计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.200100097

基于定理证明的内存安全性动态检测算法的正确性研究

孙小祥[1]　陈　哲[1,2,3]
1 南京航空航天大学计算机科学与技术学院　南京 211106
2 高安全系统的软件开发与验证技术工业和信息化部重点实验室　南京 211106
3 软件新技术与产业化协同创新中心　南京 211106
(834675173@qq.com)

摘　要　随着软件运行时验证技术的发展,出现了许多面向C语言的运行时内存安全验证工具。这些工具大多是基于源代码或者中间代码插桩技术来实现内存安全的运行时检测。但是,其中一些没有经过严格证明的验证工具往往存在两方面的问题,一是插桩程序的加入可能会改变源程序的行为及语义,二是插桩程序并不能有效保证内存安全。为了解决这些问题,文中提出了一种使用Coq定理证明器来判定内存安全验证工具算法是否正确的形式化方法,并使用该方法对C语言运行时验证工具Movec的动态检测算法的正确性进行了证明。对安全规范性质的证明结果表明了Movec的内存安全性动态检测算法是正确的。

关键词:运行时验证;源代码插桩;内存安全;定理证明;Coq
中图法分类号　TP309

Study on Correctness of Memory Security Dynamic Detection Algorithm Based on Theorem Proving

SUN Xiao-xiang[1] and CHEN Zhe[1,2,3]
1 College of Computer Science and Technology,Nanjing University of Aeronautics and Astronautics,Nanjing 211106,China
2 Key Laboratory of Safety-Critical Software,Ministry of Industry and Information Technology,Nanjing 211106,China
3 Collaborative Innovation Center of Novel Software Technology and Industrialization,Nanjing 211106,China

Abstract　With the development and improvement of software runtime verification technology,many C-oriented runtime memory security verification tools have appeared. Most of these tools are based on source code or intermediate code instrumentation technology to achieve memory-safe runtime detection. However,some of these verification tools that have not been rigorously proven often have two problems. One is that the addition of instrumentation programs may change the behavior and semantics of the source program,and the other is that instrumentation programs cannot effectively guarantee memory safety. In order to solve these two problems,this paper proposes a formal method that uses the Coq theorem prover to determine whether the memory security verification tool algorithm is correct,and uses this method to check the correctness of the dynamic runtime algorithm of the C language verification tool Movec Proven. The proof of the nature of the security specification shows that Movec's dynamic detection algorithm for memory security is correct.

Keywords　Runtime verification,Source code instrumentation,Memory safety,Theorem proof,Coq

1　引言

随着软件运行时验证技术的进步,基于各种运行时验证方法的软件验证工具被开发出来。使用这些运行时内存安全验证工具,能够发现软件系统中存在的很多内存安全漏洞。针对内存安全漏洞设计的这些验证工具,大多是基于代码插桩技术来实现的。但是,使用源代码插桩技术的验证工具存在两方面的问题:1)在源代码中加入验证代码可能会改变源程序的行为及语义,这也就违背了这项技术的初衷;2)并不能严格说明经过验证的程序就能够检测出所有的内存安全漏洞。针对这两个问题,本文提出了一种基于Coq[1-2]定理证明器的形式化证明方法来判定内存安全验证工具的算法是否正确。

运行时验证技术是在运行时对程序的内存安全性质进行验证,通过源代码插桩技术插入验证代码来监控指针的使用,可以检测出C语言程序中存在的时间和空间内存安全问题。例如,Movec[3-4],SoftBound[5],CCured[6]和CETS[7]等就是基于C语言程序插桩技术实现的运行时内存安全验证工具。

定理证明是一种以严格的逻辑推导和证明的方式来证明目标程序正确性的形式化方法[8-9]。对于软件系统的证明,通常是通过对该系统的语法语义规则和操作语义进行形式化定

到稿日期:2020-01-15　返修日期:2020-04-08　本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目: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 Joint Research Funds of National Natural Science Foundation and Civil Aviation Administration of China (U1533130) and Fundamental Research Funds for AI(NZ2020019).
通信作者:陈哲(zhechen@nuaa.edu.cn)

义来形式化地描述表示目标软件系统的算法结构、形式环境和操作语义等内容,然后通过对目标程序建立形式化模型、语义推导规则来证明形式化系统是否满足相关安全规范说明来证明目标程序的正确性。Coq 是目前用来证明安全关键性系统可靠性的使用最广泛且最高效的定理证明工具。Coq 的证明依据来源于归纳构造演算,并在其归纳数据类型和依赖积类型中结合当前程序设计语言的相关理念,从而使得 Coq 具备极其强大的形式语义描述能力。

本文将以 Movec 工具为例,通过定理证明的方法对 Movec 的内存安全性动态检测算法的正确性进行证明。Movec 是一个 C 程序运行时内存安全检测工具,利用 LLVM[10-11] 框架对源代码插桩,然后对插桩后的程序进行编译和运行,在运行时检测内存安全问题。由于插桩算法在源程序中插入了验证语句,从而更改了源程序的代码,这使得通过 Movec[12-16] 来验证 C 语言程序的内存安全的可靠性值得商榷。

本文首先使用 Coq 定理证明器来构建一个包含 Movec 验证算法的 C 语言形式化系统;第 2 节对该形式化系统的类型规则、环境、操作语义及相关公理进行形式化表示;第 3 节详细介绍 Movec 算法行为不变性和内存安全性的证明思路。

2 形式化系统的搭建

2.1 类型及环境

形式化的 C 语言片段的类型及语法规则如表 1 所列。其中,原子类型(Atomic Types)为形式化 C 语言中最基础的类型,包含了 int 类型和指针类型;结构体类型(Struct Types)表示程序中所有的结构体类型,成员变量的类型和偏移量由成员 id 确定;函数类型(Functions)表示程序中的函数调用并以 fid 作为函数的标识符;LHS 和 RHS 分别是 C 语言语法中赋值语句的左值和右值;Commands 表示函数中的一条完整语句。

表 1 C 语言类型及语法规则

Table 1 C Types and Grammar rules

<table>
<tr><th>Types</th><th></th><th>Grammar Rules</th></tr>
<tr><td>Atomic Types</td><td>a</td><td>$::=$ int | t *</td></tr>
<tr><td>Struct Types</td><td>s</td><td>$::=$ struct [id]{…; $id_i:a_i$;… }</td></tr>
<tr><td>Functions</td><td>f</td><td>$::=$ fid( ){…;id_i:[static] a_i;…;c }</td></tr>
<tr><td>LHS Expressions</td><td>lhs</td><td>$::=$ id | * lhs | lhs.id | lhs →id</td></tr>
<tr><td>RHS Expressions</td><td>rhs</td><td>:: const | fid | $rhs+rhs$ | &lhs | lhs | malloc (rhs) | (a)rhs | sizeof(t)</td></tr>
<tr><td>Commands</td><td>c</td><td>$::=$ skip | c;c | $lhs=rhs$ | fid( ) | rhs( ) | free (lhs)</td></tr>
</table>

在 C 语言的形式化表示中,为了更好地描述内存相关安全性质,我们将 C 程序的运行时内存划分为全局存储区 G、堆对象存储区 AMB、局部栈对象存储区 S,使用 M 来表示程序员可访问的内存空间。由此,可以根据内存划分将 C 语言的环境形式化表示为 $uE=(M,G,AMB,S)$,且 M,G,AMB 和 S 分别表示对应内存空间变量与地址间的映射关系。映射 M:$Loc \rightarrow Val$ 表示内存地址与存储值的映射关系;映射 AMB:$Loc \rightarrow Loc$ 表示一个记录动态分配内存块的表,由内存块起始地址可以映射到结束地址;映射 G 表示全局变量的映射关系,由 $Var \rightarrow Loc * Atype$ 可以一个从全局变量名映射到该变量的地址及原子类型;映射 S:$Var \rightarrow Loc * Atype * Loc$ 表示局部栈变量的类型地址映射关系。

表 2 中对 C 语言中的几种内存操作函数进行了形式化表示,分别形式化了内存的读写、内存块的分配与释放、函数栈的创建与销毁等操作原语。

表 2 内存操作原语

Table 2 Memory operation primitives

Primitives	Meaning
accessMem(M,Loc)	Access memory at Loc
storeMem(M,Loc,val)	Store val to the memory at Loc
malloc(E,n)	Allocate a block with size n
free(E,$ploc$)	Free a block pointed by $ploc$
createFrame(E,fr)	Allocate a frame for Funtion fr
destroyFrame(E)	Destroy the head frame

如图 1 所示,Movec 的内存安全性检测算法主要就是维护一个指针元数据的元数据表 PT(PmdTable)和一个记录指针指向对象的内存状态的状态表 ST(SndTable)。每个指针的元数据中记录了指向对象的起始地址 $base$、结束地址 end 和对象内存状态节点的地址 sa。每一个被指对象在 ST 中都有一个状态节点,记录了对象的当前内存状态 $stat$ 和当前指向对象的指针计数器 $count$。通过维护这样的内存结构,就可以动态地实时检测指针的状态是否有效。

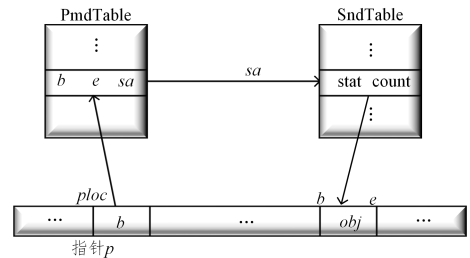

图 1 指针和对象内存结构

Fig. 1 Pointer and object memory structure

Movec 的动态检测算法是在指针解引用时通过判断指针值 b 是否位于元数据边界范围内来判断是否存在空间内存安全错误;同时通过查看 ST 表来查看指针指向的对象内存状态和判断指针是否有效,并以此来检测时间内存安全错误。另外,映射 PT:$Loc \rightarrow Pmd$ 模拟了指针元数据的访问读取;映射 ST:$Loc \rightarrow Snd$ 模拟了对象内存状态的访问与读取。因此,将带有 Movec 插桩算法的环境表示为 $E=(M,G,AMB,S,PT,ST)$。

2.2 操作语义

C 语言原本的操作语义中不会对指针的行为进行约束,这也是 C 语言程序容易出现内存安全问题的重要原因。而经过 Movec 验证输出的程序则会监控指针的使用,在指针的解引用、赋值传递时对指针元数据进行检测,以避免对指针非法解引用带来的内存安全问题。

C 语言的操作语义的形式化可以大致归纳为 Lhs,Rhs 和 Cmd,对应的语法规则的形式化表示如表 1 所列。Lhs 和 Rhs 是针对赋值语句而言的,将一个 Rhs 值存储到 Lhs 所指的内存空间上。Lhs 最终要获得的是内存地址,而 Rhs 要获得的是计算的最终值。Cmd 表示的是函数内 C 语言的语句序列。

Lhs 的形式化操作语义如图 2 所示。Lhs 上主要获得的是存储对象的地址,以及该对象的类型和内存状态。$Gvar$ 表示左值是一个类型为 t 的全局变量 vid,vid 在内存中的地址

为 loc，所有全局变量共享一个状态节点 g_sa；$Svar$ 表示类型为 t、地址为 loc 的局部变量；$Deref$ 表示指针解引用（$*Lhs$）的操作语义，首先 Lhs 的值是一个指针类型 $t*$ 的变量的地址 loc，通过 $accMP()$ 可以查询到该指针指向对象的地址和该指针的元数据，因此 $*Lhs$ 得到的是对象地址 loc' 和对象类型 t；Dot 和 $Arrow$ 分别表示通过 $Lhs.id$ 和 $Lhs \rightarrow id$ 来访问结构体成员变量的操作语义。

$$\frac{E.G(vid)=(loc,t)}{(E,vid)\Rightarrow_l loc_{g_sa}:t}(Gvar) \qquad \frac{E.S(vid)=(loc,t,sa)}{(E,vid)\Rightarrow_l loc_{sa}:t}(Svar)$$

$$\frac{(E,Lhs)\Rightarrow_l loc_{(.)}:t* \quad accMP(E.M,E.PT,loc)=loc'_{(b,e,sa)}}{(E,*Lhs)\Rightarrow_l loc'_{sa}:t}(Deref)$$

$$\frac{(E,Lhs)\Rightarrow_l loc_{sa}:s \quad getOffSet(s,id)=os \quad getType(s,id)=t}{(E,Lhs.id)\Rightarrow_l (loc'+os)_{sa}:t}(Dot)$$

$$\frac{\begin{array}{c}(E,Lhs)\Rightarrow_l loc_{(.)}:s* \quad accMP(E.M,E.PT,loc)=loc'_{(b,e,sa)} \\ getOffSet(s,id)=os \quad getType(s,id)=t \\ check_{dpv}(E.S,(loc'+os)_{(b,e,sa)},t)\end{array}}{(E,Lhs\rightarrow id)\Rightarrow_l (loc'+os)_{sa}:t}(Arrow)$$

图 2　*Lhs* 操作语义

Fig. 2　Semantics of evaluating *Lhs*

Rhs 的操作语义如图 3 所示。

$$\frac{}{(E,n)\Rightarrow_r (E,n_{(0,0,0)}:int)}(Const)$$

$$\frac{sizeof(t)=size}{(E,sizeof(t))\Rightarrow_r (E,size_{(0,0,0)}:int)}(Size)$$

$$\frac{F(fid)=(fr,c)}{(E,id)\Rightarrow_r (E,fid_{(fid,fid+1,f_sa)}:void(*)())}(Fun)$$

$$\frac{(E,Lhs)\Rightarrow_l loc_{(.)}:t \quad accMP(E.M,E.PT,loc)=v_{(b,e,sa)}}{(E,Lhs)\Rightarrow_r (E,v_{(b,e,sa)}:t)}(Lhs)$$

$$\frac{(E,Lhs)\Rightarrow_l loc_{(sa)}:t}{(E,\&Lhs)\Rightarrow_r (E,loc_{(loc,loc+sizeof(t),sa)}:t*)}(Ref)$$

$$\frac{(E,Rhs)\Rightarrow_r (E',v_{(b,e,sa)}:t) \quad dataCast(v_{(b,e,sa)},t,t')=v'_{(b',e',sa')}}{(E,(t')Rhs)\Rightarrow_r (E',v'_{(b',e',sa')}:t')}(Cast)$$

$$\frac{(E,Rhs1)\Rightarrow_r (E',m_{(.)}:int) \quad (E',Rhs2)\Rightarrow_r (E'',n_{(.)}:int)}{(E,Rhs1+Rhs2)\Rightarrow_r (E'',(m+n)_{(0,0,0)}:int)}(AddInt)$$

$$\frac{(E,Rhs1)\Rightarrow_r (E',loc_{(b,e,sa)}:t*) \quad (E',Rhs2)\Rightarrow_r (E'',n_{(.)}:int)}{(E,Rhs1+Rhs2)\Rightarrow_r (E'',(loc+n*sizeof(t))_{(b,e,sa)}:t*)}(AddPtr)$$

$$\frac{(E,Rhs1)\Rightarrow_r (E',n_{(.)}:int) \quad malloc(E',n)=(E'',loc,sa)}{(E,malloc(Rhs))\Rightarrow_r (E'',loc_{(loc,loc+n,sa)}:void*)}(Alloc)$$

图 3　*Rhs* 操作语义

Fig. 3　Semantics of evaluating *Rhs*

Rhs 最终获得一个值，当这个值是一个指针类型时，需要将指针的元数据一起传递给 *Lhs*。*Const* 和 *Size* 得到的都是一个常量，其类型为 int，因此元数据都为(0,0,0)。*Fun* 的语义是将一个函数作为赋值语句的右值，即将函数起始地址赋值给函数指针。*Lhs* 作为 *Rhs* 表示右值表达式的结果是另一个表达式的左值，即将 *Lhs* 地址作为右值赋值给另一个变量，因此 *Rhs* 的值就是 *Lhs* 对象的地址，元数据就是这个对象的边界及内存状态。

Ref 表示的是 &*Lhs*，因此 Rhs 得到的是 *Lhs* 的地址，那么赋值语句左值应该是一个指针，指针的元数据应该是 *Lhs* 对象的边界和状态。Cast 模拟 C 语言中的类型转换，将类型从 t 转换成 t'。AddInt 表示两个整数相加，结果是整数之和，元数据为(0,0,0)。AddPtr 表示的是指针运算，很多指针越界都是由指针运算引起的，AddPtr 的操作语义表达式中 *Rhs*1 表示的是指针，*Rhs*1 记录了对象的地址和元数据，*Rhs*2 为整数 n，$Rhs1+Rhs2$ 的值为($loc+n*sizeof(t)$)，元数据仍为 *Rhs*1 的元数据。如果 $Rhs1+Rhs2$ 的地址不在(b,e)之间，就会出现内存访问越界的情况，解引用时插桩算法就会调用中断程序以预防内存安全错误。Alloc 表示的是内存块动态分配的操作语义，*Rhs* 的值是分配的内存块的地址 loc，元数据为该内存块的边界和状态。

如图 4 所示为 *Cmd* 的部分插桩操作语义，C 语言的标准操作语义中是不含插桩检测函数的。

$$\frac{}{(E,skip)\Rightarrow_c (E,OK)}(Skip)$$

$$\frac{(E,c1)\Rightarrow_c (E',OK) \quad (E',c2)\Rightarrow_c (E'',OK)}{(E,c1;c2)\Rightarrow_c (E'',OK)}(Seq)$$

$$\frac{\begin{array}{c}(E,Lhs)\Rightarrow_l loc_{(.)}:t \quad (E,Rhs)\Rightarrow_r (E',v'_{(b',e',sa')}:t') \\ dataCast(v'_{(b',e',sa')},t',t)=v_{(b,e,sa)} \; isPtr(t) \\ storeMP(E'.M,E'.PT,E'.ST,loc,v_{(b,e,sa)})=(M',PT',ST')\end{array}}{(E,Lhs=Rhs)\Rightarrow_c (E'[M=M'PT=PT'ST=ST'],OK)}(AssignPtr)$$

$$\frac{\begin{array}{c}(E,Lhs)\Rightarrow_l loc_{(.)}:t \quad (E,Rhs)\Rightarrow_r (E',v'_{(b',e',sa')}:t') \\ dataCast(v'_{(b',e',sa')},t',t)=v_{(b,e,sa)} \quad \neg isPtr(t) \\ storeM(E'.M,loc,v)=M'\end{array}}{(E,Lhs=Rhs)\Rightarrow_c (E'[M=M'],OK)}(AssignNPtr)$$

$$\frac{\begin{array}{c}F(fid)=(fr,c) \quad createframe(E,fr)=E' \\ (E',c)\Rightarrow_c (E'',r) \quad \neg Error(r) \quad destroyframe(E'')=E'''\end{array}}{(E,id())\Rightarrow_c (E''',r)}(Call)$$

$$\frac{\begin{array}{c}(E,Rhs)\Rightarrow_r (E',loc_{(b,e,sa)}:void(*)()) \\ check_{dpfv}(E'.ST,loc_{(b,e,sa)}) \quad F(loc)=(fr,c) \\ createframe(E',fr)=E'' \quad (E'',c)\Rightarrow_c (E''',r) \\ \neg Error(r) \quad destroyframe(E''')=E_4\end{array}}{(E,Rhs())\Rightarrow_c (E_4,r)}(CallPtr)$$

$$\frac{\begin{array}{c}(E,Rhs)\Rightarrow_r (E',loc_{(b,e,sa)}:t*) \\ check_{free}(E'.ST,loc_{(b,e,sa)}) \quad free(E',loc,sa)=E''\end{array}}{(E,free(Rhs))\Rightarrow_c (E'',OK)}(Free)$$

图 4　*Cmd* 操作语义

Fig. 4　Semantics of evaluating *Cmd*

图 4 中的 *Cmd* 操作语义是包含了指针检测函数的操作语义，是 Movec 插桩检测算法的形式化描述，主要描述了 C 语言中几种指针相关语句的执行过程，主要会在指针赋值 AssignPtr、函数指针调用 CallPtr、指针释放 Free 的操作语义中插入对指针的检测。AssignPtr 的语义中，*Lhs* 表示的是一个类型为 t 的指针，*Rhs* 得到的是一个类型为 t' 的对象 v'，且该对象的元数据为(b',e',sa')。当类型 t 和类型 t' 不相同时，Movec 的内存安全检测算法就会执行类型转换操作 datacast，最后将 *Lhs* 的地址写入 *Rhs* 对象内存中，并更新该对象的指针元数据。由于指针赋值并不会使用指针，因此不需要检测指针，只需要更新元数据即可。对于函数指针 CallPtr 的操作语义，*Rhs* 首先获得该函数指针所指向的函数入口地址及该指针的元数据，在调用函数前先检查函数指针值是否在指针元数据记录的边界范围内，并判断状态节点中的状态是否为 function。如果符合规范说明，则根据函数入口地址找到该函数并创建相应的函数栈存储局部变量。最后，如果函数语句执行没有错误，就销毁函数栈，函数执行结束。Free 函数的操作语义中，*Rhs* 首先得到一个指针及元数据，在释放指针指向的内存空间之前需要检查该指针值是否在元数据记录的边界范围内，并检查对象的内存状态是否为 *heap*。若不在边界范围内，表示当前指针已越界；若内存状态不是 *heap*，表示当前释放的内存不是一个合法内存。

2.3　相关公理

C 语言环境的形式化表示中对内存读写、内存分配与释放、函数栈创建与销毁等系统调用进行了定义。Movec 的插桩算法中对这些系统调用进行了包装，在调用系统函数前后处理指针元数据。此外，公理系统中还将内存按照其存储值

的类型划分为了程序区、全局和静态变量区、堆、栈。

公理 1 （umalloc） If umalloc uE size = some （M', AMB', l）, then

性质 1 baseAddr $\leqslant l \wedge l+$size$<$maxAddr $\wedge$ size>0

性质 2 $\forall$（accessMem $M\ l'$ = some data）. accessMem $M'\ l'$ = some data

性质 3 $\forall(l'<l \vee l' \geqslant l+$size$)$. accessMem $M\ l'$ = none$\Rightarrow$ accessMem $M'\ l'$ = none

性质 4 $\forall(l \leqslant l' < l+$size$)$. accessMem $M\ l'$ = none $\wedge$ accessMem $M'\ l' = 0$

性质 5 AMB' = addAllocMemBlock uAMB $b(b+n)$

内存分配应保持如公理 1 所示性质。性质 1 表示系统只在堆内存[baseAddr，maxAddr）上分配空间；性质 2 和性质 3 表示堆内存分配不影响区域外的内存块分配，包括已分配的和未分配的；当前分配的内存块之前是未分配的，且赋值为 0；更新 AMB 内存块映射表。内存释放应保持的性质与内存分配类似：系统释放的内存在堆内存上；堆内存释放不影响区域外的内存块释放；释放的内存块之前是已分配的，释放后为未分配状态；更新 AMB 内存块映射表。

函数栈分配应保持公理 2 所示的性质。性质 1 表示系统只在栈顶分配空间；性质 2 和性质 3 表示函数栈分配不影响区域外的内存分配；性质 4 表示分配的函数栈之前是未分配的，赋值为 0。销毁栈顶函数栈应保持如下性质：系统只销毁当前栈顶的函数栈；不影响区域外的内存块。函数栈创建与函数栈销毁类似，在此不赘述。

公理 2 （ucreateFrame） If ucreateFrame $uE\ fr$ = some uE'，then

性质 1 $fr.(to) = uS.(top) \wedge fr.(from) = uS'.(top)$

性质 2 $\forall$（accessMem Ml = some data）. accessMem $M'l$ = some data

性质 3 $\forall(l < fr.(\mathrm{from}) \vee l \geqslant fr.(\mathrm{to}))$. accessMem Ml = none$\rightarrow$accessMem $M'l$ = none

性质 4 $\forall(fr.(\mathrm{from}) \leqslant l < fr.(\mathrm{to}))$. accessMem Ml = none $\wedge$ accessMem $M'l = 0$

3 证明定理

3.1 行为不变性证明

行为不变性是指如果插桩程序正确执行，那么未插桩的 C 程序运行结果与之相同。行为不变性是保证运行时内存检测工具算法有效性的前提。首先介绍内存操作的行为不变性。下面以 malloc 包装函数为例来说明内存操作不变性的证明，其他内存操作的证明与之类似。这个引理表示插桩程序在环境 E 中分配一个大小为 n 的内存块，返回一个新环境 E' 和内存块的起始地址及内存状态地址，原程序结果与之相同。

```
Lemma malloc_invariance: forall E n E' b sa,
  malloc E n = Some(E', b, sa) ->
  umalloc(Env2uEnv E) n = Some(Env2uEnv E', b).
```

在 Movec 的具体实现中，为了跟踪指针元数据，malloc 等函数的包装函数是通过对 C 语言提供的系统调用进行包装实现的。如果包装函数成功执行，那么包装函数内的系统调用必然成功执行。上述引理 malloc_invariance 描述的是包装函数的行为不会改变原有系统调用的行为，因为包装函数内部就是原有的系统调用。在形式化表示包装函数时，也是通过 C 语言系统调用来实现的。因此。在 Coq 中证明该引理时，只要展开包装函数 malloc 的定义，就可以直接归纳推出 umalloc。

Movec 的插桩操作语义是通过扩展 C 语言操作语义来实现的，对 C 中各种未定义内存安全错误进行了语义覆盖——在可能出现内存安全错误的地方插入指针检测函数。

```
Lemma lhs_invariance: forall E lhs loc sa t,
  s_lhs E lhs(RLoc(loc, sa)) t ->
  us_lhs(Env2uEnv E) lhs(uRLoc loc) t.
Lemma rhs_invariance: forall E rhs v p t E',
  s_rhs E rhs(RVal(v, p)) t E' ->
  us_rhs(Env2uEnv E) rhs v t(Env2uEnv E').
Lemma cmd_invariance: forall E cmd E',
  s_cmd E cmd OK E' ->
  us_cmd(Env2uEnv E) cmd uOK(Env2uEnvE').
```

上面 3 个引理分别描述了 Movec 的插桩操作语义的行为不变性。如果插桩操作语义所推出的结果是正确的，那么非插桩操作语义的执行结果也是正确的。证明操作语义的行为不变性时，需要将 C 语言的标准操作语义和 Movec 插桩操作语义展开并逐个归纳证明（由于证明代码过于复杂，此处不展示）。

3.2 内存安全性证明

证明 Movec 内存操作和操作语义的内存安全性，就是要证明经过这些操作之后的环境不会出现内存安全错误。本文将没有内存安全错误的环境称为良构环境。首先要定义良构环境所满足的内存安全规范说明：一个环境是良构的，需要对内存中各个部分的对象保持良构性质。这里简要介绍部分内存安全规范性质：存在区域合理性，比如全局变量应该位于全局数据区，堆栈变量位于堆栈内存空间；内存对象的存储状态是有效的，如堆对象状态为 heap，局部栈对象的状态为 stack 等；内存有效性，即良构的内存对象的地址空间都是已分配的、可访问的；对象间无重叠，即任意两个对象之间不存在交集等。下面以内存分配函数 malloc 来说明内存操作安全性的证明，其他内存操作的证明与之类似。

```
Lemma malloc_wfEnv : forall E n E' b sa,
  wfEnv E -> malloc E n = Some(E', b, sa)
  -> wfEnv E'.
```

malloc_wfEnv 中描述了：如果初始环境 E 是良构的，malloc 函数执行后返回一个新的环境 E'、内存块首地址和状态地址 sa，那么 E' 也是良构的。要证明 E' 是良构的，只要证明环境 E' 的内存满足上面定义的所有内存安全规范说明即可。证明时，需要展开良构环境 wfEnv 的定义，通过 E 的良构性质并结合 malloc 包装函数所保持的性质去推导 E' 所有的良构性质即可得证。

```
Lemma wf_lhs: forall E fr lhs t,
  wfEnv E -> fr = StackVar2frame S -> wf_lhs G fr
```

lhs t—>
(exists l sa, s_lhs E lhs(RLoc(l, sa)) t) \/s_lhs E lhs RAbort t.
Lemma wf_rhs: forall E S fr rhs t,
wfEnv E—> fr=StackVar2frame S—>
wf_rhs G fr rhs t—>
(exists vp E', s_rhs E rhs(RVal vp) t E') \/(exists t' E', s_rhs E rhs RAbort t' E') \/(exists t' E', s_rhs E rhs RSErr $t'E'$).
Theorem wf_cmd: forall E S fr c,
wfEnv E —> fr=StackVar2frame S—>
wf_cmd G fr c—>
(exists E', s_cmd E c ROK E') \/
(exists E', s_cmd E c RAbort E') \/
(exists E', s_cmd E c RSystemError E').

wf_lhs 中证明了 *Lhs* 要么得到一个对象的地址，要么出错被中断。wf_rhs 中证明了 *Rhs* 的执行结果可能是一个良构的值，可能是由于内存安全错误被 Movec 算法中断，亦可能由于系统调用出错而终止。wf_cmd 中证明了 C 程序的执行结果要么是正确的，要么是出现内存安全错误被 Movec 终止，要么是系统调用出错中断(如内存空间不够分配失败等)。证明 *wf_lhs*、*wf_rhs* 和 *wf_cmd* 时，需要分别展开这几个操作语义的定义，分别从操作语义的每一种情况来证明目标结论。插桩操作语义中对所有的指针解引用都插入了检测函数，如果检测函数检测到错误，则结果是 Abort；如果检测函数结果正确，则结果是 OK；若出现系统调用失败，则结果是 SystemError，从而保证了对内存错误检测的有效性。

结束语 本文设计并实现了一种基于定理证明的判定内存安全性动态检测算法正确性的方法。对于任意的内存安全工具，只要能在 Coq 中形式化描述其算法过程，就能判断该工具是否满足相关安全规范说明。最终的证明表明，Movec 工具的算法满足了行为不变性和内存安全性两方面的要求。在接下来的工作中，我们将进一步优化证明步骤，提高证明方法的效率。

参考文献

[1] BERTOT Y, CASTÉRAN P. Interactive theorem proving and program development: Coq' Art: the calculus of inductive constructions[M]. Springer Science & Business Media, 2013: 12-25.

[2] CORBINEAU P. A declarative language for the coq proof assistant[C]// International Workshop on Types for Proofs and Programs. Berlin: Springer, 2007: 69-84.

[3] CHEN Z, MOTET G. Nevertrace claims for model checking [C]// International SPIN Workshop on Model Checking of Software. Berlin: Springer, 2010: 162-179.

[4] CHEN Z, WANG Z, ZHU Y, et al. Parametric Runtime Verification of C Programs[C]// International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2016: 299-315.

[5] NAGARAKATTE S, ZHAO J, MARTIN M M K, et al. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C[J]. ACM SIGPLAN Notices, 2009, 44(6): 245.

[6] NECULA G C, CONDIT J, HARREN M, et al. CCured: type-safe retrofitting of legacy software[J]. ACM Transactions on Programming Languages and Systems(TOPLAS), 2005, 27(3): 477-526.

[7] NAGARAKATTE S, ZHAO J, MARTIN M M, et al. CETS: compiler enforced temporal safety for C[J]. ACM Sigplan Notices, 2010, 45(8): 31-40.

[8] MARONEZE A, PERRELLE V, KIRCHNER F. Advances in Usability of Formal Methods for Code Verification with Frama-C[J]. Electronic Communications of the EASST, 2019, 77.

[9] FULARA J, JAKUBCZYK K. Practically applicable formal methods[C]// International Conference on Current Trends in Theory and Practice of Computer Science. Berlin: Springer, 2010: 407-418.

[10] SEREBRYANY K, POTAPENKO A, ISKHODZHANOV T, et al. Dynamic race detection with LLVM compiler[C]// International Conference on Runtime Verification. Berlin: Springer, 2011: 110-114.

[11] YANG L, GURUMANI S, CHEN D, et al. AutoSLIDE: Automatic source-level instrumentation and debugging for HLS [C]// 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2016: 127-130.

[12] CHEN Z. Parametric runtime verification is NP-complete and coNP-complete[J]. Information Processing Letters, 2017, 123: 14-20.

[13] CHEN Z, GU Y, Huang Z, et al. Model checking aircraft controller software: a case study[J]. Software: Practice and Experience, 2015, 45(7): 989-1017.

[14] 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.

[15] CHEN Z, TAO C, ZHANG Z, et al. Poster: Beyond Spatial and Temporal Memory Safety[C]// 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion). IEEE, 2018: 189-190.

[16] 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.

SUN Xiao-xiang, born in 1994, postgraduate. His main research interests include software verification and so on.

CHEN Zhe, born in 1981, Ph.D, professor, Ph.D supervisor, is a member of China Computer Federation. His main research interests include software verification and so on.