计算机科学
COMPUTER SCIENCE

多线程C程序内存安全性动态分析方法

严瑞, 陈哲

引用本文

严瑞, 陈哲. 多线程C程序内存安全性动态分析方法[J]. 计算机科学, 2024, 51(6A): 230900115-6.

YAN Rui, CHEN Zhe. Dynamic Analysis Method for Memory Safety of Multithreaded C Programs[J]. Computer Science, 2024, 51(6A): 230900115-6.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.230900115

多线程C程序内存安全性动态分析方法

严　瑞[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 210023
(yanruissy@nuaa.edu.cn)

摘　要　随着软件结构越来越复杂以及其要求更高级别的并发量,出现了越来越多的多线程程序,同时C语言程序缺乏检测其内存安全的能力,进而导致C语言实现的程序可能会存在较多的隐藏漏洞,因此对多线程C程序的内存安全检测尤为的重要。较为前沿且可靠的检测内存安全的技术主要为动态分析技术,且现在对于多线程C程序内存安全检测的工具不是特别完善,错误检测不完全,性能不是很高。因此提出了基于指针的动态分析技术,同时结合无锁技术、源代码插桩技术实现了工具Movec来对多线程C程序的内存安全性进行检测,并且选取专业测试集来进行实验,验证了本工具对于多线程C程序检测内存安全是有效的,检测的错误更多且性能较为优秀。

关键词:多线程;内存安全;动态分析;源代码插桩
中图分类号　TP311

Dynamic Analysis Method for Memory Safety of Multithreaded C Programs

YAN Rui[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　As software results become increasingly complex and require higher levels of concurrency,more and more multithreaded programs are emerging. At the same time,C language programs lack the ability to detect memory security,which may lead to more hidden vulnerabilities in C language implemented programs. Therefore, memory security detection for C language multithreaded programs is particularly important. At present,the most cutting-edge and reliable technology for detecting memory security is dynamic analysis technology,and the tools for detecting memory safety in C language multithreaded programs are not particularly perfect. Therefore,this paper proposes a pointer based dynamic analysis technology,and combines lockless technology and source code instrumentation technology to implement the tool Movec to detect the memory security of C language multithreaded programs. And by selecting a professional test set for experiments,it is verified that this tool is effective in detecting memory security in C language multithreaded programs and has excellent performance.

Keywords　Multithreading,Memory safety,Dynamic analysis,Source code instrumentation

1　引言

伴随着技术的进步,软件的结构变得愈发复杂,体量也变得愈发庞大,随之出现的是更多的多线程程序。多线程程序的复杂性和不确定性使得对多线程软件可靠性的要求变得越来越高。尤其对C语言来说,这种可靠性更加重要。C语言和其他语言相比,更加接近低层,开发者可以通过操作指针的方式对内存进行直接使用,同时编译器并没有对内存安全的检测,因而编写C语言代码时可能会产生内存安全的隐患。当程序越来越庞大,这样的隐患愈发被不易发现,而这些隐患可能会导致程序的崩溃。内存安全错误包括指针访问内存越界、缓冲区溢出、内存泄漏、引用释放指针等等。因此,提出一种有效的对C语言多线程程序的内存安全检测方法尤为重要。

动态分析技术指在程序运行过程中对程序运行的信息进行获取,同时进行内存安全方面的分析,监测运行时是否出现内存安全的问题。相对地,静态分析技术指在程序运行之前,对编写的代码进行词法语义分析,找到代码编写中的错误,可以在程序运行之前分析大多数编码问题,以此减少程序运行中的错误。相比静态分析技术,动态分析技术更容易捕获到内存的使用情况,因此采用该方法对内存安全进行检测。

本文提出了一种针对C语言内存安全进行检测的动态分析方法。本文针对不同的内存安全漏洞设计了对应的检测

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);CCF-华为胡杨林基金形式化专项资助
This work was supported by the National Natural Science Foundation of China(62172217),National Natural Science Foundation of China—Civil Aviation Administration of China Joint Research Fund for Civil Aviation(U1533130) and CCF Huawei Populus euphratica Forest Fund formal special support.
通信作者:陈哲(zhechen@nuaa.edu.cn)

方法，同时针对C多线程程序不同的编程方式进行总结归纳分类，采取对应的方法进行源代码的插桩，使其使用更加广泛。采取的技术包括指针元数据、CAS无锁技术、源代码插桩技术。我们对专业测试集SPEC，PARSEC和MPI2007使用本文方法进行实验，将动态分析的结果和预期进行对比，结果表明，本文提出的内存安全动态分析方法对C语言多线程是适用且有效的。本文的主要贡献如下：

1）使用Clang编译器，对C程序预处理后的语法树进行分析、插桩，插桩后程序运行时提示相关内存安全错误。

2）通过对不同的C多线程程序的编程方式，包括OpenMP，Pthread，MPI，采取对应的方法进行源代码插桩，使得方法的应用性更广。

3）选择3种不同C语言多线程编程方式的专业测试集对方法进行的验证，结果证明本文提出的方法在对源程序运行时间略有增加的情况下，可以有效地对内存安全性进行检测。

2 背景和相关工作

2.1 动态分析技术

目前，针对C程序的内存安全检测方法包括静态分析和动态分析。动态分析指在程序运行时对代码进行相关分析，以此根据相关规则跟踪相关代码来检测程序中是否存在安全问题。如今，已经有许多动态分析方法[1-4]被提出用以检测运行时内存错误。它们通常维护元数据跟踪每个内存对象的空间或时间信息[5]，或者每个指针的引用，例如指针可以访问的合法内存的边界。本文使用改进的指针技术来对程序进行运行时动态分析，通过对源代码进行插桩来实现，其支持随多线程C程序的运行时检测。

2.2 智能状态指针技术

智能状态指针算法[6-8]是基于指针技术[9]的动态分析技术，对程序中的每一个指针维护一个指针元数据（Pointer Metadata，pmd）跟踪指针引用的边界，对每一个参数中或者返回值存在指针变量的函数维护一个函数指针元数据（Function Pointer Metadata，fmd），同时对每一个内存对象维护一个状态节点（Status Node）以跟踪内存对象的状态和引用计数。内存对象的状态即内存的类型，包括无效类型、堆栈类型、全局类型、静态类型、函数以及系统函数类型。所有指向同一内存对象的指针在其pmd的字段中共享同一状态节点。状态节点是智能的，因为当其引用计数达到零时，它会自动删除，即没有指针指向相应的对象，也没有pmd使用此节点。如果访问的位置超出合法的范围，则会捕获空间错误。如果状态节点中跟踪的状态无效，则会捕获时间错误。如果指针的使用方式与跟踪的段类型不匹配，则会捕获段错误。当堆对象的跟踪引用计数达到零时，会当场发现内存泄漏。然而其对多线程的程序没有做到兼容，所以对多线程C程序插桩该方法并不是有效的。

2.3 不同的多线程编程

C语言的多线程程序编写方式有很多，我们在Linux系统环境下专注于以下3种：OpenMP，pthread，MPI。

OpenMP使用fork-join的执行模式，即程序运行时最初仅存在一个主线程，执行并行代码时，会出现所需数量的分支线程来并行执行相应的任务作业。当并行块完成之后，需要并行的线程全部执行完毕才会把程序的控制交给主线程。

Pthread是POSIX的线程标准。其定义了一组线程相关的函数和数据类型，包括线程的创建、销毁、同步等操作。根据需要，我们使用其中的操作可以创建多个线程来实现多线程执行，其主要专注于使用定义的接口来实现多线程的运行。

MPI是一种基于消息传递的并行编程。MPI标准定义了一组编程接口，它们同时具有可移植性。MPI程序基于消息传递，每个并行执行的进程会维护其独立的存储空间，各自独立运行。进程间交互全部通过调用通信函数，即定义的相关消息传递接口来完成。其主要专注于使用定义的接口来实现进程之间的消息通信。

针对以上几种多线程编写方式，需要对其进行总结归纳得到一种适用于普遍C语言多线程程序的内存安全动态分析方法。

2.4 CAS技术

CAS（Compare-and-Swap）是一种无锁的、非阻塞的线程并发安全的技术。即对某个数据进行操作时，别的线程无法进入，其用于实现无锁数据结构中的原子更新，以确保无锁数据结构的正确性和性能。使用CAS可以避免在多线程环境下使用传统锁机制带来的巨大性能开销。CAS有3个参数：内存位置值（V）、预期值（A）以及新的需要修改的值（B），其具体步骤如下：

1）比较：比较某一个内存位置的存的值（V）是否和期望（A）的值相同，相等则继续，不相等则返回；

2）CAS将新值（B）写入内存位置；

3）成功返回，提示修改成功。

因为CAS的特性，其可以用于实现无锁数据结构，例如无锁链表、无锁队列等。通过CAS操作保证存储的数据的一致性，避免数据竞争访问。

3 研究方法

3.1 基于智能指针技术的优化

优化的智能指针技术使用源代码级别的插桩技术实现，对指针元数据和函数指针元数据进行一定的优化。指针元数据中储存程序运行过程中每一个指针的相关信息，其中包括如下字段：指针指向内存对象的首地址（base）和末地址（bound）、指针所指向内存对象的状态节点的地址（snda）、指针的地址（ptra）、为解决哈希冲突存在的下一个pmd的地址（pmd存储为哈希表存储）、分割序键（so_key）。pmd结构如下：其中stat_node是状态节点类型，ptr_addr是指针地址类型，分割序键的值与指针变量地址的二进制逆转相关。

函数指针元数据fmd针对所有参数中包含指针或返回值为指针的函数，通过fmd来对函数形参中以及返回值的pmd进行传递，以此将指针元数据的值传入函数内部进行进一步维护，因此fmd包含以下字段：函数参数以及返回值中指针类型变量的指针元数据（pmds）和数量（capacity）、函数地址（func）、分割序键、为解决哈希冲突存在的下一个fmd的地址（fmd存储为哈希表存储）、函数调用所在的线程ID（t_id）。fmd结构如下：其中stat_node是状态节点类型，ptr_addr是指针地址类型，分割序键的值与指针变量地址的二进制逆转相关，t_id是为了解决多个线程进入同一个函数造成的数据竞争问题。由于多线程调度的不确定性以及可能出现的数据竞争问题，指针元数据结构的设计需要确保不同线程插入、修改和删除指针元数据时的正确性。

3.2 CAS结合自旋锁相关算法

因为多线程调度的不确定性以及可能出现的数据竞争，

指针元数据和函数指针元数据的存储结构需要保证在原程序正确运行的情况下不会出现数据的竞争。这里我们采用无锁哈希表的数据结构存储 pmd 和 fmd，该数据结构不再是像传统的哈希表一样在哈系桶中插入项元素，而是将每一个桶元素指向相应的项元素前面。

项元素分为虚节点和实节点，虚节点为桶元素节点，实线节点为项元素节点，以此来形成元数据的链表，同时哈希表中哈希槽索引指向相应的虚节点，以此实现快速定位。

同时，哈希表使用 pmd 中的分割序键（so_key）来进行无锁有序链表的排序。其中，分割序键的值是由 pmd 对应的指针变量地址的值转换得到，具体分为两种情况：当指针元数据是一个桶元素时，先计算指针变量地址的哈希值，然后进行二进制逆转得到分割序键的值，其末尾为 0；当指针元数据是一个项元素时，先计算指针变量地址的哈希值，然后进行二进制逆转并将最低位设置为 1 得到分割序键的值。通过这样的操作区分了桶元素和项元素，同时对哈希表中的节点进行了排序。

为了实现数据分隔，确保多个线程对哈希表中数据进行访问时的正确性，避免对指针元数据进行查找、修改和删除时出现脏读、幻读、重复删除等问题。我们对指针元数据进行添加、修改、删除时应当为原子操作，使用 CAS 结合自旋的方式来保证多线程并发时的安全。

同时，CAS 结合自旋操作指，如果 CAS 操作失败，即内存位置的值和期望的值不相同，说明其他线程对这块内存进行了修改，本线程会跳转到 CAS 操作之前的位置继续执行，直到成功执行，这样便不需要锁整个表或者整个哈希槽了。在本工具中，对于每一个线程都维护 3 个线程私有变量 *prev*，*cur* 和 *next*，通过它们每一个线程可以单独访问链表中的节点。其中 cur 指向当前线程正在访问的元数据节点，即可以作为 CAS 操作参数中的预期值；*prev* 指向链表中该元数据前一个节点的下一节点的地址成员，对其解引用即可以作为 CAS 操作中的内存位置存放的值；next 变量方便在链表中进行数据的插入、删除。同时，定义一个危险数组（hazard array），如果每个线程正在有访问操作，则其私有变量都存在该数组中，该线程访问操作结束后再将其私有变量从危险数组中移除。每个线程在销毁节点时都需要判断需要销毁的节点是否存在于危险数组中，若存在则不能释放，反之则说明该节点没有被其他线程访问，可以直接销毁指定的节点。

根据哈希表的存储结构以及 CAS 结合自旋锁技术，对元数据操作的核心函数包括：查找元数据、插入元数据、更新元数据、删除元数据。

1）查找元数据：根据需要查找的 pmd 的指针地址去哈希表中寻找，首先根据指针地址计算出其对应的哈希槽，定位到链表中桶元素，通过遍历的方法进行寻找，需要查找 pmd 中的指针变量地址值和分割序键是否和正在遍历的项元素相同，相同则返回。线程私有变量 *cur* 指向该指针元数据，变量 *prev* 指向链表中该指针元数据前一个节点的下一节点地址成员，返回 *cur* 指向的 pmd，如果不存在则返回空。

代码如下，其中 PRFpmd 为指针元数据类型，tbl 为存放 pmd 的哈希表结构，ptr_addr 为指针地址类型，tbl 的成员变量 slots 代表哈希槽。

```
1. PRFpmd * find(hashtbl * tbl,ptr_addr ptra){
2.   /* index 为 ptra 的 hash 值 */
3.   if(tbl->slots[index]==NULL)
4.     hashtbl_init_bucket_pmd(tbl,index); // init
5.   /* 到链表中直接从桶元素开始查找对应的 pmd */
6.   /* 即遍历其哈希值相同的值的项元素 */
7.   if(list_find((PRFpmd * *)(tbl->slots+index),ptra,so_regularkey(ptra))){
8.   hazard_ptrs_clear();//删除危险数组中元素
9.   return(PRF*) * _get_cur_ref();//找到了对应的 pmd }
10.  hazard_ptrs_clear();
11.  //没有找到对应的 pmd
12.  return NULL;}
```

2）插入元数据：根据待插入的元数据查找哈希表，判断哈希表中是否存在和待插入 pmd 中指针变量地址值和分割序键相同的 pmd。

如果不存在这样的指针元数据，变量 *cur* 指向的指针元数据的分割序键的值是链表中大于待插入指针元数据 pmd 的分割序键值的最小值，因此将待插入指针元数据 pmd 中的下一节点地址的值设置为 *cur*，然后通过 CAS 操作将变量 *prev* 指向的变量的值设置为待插入指针元数据 pmd 的地址。如果 CAS 操作失败，则重新执行插入操作。

代码如下，其中，PRFpmd 为指针元数据类型，PRFpmd 的 ptra 成员为指针变量地址，参数 *head* 为哈希表的地址，函数 *Find* 用于在哈希表中查找待插入指针元数据，函数 *PRFpmd_free* 用于删除指针元数据。

```
1. /* 全局变量定义：PRFpmd * * prev, * cur; */
2. int insert(PRFpmd * * head,PRFpmd * pmd) {
3.   while(1) {
4.   /* 找到相同 pmd，替换并释放已有 pmd */
5.   if(Find(head,pmd->ptra)) {
     /* 函数 Find 会设置 prev 和 cur */
6.     pmd->next=cur->next;
7.     if(CAS(prev,cur,pmd)) {
8.       PRFpmd_free(cur);cur=pmd;
9.       return 0;} }
10.  /* 未找到相同的指针元数据，插入新的指针元数据 */
11.  pmd->next=cur;
12.  if(CAS(prev,cur,pmd)) return 1;} }
```

3）更新元数据：即上述 2）中插入 pmd 时，如果存在这样的指针元数据，线程私有变量 *cur* 指向该指针元数据，变量 *prev* 指向链表中该指针元数据前一个节点的下一节点地址成员，因此将待插入指针元数据 pmd 中的下一节点地址的值设置为 *cur* 指向的指针元数据中的下一节点地址的值，然后将变量 *prev* 指向的变量的值设置为待插入指针元数据 pmd 的地址，删除 *cur* 指向的指针元数据，最后将变量 *cur* 的值设置为待插入指针元数据 pmd 的地址。

4）删除元数据：根据给定的指针变量地址和分割序键查找哈希表，判断是否存在与它们相等的指针元数据。

如果不存在这样的指针元数据，则不需要删除任何指针元数据。

如果存在这样的指针元数据，变量 *cur* 指向该指针元数据，变量 *prev* 指向链表中该指针元数据前一个节点的下一节点地址成员，因此通过 CAS 操作将该指针元数据的下一节点地址的二进制最低位设为 1，表示将该节点标记为已删除，如果操作失败则重新执行删除操作；如果操作成功则通过 CAS 操作将变量 *prev* 指向的变量的值设置为 *cur* 指向的指针元

数据的下一节点地址，并删除 *cur* 指向的指针元数据；如果变量 *prev* 指向的变量值不等于变量 *cur* 的值，说明别的线程修改了节点，需要重新执行删除操作。

代码如下，其中，PRFpmd 为指针元数据类型，PRFpmd 的 ptra 成员为指针变量地址，PRFpmd 的 so_key 成员为分割序键，参数 *head* 为哈希表的地址，函数 *Find* 用于在哈希表中查找待删除指针元数据，函数 *PRFmark_ptr* 将变量的二进制最低位设置为 1，函数 *PRFpmd_free* 用于删除指针元数据。

```
1. /* 全局变量定义:PRFpmd * * prev, * cur; */
2. int remove ( PRFpmd * * head, PRFptr _ addr ptra, PRFso _
   key_t so_key) {
3.   while(1) {
4.   if(! Find(head,ptra,so_key)) return 0;
5.   if(!CAS(cur->next,cur->next,PRFmark_ptr(cur->next)))
6.     continue;
7.   if(CAS(prev,cur,cur->next)) PRFpmd_free(cur);
8.   else continue;
9.   return 1;} }
```

3.3 OpenMP 特殊语言结构的插桩

因为 OpenMP(下文简称 omp)进行多线程代码编写的特殊性，同时我们只专注于源代码层面的插桩，需要对使用 OpenMP 的多线程程序进行特殊的处理。下面分别对其特殊情况进行分析。

omp 一般性语法如下：

#pragma omp 指令[子句[子句]…]

并行运行代码块

其中指令包括 parallel, for, parallel for, sections, parallel sections, critical, single, barrier, atomic, master, ordered, threadprivate；子句包括 private, firstprivate, lastprivate, reduce, nowait, num_threads, schedule, shared, ordered, copyprivate, copyin, default。下面主要挑选对运行时插桩需要进行特殊处理的几种重要的子句进行分析。

对于 private, firstprivate, lastprivate 子句中指定的每个指针变量，每个线程都会创建该指针变量的独立副本，因此，在该子句中插入该指针变量的指针元数据变量名，使得每个线程各自创建该指针元数据的独立副本，并在 omp 块中插入对该指针元数据的赋值语句，并修改该指针元数据中的指针变量地址成员的值。

1)对于 private 子句中指定的每个指针变量，由于在并行块中各个线程会用到其复制的指针变量(但是不会进行赋初值操作)，因此在 private 子句中插入该指针变量的指针元数据变量名，并在 omp 块中插入对该指针元数据的赋值语句，修改该指针元数据中的指针变量地址成员的值。

对图 1 中代码进行相关插桩操作之后得到图 2。由于通过上述操作 omp 块为每个线程复制了一份同名的指针元数据变量，但是未执行初始化，因此在 omp 块中插入了对该指针元数据变量的赋值语句。由于每个线程新的指针变量 *ch* 的地址各不相同，因此在 omp 块中将该指针元数据变量的 ptra 成员的值修改为当前线程的变量 *ch* 的地址。

其中，PRFpmd 是指针元数据类型，PRFpmd_ch 是指针 *ch* 的指针元数据变量，*PRFpmd_init_val* 是指针元数据的初始值，函数 *PRFpmd_set_ret* 用于设置指针元数据变量的值，*PRFglobal_sa* 是全局内存对象的状态节点，*PRFinvalid* 表示无效内存状态的值。

```
1. void test() {
2. char * ch="a";
3. #pragma omp parallel private(ch)
4.   {ch="b";}}
```

图 1 private 子句使用代码示例

Fig. 1 Example of codes used by private clause

```
1. void test() {
2.   PRFpmd PRFpmd_ch=PRFpmd_init_val;
3.   char * ch=(char *)PRFpmd_set_ret(&PRFpmd_ch,
4.   PRFglobal_sa,PRFinvalid,"a","a"+2,(void *)("a"));
5.   #pragma omp parallel private(ch,PRFpmd_ch)
6.   { PRFpmd_ch=PRFpmd_init_val;
7.     PRFpmd_ch. ptra=&ch;
8.     ch=(char *)PRFpmd_set_ret(&PRFpmd_ch,
9.     PRFglobal_sa,PRFinvalid,"b","b"+2,(void *)("b"));
10. }}
```

图 2 private 子句插桩后示例

Fig. 2 Example of instrumented private clause

2)对于 firstprivate 子句中指定的每个指针变量，由于在并行块中各个线程会用到其复制的指针变量并且会赋初值为进入并行块之前的同名变量的值，因此在 firstprivate 子句中插入该指针变量的指针元数据变量名，并在 omp 块中插入对该指针元数据的赋值语句，仅需修改该指针元数据中的指针变量地址成员的值。

对图 3 中代码进行上述插桩操作后得到图 4，omp 块为每个线程复制了一份同名的指针元数据变量，并执行了初始化，因此在 omp 块中只需要将该指针元数据变量的 ptra 成员的值修改为当前线程的变量 *ch* 的地址。其中相关变量说明同上述 1)中描述。

```
1. void test() {
2.   char * ch="a";
3.   #pragma omp parallel firstprivate(ch)
4.   {ch="b";}}
```

图 3 firstprivate 子句使用代码示例

Fig. 3 Example of codes used by firstprivate clause

```
1. void test() {
2. PRFpmd PRFpmd_ch_=PRFpmd_init_val;
3.   char * ch=(char *)PRFpmd_set_ret(&PRFpmd_ch,
4.   PRFglobal_sa,PRFinvalid,"a","a"+2,(void *)("a"));
5.   #pragma omp parallel firstprivate(ch,PRFpmd_ch)
6.   {PRFpmd_ch. ptra=&ch;
7.     ch=(char *)PRFpmd_set_ret(&PRFpmd_ch,
8.     PRFglobal_sa,PRFinvalid,"b","b"+2,(void *)("b"));
9.     }}
```

图 4 firstprivate 子句插桩后示例

Fig. 4 Example of instrumented firstprivate clause

3)对于 lastprivate 子句中指定的每个指针变量，在 lastprivate 子句中插入该指针变量的指针元数据变量名，并在 omp 块中插入对该指针元数据的赋值语句，并修改该指针元数据中的指针变量地址成员的值。lastprivate 会将最后的运行完成线程中 lastprivate 子句中的变量赋值给并行块外的同名变量，如果是指针变量，由于我们已经将其同名指针元数据放入 lastprivate 子句当中，因此最终其并行块之外的同名指针元数据也会被赋值。对图 5 中代码进行上述插桩操作后得

到图 6，其中相关变量说明同上述 1)中描述。

```
1. void test() {
2.    char * ch="a";
3.    #pragma omp parallel for lastprivate(ch,x)
4.    for(int i=0;i<10;i++) { ch="b";}}
```

图 5　lastprivate 子句使用代码示例

Fig. 5　Example of codes used by lastprivate clause

```
1. void test() {
2.    PRFpmd PRFpmd_ch=PRFpmd_init_val;
3.    char * ch=(char *)PRFpmd_set_ret(&PRFpmd_ch,
4.    PRFglobal_sa,PRFinvalid,"a","a"+2,(void *)("a"));
6.    #pragma omp parallel for lastprivate(ch,PRFpmd_ch)
7.    for(int i=0;i<10;i++) {
8.    PRFpmd_ch=PRFpmd_init_val;
9.    PRFpmd_ch.ptra=&ch;
10.   ch=(char *)PRFpmd_set_ret(&PRFpmd_ch,
11.   PRFglobal_sa,PRFinvalid,"b","b"+2,(void *)("b"));}}
```

图 6　lastprivate 子句插桩后示例

Fig. 6　Example of instrumented lastprivate clause

3.4　包装函数的生成

包装函数是我们为处理函数参数或者返回值中存在指针变量而对原来函数进行的操作，具体为将所有的参数和返回值中的指针变量及其指针元数据变量作为参数写入新的函数中，同时通过函数指针元数据表来进行存储。运行函数时，会根据该表将这些指针元数据传入函数内部，以此来实现指针元数据的传递和跟踪。

1)包装函数的数据竞争问题

智能状态指针技术的包装函数可能会在多线程环境下造成的数据竞争(当 fmd 结构中没有线程 id 字段时)，下面为包装函数代码：

```
1. int PRFf(PRFpmd * p1_pmd,PRFpmd * p2_pmd,
2.      int * p1,int * p2){
3.   int ret_val;
4.   fmd_tbl_create(f,4);
5.   fmd_pmd(f,0,p1_pmd);fmd_pmd(f,1,p2_pmd);
6   ret_val=f(p1,p2);
7.   fmd_tbl_remove(f);
8. pmd_freenull_ptr(p1_pmd);pmd_freenull_ptr(p2_pmd);
9. return ret_val;}
```

其中，f 为原函数，$PRFf$ 为包装函数，fmd_tbl_create 为根据函数地址来创建函数指针哈希表函数，fmd_pmd 为根据指针变量的次序依次更新 fmd 中 pmd 的具体值，fmd_tbl_remove 为移除哈希表。当多个线程进入该函数时，由于线程的不确定性，可能会出现一个线程修改另一个线程 fmd 中字段的值，一个线程删除哈希表而另一个线程还在使用同一个哈希表。针对该问题，采取的方式是通过增加一定空间的方式来避免包装函数中的加锁操作，从而减少了不必要的时间消耗。我们在 fmd 中会有线程 id 的字段，当创建 fmd 时，不再只是根据函数地址来映射该函数的 fmd，而是通过函数地址加上线程 id 的方式来映射 fmd，这样避免了不同的线程进入同一个 fmd，消除了数据竞争。

2)pthread 中函数的一些包装函数

pthread 通过接口的方式来实现多线程，因此它们的一些包装函数需要进行相关的编写。下面为几个重要的包装函数编写。

下面给出了 pthread 处理线程私有变量的函数：

```
1. / * Create a key value * /
2. int pthread_key_create(pthread_key_t * __key,
3.          void( * __destr_function)(void * ));
4. / * Destroy KEY. * /
5. int pthread_key_delete(pthread_key_t __key);
6. / * Return current value of the thread-specific data slot identified by KEY.
* /
7. void * pthread_getspecific(pthread_key_t __key);
8. / * Store POINTER in the thread-specific data slot identified by KEY. * /
9. int pthread_setspecific(pthread_key_t __key,
10.       const void * __pointer)。
```

上述函数主要用于线程存储，通过 key-value 来存储。每一个线程，私有数据可以公共访问，而对其他线程不可见。线程私有数据采用了一键多值的技术，即一个键对应多个值。不同的线程都通过 key 访问 value，表面上对一个变量进行访问，实际上数据并不相同。其中 *pthread_key_create* 函数用来创建存储的 val，*pthread_setspecific* 函数用来将 value 的值和 key 相关联，* *pthread_getspecific* 函数不通过线程而是通过 key 获得 value，*pthread_key_delete* 函数用来销毁 key。由于 * *pthread_getspecific* 函数返回值不能确定其准确的返回类型，而我们需要对其返回的指针类型进行内存安全的检测，因而添加另外的映射关系{〈key，thread_id〉，〈pmd_value〉}，键为 key 变量和线程 id 的组合，值为 value 的 pmd。

对 *pthread_key_create* 函数进行包装函数的编写时，将{〈key，thread_id〉，null}的映射关系存入哈希表中。使用 pthread_setspecific 赋值时，更新哈希表中{〈key，thread_id〉，〈pmd_value〉}，当调用 * *pthread_getspecific* 时，从哈希表中通过键得到对应的值。调用 *pthread_key_create* 时将对应的键值对删除。

4　实验与结论

针对以上的相关算法，我们对专业测试集 SPEC，PARSEC，MPI2007 中多线程程序进行实验，将动态分析的结果和预期进行对比，验证其有效性和高效性。

4.1　有效性实验

从测试集 SPEC，PARSEC，MPI2007 中选取 12 个测试集进行有效性的验证，本实验平台为 64 位 Ubuntu22.04 系统，处理器 Intel(R) Core(TM) i5-12400，内存为 16GB，编译器为 gcc-11.4.0。

通过实验可以得出结论，Movec 可以对选取的测试集实现 C 语言多线程程序的内存安全检测。同时将结果和其他插桩工具 Asan，Valgrind 进行对比，其他工具可以检测到的错误，Movec 同样可以有效地检测出。而其他工具检测不到的错误，Movec 却能够检测出来，例如子对象越界问题。

4.2　性能试验

选取 Asan，Valgrind 插桩工具来做对比分析，比 Movec Asan[10] 时间长，不过相比 Valgrind[11-12] 来说平均时间减少了。Valgrind 采用了二进制代码插桩，在选取的测试集的实验中，其运行时间比其他工具更长。ASan 更加轻量，其时间消耗比 Movec 少，而其检测的错误可能会有漏报。Movec 由于其数据结构的存储以及一些特殊情况的处理，其运行时间较 ASan 长一些，但在检测错误的数量上 Movec 可以做得更好，在测试集 xz_s，botsalgn，cholesky，ffsb，fft，lu_cb，water_nsquared 中，Movec 可以检测出更多的内存安全错误，且通过

验证 Movec 检测的结果是有效的。工具的运行时间以及检测错误数量的对比如表 1、表 2 所列，其中表 2 中 ML 表示内存泄漏，SP 表示空间内存错误，TP 表示时间内存错误。

表 1 运行时间对比

Table 1 Comparison of run time

测试集	Origin	Movec	Valgrind	Asan
lbm_s	9.12	45.97	227.36	15.77
nab_s	0.6	16.46	3094.05	0.84
xz_s	11.63	487.18	225.33	19.85
botsalgn	0.06	4.89	0.24	0.34
botsspar	0.01	0.74	0.11	0.25
smithwa	0.33	36.92	1.08	2.64
cholesky	0.01	0.01	28329	0.07
ffsb	10.2	10.58	11.25	10.57
fft	0.01	0.01	0.41	0.10
lu_cb	0.08	2.42	1.87	0.13
ocean_cp	0.07	4.7	3.53	0.10
water_nsquared	0.04	3.02	2.48	0.06

表 2 检测错误数量对比

Table 2 Comparison of the number of detection errors

测试集	Movec	Valgrind	Asan
lbm_s	0	0	0
nab_s	$ML=1$	1	1($ML=1$)
xz_s	$SP=435$, $ML=48$	39	12($ML=12$)
botsalgn	$ML=5$	0	0
botsspar	$ML=1$	0	1($ML=1$)
smithwa	0	0	0
cholesky	$ML=3$	30850	1($ML=1$)
ffsb	$SP=58$, $TP=3941810$, $ML=43$	195	0
fft	$ML=6$	4	1($ML=1$)
lu_cb	$ML=6$	2	0
ocean_cp	0	6554311	0
water_nsquared	$ML=3$	0	0

由上述分析结果可知，Movec 可以对选取的专业测试集进行有效的插桩和错误的检测，其有效性以及综合性能比其他工具更加优秀。

结束语 针对 C 语言多线程程序的源代码插桩，提出了相关技术方案和算法，采取的技术包括指针元数据、CAS 无锁技术、源代码动态插桩技术。同时选取权威的专业测试集进行有效性和性能的验证。通过和其他工具的对比，结果表明，本文提出的内存安全动态分析方法对 C 语言多线是程有效的，综合性能也较为优秀。

下一步的工作是提升工具的性能，提高插桩后程序的运行速度，减少不必要的时间开销，减少插桩后程序增加的内存空间。

参 考 文 献

[1] CHEN Z, TAO C Q, ZHANG Z Y, et al. Beyond spatial and temporal memory safety[C]// Proceedings of the 40th International Conference on Software Engineering (ICSE 2018), Companion Volume. ACM, 2018: 189-190.

[2] XU W, DUVARNEY D C, SEKAR R. An efficient and backwards-compatible transformation to ensure memory safety of C programs[C]// Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering. 2004: 117-126.

[3] NETHERCOTE N, SEWARD J. How to shadow every byte of memory used by a program[C]// Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE 2007). ACM, 2007: 65-74.

[4] NAGARAKATTE S, ZHAO J Z, MARTIN M M K, 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 (PLDI 2009). ACM, 2009: 245-258.

[5] SIMPSON M S, BARUA R K. MemSafe: ensuring the spatial and temporal memory safety of C at runtime[J]. Software: Practice and Experience, 2013, 43(1): 93-128.

[6] CHEN Z, WANG C, YAN J Q, et al. Runtime Detection of Memory Errors with Smart Status[C]// Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021). Virtual, Denmark, ACM, 2021: 296-308.

[7] CHEN Z, YAN J Q, 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 (ISSTA 2019). Beijing, China, ACM, 2019: 341-351.

[8] CHEN Z, YAN J Q, LI W M, et al. Runtime verification of memory safety via source transformation[C]// Proceedings of the 40th International Conference on Software Engineering (ICSE 2018), Companion Volume. ACM, 2018: 264-265.

[9] MA R, CHEN L, HU C, et al. A dynamic detection method to C/C++ programs memory vulnerabilities based on pointer analysis[C]// 2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing. IEEE, 2013: 52-57.

[10] SEREBRYANY K, BRUENING D, POTAPENKO A, et al. AddressSanitizer: A fast address sanity checker[C]// 2012 {USENIX} Annual Technical Conference ({USENIX} ATC} 12). 2012: 309-318.

[11] NETHERCOTE N, SEWARD J. Valgrind: A program supervision framework[J]. Electronic Notes in theoretical Computer Science, 2003, 89(2): 44-66.

[12] NETHERCOTE N, SEWARD J. Valgrind: a framework for heavyweight dynamic binary instrumentation[C]// Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007). ACM, 2007: 89-100.

YAN Rui, born in 1998, postgraduate. His main research interest is software verification.

CHEN Zhe, born in 1981, Ph. D, associate professor. His main research interest is software verification.