函数插桩。

4.5.1　影子栈插入顺序分析

由于 Movec 基于 C99 标准[19]插桩，而 C99 中并没有规定函数的传参顺序，且编译器也可能优化参数计算顺序。这意味着参数元数据的出入栈顺序可能与二进制代码中的参数计算顺序不一致。为了解决此问题，影子栈传递的参数增加了内存地址，LISA 在检查时首先判断地址是否与当前内存地址一致，如果不一致，则布尔表达式为假，检查控制权交给 Movec，因此不会造成漏报和误报。本文采用了实践中最常见的自右向左、自内向外的函数参数计算顺序。

首先，针对每一个函数定义的子抽象语法树，分析算法递归地访问每一个调用表达式，得到调用表达式的嵌套关系，其数据结构的组织形式是根节点为最外层调用表达式、叶子节点为其内层调用表达式的树。其次，分析算法对每一个最外层表达式进行模拟调用：从右向左遍历其实参，若实参为元数据的过程内、过程间别名，则将其暂存在待插桩元素列表内；若实参为一个调用表达式，则调用模拟调用算法，递归地进行分析。最后，分析算法输出一个函数参数入栈顺序列表，该列表存储着某一个最外层调用表达式下所有实参和嵌套调用表达式的函数参数入参顺序。

例如，假设 foo 函数的所有参数都有可能含有别名实参 foo(arg1，foo(arg21，arg22，arg23)，arg3)。在这个例子中，LISA 首先处理最外最右的参数 arg3；然后针对第二个参数处的嵌套调用，自右向左处理内层函数参数 arg23，arg22，arg21，由于内层函数返回值作为外层函数参数，外层函数第二个参数 arg2 入栈参数将会被保守化插桩为 0 和 NULL；最后处理最外层最左参数 arg1。最终得出 foo 函数影子栈的入栈顺序为 arg3，arg23，arg22，arg21，arg2，arg1。

4.5.2　影子栈插入点分析

在源代码级别，函数调用所处的上下文环境复杂，需要影子栈插入点分析算法保证影子栈执行和函数调用的原子性。首先记录翻译单元声明根节点到调用表达式节点的抽象语法树路径，在路径上寻找距离调用表达式节点最近关键节点，针对不同关键节点，执行不同的插桩策略。例如，对于代码片段 int main(){return foo();}，其抽象语法树根节点到调用表达式节点的路径为 TranslationUnitDecl－>FunctionDecl－>CompoundStmt－>ReturnStmt－>CallExpr，距离调用表达式节点最近的关键节点为 ReturnStmt，按照 ReturnStmt 的策略插桩影子栈。表 3 列出了各种最近关键节点下的影子栈插桩点，SS1 和 SS2 代表影子栈插桩语句，加粗部分为 LISA 插桩的代码。

表 3 影子栈插入点示例

Table 3 Example of shadow stack insertion point

最近关键节点	示例
CompoundStmt	{ **SS1**;**SS2**; foo(); }
ConditionExpr	if(**SS1**,**SS2**,foo()){…} else**SS1**,**SS2**,foo();
LogicOperator	Expr ‖ (**SS1**,**SS2**,foo())&&(**SS1**,**SS2**,foo())
ForStmt	for(**SS1**,**SS2**,int i=foo(); **SS1**,**SS2**,i < foo();**SS1**,**SS2**,i += foo())
ReturnStmt	return(**SS1**,**SS2**,foo());
SwitchStmt	switch(**SS1**,**SS2**,foo())
CaseStmt	case 0: **SS1**,**SS2**; foo();
WhileStmt	while(**SS1**,**SS2**,foo())
ConditionalOperator	**SS1**,**SS2**,foo()?(**SS1**,**SS2**,foo()):(**SS1**,**SS2**,foo())
Others	(**SS1**,**SS2**,foo())

如表 3 所列,影子栈有 3 类插入方式:1)对于没有分支结构的语法树关键节点,直接在函数上一行插桩,以分号分隔;2)对于可能有分支结构的语法树关键节点,在最近关键节点同一行插桩,以逗号分隔;3)对于关键节点存在运算优先级小于逗号运算符的情况,在关键节点同一行最近插桩,以逗号分隔,且使用一对括号包裹插桩代码和函数调用表达式。

4.5.3 函数可能通过指针调用的处理

由于 C 语言可以通过函数指针完成运行时分派,对于这部分可能通过函数指针调用的函数,LISA 将跳过影子栈插桩以确保出入栈操作的对称性。函数通常会被 DeclRefExpr 节点引用,而 CallExpr 节点通常有一个 DeclRefExpr 直接子节点,当引用函数的 DeclRefExpr 的直接父节点不是 CallExpr 节点时,说明函数可能被赋值,进而通过函数指针调用。例如,典型的函数调用表达式的抽象语法树结构如下:

```
-CallExpr 'int'
 `-DeclRefExpr 'int ()' Function 'foo' 'int ()'
```

而函数被赋值为函数指针的结构如下:

```
-BinaryOperator 'int (*)()' '='
|-DeclRefExpr 'int (*)()' Var 'fun_p' 'int (*)()'
`-DeclRefExpr 'int ()' Function 'foo' 'int ()'
```

CallExpr 的直接子节点为 DeclRefExpr,若不符合此结构则,认为该函数可能被赋值给函数指针,且有可能通过函数指针调用。LISA 将记录此类函数用于判断某函数是否可能通过函数指针调用,在调用方不插桩影子栈入栈逻辑,在该函数的函数体内不插桩影子栈的出栈逻辑。该方法是一种对指针调用函数的过近似分析,即可能将非指针调用函数误报为指针调用函数,但不会漏报指针调用函数,以此保证影子出栈和入栈操作的对称性。

4.5.4 元数据查询与影子栈插桩

影子栈入栈插桩算法在最外层的调用表达式侧,根据影子栈插入顺序分析算法生成的函数参数入栈顺序列表查询编译时元数据表 M_{meta}、指针指向表 M_{ref}、函数参数序号表 M_{idx},以确定一个指针是否可能具有编译时刻元数据信息或为元数据别名,具体分 4 种情况:1)待插桩参数引用的 VarDecl 节点在编译时元数据表 M_{meta} 内,直接从 M_{meta} 中获取其元数据信息;2)待插桩参数引用的 VarDecl 节点在指针指向表 M_{ref} 内,其不具有多个引用对象,且别名不为函数参数别名,从 M_{ref} 获取其唯一别名的 VarDecl,根据别名的 VarDecl 获取其元数据信息;3)满足 2)中其余条件,但是其为函数参数别名(M_{idx} 有该函数参数对应的函数名和参数下标),则根据函数名和参数下标信息获取其影子栈出栈变量名作为其元数据,例如 foo(int * arr)的元数据长度变量名为 PRFaddr_foo_0,运行时地址变量名为 PRFlen_foo_0,将这两个变量作为影子栈传递的参数;4)不满足以上 3 种情况,则影子栈入栈地址为 NULL,入栈长度为 0。

插桩算法在函数第一句代码前插桩出栈函数,并定义两个类型分别为 void * 和 unsigned long 的变量 PRFaddr_func_idx 和 PRFlen_func_idx,用于存储影子栈传递来的内存地址和内存区域长度。图 5 展示了影子栈插桩示例代码,静态分析的结果展示在右下角,覆盖了各种情况。对于 case(1),在 M_{meta} 中查找到 arr2 的元数据,将其元数据作为入栈参数插桩;对于 case(2),在 M_{ref} 查找 alias2 别名,发现其唯一别名为 arr1,则在 M_{meta} 查找 arr1 元数据进行入栈插桩;对于 case(3),发现 arr 为 foo 函数下标为 0 处的形参,则查找 M_{ref} 中 foo 函数的可能具有跨过程别名参数下标集合,发现下标为 0 的形参可能具有跨过程别名,根据函数名和下标组合获取到其跨过程元数据分别为 PRFaddr_foo_0 和 PRFlen_foo_0,将其作为影子栈插桩入栈的参数;对于 case(4),在 M_{ref} 中查找 alias1 的别名,发现其可能有两个别名,则进行保守化插桩,将 NULL 和 0 分别入栈。

4.6 布尔表达式构建与插桩

布尔表达式构建与插桩查找在每一个数组解引用处插入 LISA 的布尔约束,需要对复杂数组下标表达式、多维数组访问做特殊处理。

4.6.1 布尔表达式构建

首先,LISA 查询内存区域的元数据信息,具体步骤已在前文提及,此处不再赘述。然后,比对元数据类型信息是否与内存区域元素类型一致,若不一致,则根据 4.1 节中的方法缩放长度。最后,构建布尔表达式,以访问下标 i 为例,有以下两种情况:1)如果数组名在编译时元数据表 M_{meta} 内,其布尔表达式的形式为 0<=i&&i<len,其中 len 为元数据中的长度字段;2)如果数组名是指针 alias 的别名,其布尔表达式形式为 alias==arr&&0<=i&&i<len,其中 arr 为数组变量名,即内存区域基址,aliasarr 别名的地址。

4.6.2 复杂数组下标表达式插桩

复杂数组下标表达式可能有副作用,需要保证其只执行一次。LISA 使用一个临时变量存储表达式运算后的值,然后使用这个临时变量进行布尔逻辑的构建插桩。例如,以下代码插及桩结果为:arr[**PRFindex_SFX=i++**,0<=**PRFindex_SFX && PRFindex_SFX<len**? **PRFindex_SFX**:PRFcheck(…)],其中加粗部分为 LISA 插桩的代码。LISA 使用一个临时变量 PRFindex_SFX 来存储含副作用表达式 i++运算后的结果,其中 SFX 后缀包含一个随机数以保证变量唯一性。

4.6.3 多维数组插桩

对于多维数组,LISA 在数组低维处插桩的布尔表达式会递归地继承并合取高维处插桩的布尔表达式,以确保不会漏报任何维度的内存错误。例如,以下代码及其插桩结果为:

arr[**0 <= i && i < len_i** ? **i**:PRFcheck(…)][**0 <= i && i < len_i && 0 <= j && j < len_j** ? **j**:PRFcheck(…)]。例如，arr 被定义为形如 arr[2][17]的二维数组，当访问 arr[2][16]处的内存时，LISA 应在一维解引用和二维解引用处都报出越界错误，一维处布尔表达式为假，二维继承一维处布尔表达式后同样为假，两个维度的解引用都会落入 PRFcheck 函数进行检查且报出错误，最终不会产生漏报。

5 实验分析

本章选取 MiBench、SPEC 和常用算法 3 个测试集上的 27 个程序作为实验对象。实验环境：CPU 为 Intel i5-13400，物理内存为 64 GB，操作系统为 Ubuntu 22.04。实验将对比 LISA 内存安全运行时检测工具 Movec 的时间开销和空间开销，并评估引入 LISA 优化后对运行时检测准确性的影响，主要研究以下问题：1) LISA 对 Movec 插桩后的程序运行效率提升程度以及引入额外的内存开销的程度，即算法的性能如何？ 2) LISA 对 Movec 插桩后程序安全性是否有降低，即算法的有效性如何？

5.1 性能

在没有开启编译器优化的情况下，分别对原始程序、Movec 插桩后程序、LISA 优化插桩后程序开展实验。实验结果如表 4 所列。error 列中的 TP 表示内存时间错误，例如释放后使用、多次释放等；SP 表示内存空间错误，例如数组越界、缓冲区溢出等；ML 表示内存泄漏。T. R. 表示 Movec 和 LISA 相对于未插桩程序的时间开销倍数，相似的，M. R. 为内存开销倍数。时间开销缩减的比例是 LISA 相对于 Movec 的 T. R. 的减少比例，即($Movec_{T.R.}-LISA_{T.R.}$)/$Movec_{T.R.}$，空间开销缩减比例的定义类似，即($Movec_{M.R.}-LISA_{M.R.}$)/$Movec_{M.R.}$。

LISA 在 27 个测试程序上平均时间开销缩减比例达到 36.45%，额外引入了 0.52%的内存空间开销，相对于大幅度缩减的时间开销，引入的内存空间开销是可接受的。在时间开销方面，typeset 测试程序由于内存错误过多，导致大量 LISA 布尔表达式检查在运行时为假，落入 Movec 的检查机制，引入了额外的时间开销；lbm_r 是一个访存密集型程序，该程序实现了格子波尔兹曼方法，以模拟 3D 不可压缩流体，程序内部使用一个大数组 LBM_Grid 存储格子内的数据，由于其花费大量时间开销用于访问此类数据，因此 LISA 能够很好地使用轻量化检测机制缩减运行时检测开销；rijndael 是一种高级加密标准(AES)算法，其包括逆字节替代、逆行移位、逆列混淆、轮密钥加等过程，需要密集访存，LISA 在削减其时间开销的同时，减少 Movec 检测逻辑调用，同时减少空间开销。常见算法测试集中的程序大多是访存密集性的常用算法，可见 LISA 对此类程序的优化敏感性可以大幅度减少其运行时检测时间开销。

5.2 有效性

表 4 的 error 列展示了 Movec 和 LISA 报出的错误。可见在 LISA 过近似分析和保守化插桩的机制下，LISA 不会减少 Movec 报出的错误数量与类型，也不会报出实际不存在的内存错误，即 LISA 不会引入误报和漏报。由于 LISA 的轻量化一定发生在 Movec 检测逻辑之前，而 Movec 的检测发生在访存之前，故 LISA 可以保证报错在访存之前发生，可以应用于在线系统，实时保证内存安全。LISA 基于 Movec 检测机制扩展实现，编译程序时仅需增加 LISA 的编译选项，无需手动更改已有代码，故 LISA 工具能保证已有代码的兼容性。

表 4 实验结果

Table 4 Experimental result

测试集	Movec			LISA			时间开销缩减比例/%	空间开销缩减比例/%
	error	T. R.	M. R.	error	T. R.	M. R.		
CRC32		7.49	2.15		5.84	2.17	22.00	−0.79
adpcm		2.24	1.90		1.71	2.05	23.72	−7.88
bitcount		4.80	2.17		3.37	2.30	29.79	−6.13
blowfish	SP=16	16.59	2.62	SP=16	15.63	2.64	5.80	−0.78
dijkstra		14.67	2.42		10.27	2.42	30.00	0.00
ghostscript	SP=140586 TP=3770	10.61	4.25	SP=140586 TP=3770	9.25	4.34	12.83	−2.21
jpeg	SP=96	11.65	1.59	SP=96	10.50	1.59	9.87	0.36
lame	SP=1 ML=23	15.01	1.32	SP=1 ML=23	13.28	1.34	11.55	−1.47
patricia	ML=61334	6.83	7.29	ML=61334	6.70	7.30	1.95	−0.07
rijndael	SP=811899	34.39	2.50	SP=811899	26.76	2.33	22.19	6.67
sha		11.67	1.19		5.00	1.19	57.14	0.00
typeset	SP=514846 ML=16	65.17	5.32	SP=514846 ML=16	65.53	5.39	−0.55	−1.33
lbm_r		35.28	1.00		4.04	1.00	88.54	0.09
x264_r	SP=3712628	49.99	1.10	SP=3712628	46.28	1.10	7.42	−0.07
nab_r	ML=1015	16.43	1.74	ML=1015	16.43	1.74	0.00	0.00
xz_r	SP=348 ML=48	29.57	1.00	SP=348 ML=48	26.90	1.00	9.02	0.05
bellmanford		68.00	1.00		7.57	1.00	88.87	0.00
binarysearch		17.29	1.00		7.43	1.00	57.02	0.00
dft		3.01	1.11		1.36	1.06	54.66	5.00
floyd		120.63	1.00		60.51	1.01	49.84	−1.37
fordfulkerson		16.11	1.01		3.57	0.99	77.83	2.70
kmp		29.27	1.00		5.18	1.06	82.30	−5.56
matrixchain		69.08	1.71		7.15	1.74	89.64	−1.39
mergesort		14.13	1.01		10.63	1.01	24.78	0.00
prim		3.50	1.00		2.00	1.00	42.86	0.00
quicksort		146.25	1.00		139.00	1.00	4.96	0.00
shellsort		13.08	1.02		2.58	1.02	80.25	0.00
AVERAGE							**36.45**	**−0.52**

结束语 本文提出了一种针对栈上内存区域的运行时检测优化算法,在编译时维护内存区域的元数据查找表,使用跨过程的流不敏感别名分析传播内存区域元数据;插桩影子栈,在运行时传播内存区域元数据,使用轻量化的布尔表达式计算代替重量级的运行时检测。本文实现了原型工具 LISA,实验结果表明,LISA 能有效降低运行时检测工具的时间开销,且解决了已有方法不兼容已有代码、运行时检测有效性低、无法实时保证内存安全的问题。

然而 LISA 仍存在别名分析不够精确等局限性。未来的工作是提高别名分析的性能和有效性,以减少 LISA 布尔表达式失效的次数,进一步提高运行时检测的效率;将算法扩展到支持任意内存分配方式,进一步提高对不同内存分配方式的优化覆盖率。

参 考 文 献

[1] YE D,SU Y,SUI Y,et al. WPBOUND:Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions[C]//IEEE 25th International Symposium on Software Reliability Engineering. 2014:88-99.

[2] CHEN Z,WANG C,YAN J,et al. Runtime Detection of Memory Errors with Smart Status[C]//30th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2021:296-308.

[3] SANTOSH N,JIANZHOU Z,MILO M,et al. SoftBound:highly compatible and complete spatial memory safety for C[C]//Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. 2009:245-258.

[4] KONSTANTIN S,DEREK B,ALEXANDER P,et al. AddressSanitizer:A Fast Address Sanity Checker[C]//2012 USENIX Annual Technical Conference. 2012:309-318.

[5] NICHOLAS N,JULIAN S. Valgrind:a framework for heavyweight dynamic binary instrumentation[C]//Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation. 2007:89-100.

[6] GEORGE C,JEREMY C,MATTHEW H,et al. CCured:typesafe retrofitting of legacy software[J]. ACM Transactions on Programming Languages and Systems,2005,27(3):477-526.

[7] ARCHIBALD S,ANDREW R,MICHAEL H,et al. Checked C:Making C Safe by Extension[C]//2018 IEEE Cybersecurity Development. 2018:53-60.

[8] JONAS W,VOLODYMYR K,GEORGEC,et al. High systemcode security with low overhead[C]//IEEE Symposium on Security and Privacy. 2015:866-879.

[9] ZHANG J,WANG S,MANUEL R,et al. SANRAZOR:Reducing Redundant Sanitizer Checks in C/C++ Programs[C]//15th USENIX Symposium on Operating Systems Design and Implementation. 2021:479-494.

[10] ZHANG Y,LIU T,SUN Z,et al. Catamaran:Low-overhead memory safety enforcement via parallel acceleration[C]//Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2023:816-828.

[11] CHEN Z,WANG C,KAN S L,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.

[12] 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 44th ACM/IEEE International Conference on Software Engineering. 2022:22-24.

[13] CHEN Z,ZHANG Q,WU J,et al. A Source-Level Instrumentation Framework for the Dynamic Analysis of Memory Safety[J]. IEEE Transactions on Software Engineering,2023,49(4):2107-2127.

[14] CHEN Z,YAN R,MA Y Z,et al. A Smart Status Based Monitoring Algorithm for the Dynamic Analysis of Memory Safety[J]. ACM Transactions on Software Engineering and Methodology,2024,33(4):1-17.

[15] MATTHEW G,JEFFREY S,DAN E,et al. MiBench:A free,commercially representative embedded benchmark suite[C]//Proceedings of the IEEE 4th Annual Workshop on Workload Characterization. 2001:3-14.

[16] Facebook infer[EB/OL]. https://fbinfer.com/.

[17] Clang static analyzer [EB/OL]. https://clang-analyzer.llvm.org/

[18] SUI Y,YE D,XUE J. Static memory leak detection using full-sparse value-flow analysis[C]//Proceedings of the 2012 International Symposium on Software Testing and Analysis. 2012:254-264.

[19] International Organization for Standardization,ISO/IEC 9899:1999:Programming Languages-C[S]. ISO,1999.

MAO Ruiqi, born in 1999, postgraduate. His main research interests include software verification and program analysis.

CHEN Zhe,born in 1981,professor,is a senior member of CCF(No. 222345). His main research interests include software verification, software engineering and network security.