

一种汇编程序的形式验证框架

李兆鹏 陈意云 葛琳 华保健

(中国科学技术大学计算机科学与技术系 合肥 230026)

(中国科学技术大学苏州研究院软件安全实验室 苏州 215123)

(zpli@mail.ustc.edu.cn)

A Formal Certifying Framework for Assembly Programs

Li Zhaopeng, Chen Yiyun, Ge Lin, and Hua Baojian

(Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230026)

(Software Security Laboratory, Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123)

Abstract Proof-carrying code brings two grand challenges to the research field of programming languages. One is to study the technology of certifying compilation. The other is to seek more expressive program logics or type systems to specify or reason about the properties of high-level or low-level programs. And safety is an important issue among the properties of high-assurance software. The verification method for software to meet its safety policies is one of the hot researches. In terms of the framework to design and verification of safety programs, and the pointer logic proof system, this paper introduces the research on the formal description of target machine, the formal certifying framework for assembly programs and property proof of assembly pointer programs. The main characteristics of the design and implementation are as follows: first, the design of the certifying framework is based on program verification method of Hoare style; second, program property related with pointers is proved using a pointer logic which is similar to the counterpart in the level of source language; and finally, a simple type system is designed to fulfill type checking on pointers. Moreover, this work has been formalized in the proof assistant Coq and all code is available on the website of the authors' laboratory.

Key words software safety; certifying compiler; pointer logic; Hoare logic; certified assembly program

摘要 在高可信软件的各种性质中,安全性是关注的重点.软件满足安全策略的证明方法是安全性研究的热点之一.根据前期提出的安全程序设计与证明的框架以及指针逻辑推理系统,介绍在所实现的出具证明编译器(certifying compiler)原型系统中有关目标机器的形式定义、汇编程序的形式验证框架以及汇编程序指针程序性质证明等方面的研究.它们的主要特点是汇编验证框架是基于 Hoare 风格的程序验证方式;与指针有关的性质使用和源语言一级类似的指针逻辑推理系统进行证明;使用一个简单的类型系统完成有关指针的类型检查.

关键词 软件安全;出具证明编译器;指针逻辑;Hoare 逻辑;携带证明的汇编程序

中图法分类号 TP301

随着国家和社会对软件的依赖程度日益增长,关键软件的高可信性质显得越来越重要,其中安全性(包括可靠安全性和保密安全性)是关注的重点.可靠安全性是指软件运行时不引起危险、灾难的能力,而保密安全性是指软件系统对数据和信息提供保密性、完整性、可用性、真实性保障的能力.本文所讲的安全性主要是指可靠安全性.但是两者也是紧密相连的,提高可靠安全性有助于保证保密安全性.

提高软件安全性的目标是所有的程序错误在程序运行前被发现或者在程序运行时被温和地捕获,以保证程序不会导致不可预测的系统行为.而软件安全性研究主要是探索如何建立一个管理安全性的健全的科学和技术基础,其中软件满足安全策略的程序性质证明方法是研究的热点之一.

近 10 年来,程序性质证明研究领域有了很大的发展.许多学者提出了不同的思路,这些思路主要采用基于类型的或基于逻辑的方法来证明高级语言程序或低级语言程序的性质.其中基于类型方法的典型研究有类型化汇编语言(typed assembly language)^[1]和类型细化(type refinement)^[2];基于逻辑方法的典型研究有携带证明的代码(proof-carrying code, PCC)^[3]和 FPCC(foundational proof-carrying code)框架^[4].基于类型的方法可以实现验证的完全自动化,但是要么表达能力差,要么类型系统过于复杂;基于逻辑的方法优点是表达能力强,但是一般来说使用该方法不能自动完成程序性质的证明,而需要程序员与定理证明工具进行交互.鉴于这两种方法具有很大的互补性,近年来出现了一些结合这两种方法的研究.一种结合类型和逻辑方法的研究是 Xi 的应用类型系统(applied type system, ATS)项目^[5],他们扩展类型系统,将程序状态引入类型系统,依靠 ATS 与 Hoare 逻辑的相似性,以 ATS 来编码 Hoare 逻辑,从而可以在类型系统上模拟 Hoare 逻辑的推理.另外,Shao 等人在携带证明的汇编编程 CAP(certified assembly programming)^[6-8]等项目中,使用了一种语法方式的 FPCC,并且提出一种采用 Hoare 风格、简单而又灵活的汇编代码模块验证框架.

分析当前研究现状,程序性质证明采用类型方法和逻辑方法相结合方式可以兼得两种方法的长处.而 ATS 的类型系统过于复杂,以致普通程序员难以掌握;CAP 方式的验证需要程序员进行繁复的汇编程序手工证明,不适合用于大规模开发.

因此,我们选择采用一个简单的类型系统结合一个自动的逻辑推理系统的方法来证明程序满足安全策略.对那些有高安全要求的关键软件、程序设计

及安全性证明可以在一种新的编程和编译框架下进行^[9-10].在这个框架下,我们选择类 C 语言 PointerC(带有动态存储分配)^[11]作为源语言,实现了一个出具证明编译器的原型;并且实现了类型安全和内存安全等基本安全策略.该出具证明的编译器在把源程序和规范翻译成汇编程序和等效规范的同时,将源程序满足规范的证明翻译成汇编代码满足等效规范的证明.在汇编语言一级由证明检查器(proof checker)利用代码所携带的证明进行代码满足规范的检验.汇编语言上的验证可以将编译器、验证条件生成器、定理证明器等排除出被信任的计算基础(trusted computing base, TCB),以尽量缩小系统的 TCB.因而整个编译框架最终依赖于汇编程序形式验证框架,来保证产生的汇编程序满足汇编语言一级等效的安全规范.

本文重点介绍我们设计的汇编程序形式验证框架,它基于 Intel x86 体系结构的目标机器模型,采用 Hoare 风格的程序验证方式.该框架包括断言与规范语言、合式公式、推理规则和可靠性定理.此外,本文还将介绍一个汇编语言指针逻辑系统,它保证通过验证的程序不存在悬空指针引用,或对动态分配存储的访问越界等.部分内容限于篇幅只做简要叙述.

1 方法概述

为便于安全性检查,在 PointerC 中对指针运算进行了限制.指针型的变量只能用于赋值、相等和不相等比较、存取指向对象等运算以及作为函数的参数,而指针算术和取地址运算等被禁止.在 PointerC 程序的性质证明方面,采用类型方法和逻辑方法相结合的方式.一方面,PointerC 定型规则使用附加条件约束表达式的值,以求得类型系统的简单明了和语言的安全;另一方面,为尽可能完成对这些附加条件的静态检查,以保证程序运行时不会违反基本安全策略,我们为 PointerC 设计了一种指针逻辑^[12].它是 Hoare 逻辑的一种拓展,本质上是一种精确指针分析工具.它可用来从前向后收集程序点各个指针的信息.这些信息包括指针是否为 NULL 指针、悬空指针还是有效指针(有指向对象的指针)的信息,以及各有效指针之间是否相等的信息.这些信息可以用来证明指针程序是否满足有关定型规则的附加条件,支持同值有关的静态检查.

我们设计的出具证明编译器在遍历源程序时,完成词法和语法分析、类型检查、断言生成、验证条件生成和中间代码生成,同时完成验证条件的证明;

然后基于汇编程序验证框架,生成携带断言和证明的汇编程序。

为了理解所采用的汇编程序形式化及其性质证明方法,需要回顾 Hoare 逻辑用于证明源语言程序性质的方法:首先用 Hoare 逻辑定义语言的公理语义,然后定义语言的操作语义,最后证明公理语义对操作语义是可靠的. 公理语义按照如下步骤定义:

1) 先定义断言语言,通常它是在该语言布尔表达式语法的基础上扩展而成;

2) 再定义 Hoare 逻辑的合式公式(well-formed formula),其形式为 $\{P\} S \{Q\}$,其中 P 和 Q 为断言, S 是程序段;

3) 最后定义一组面向这种合式公式的结构化推理规则,不同的语法结构有不同的推理规则,例如赋值公理为 $\{Q[x \leftarrow e]\} x = e \{Q\}$.

这就得到一个证明系统,它也是该语言的公理语义,可用来证明程序的性质。

定义操作语义通常是先定义机器状态,然后再定义各语法结构的结构化操作语义,它们都是状态转换函数. 而证明公理语义对操作语义可靠性,主要是证明公理语义的所有推理规则在操作语义中都有效. 例如对于赋值公理就是证明在任何满足 $Q[x \leftarrow e]$ 的状态下执行 $x = e$,其结果状态一定满足 Q .

由于所验证程序的性质并不能用简单类型系统来表达,我们在汇编程序验证框架中借鉴了 SCAP (stack-based CAP)的方式. 它是在汇编语言级使用 Hoare 风格的程序验证方法来开发携带证明的代码的通用框架. 由于只要求对一种(或一个系列)机器的操作语义有效,因此我们将 Hoare 逻辑方法直接绑定到该机器的操作语义上,使其与源语言级证明方法有显著区别:

① 断言以机器状态为参数,是直接对机器状态性质的描述;

② 基础的推理规则是基于指令的操作语义,而不像 Hoare 逻辑赋值公理那样基于语法代换;

③ 可靠性证明是对合式程序能维持安全运行的一种归纳证明。

总的来说,我们提出的汇编程序验证框架是基于一个 Intel x86 目标机器模型,它包括断言与规范语言、合式公式、推理规则和可靠性证明. 其中目标机器形式定义了汇编程序的语法以及指令的操作语义,是该框架的基础;断言与规范是用来描述程序满足的安全性质. 而合式公式以及推理规则构成了该验证框架的逻辑推理系统,可靠性证明保证了经过验证的程序可以一直安全执行. 下面分节进行详细阐述。

2 目标机器的形式化

在 FPCC 验证框架中,机器指令的操作语义采用元逻辑(meta-logic)形式化. 另外,程序逻辑系统和类型系统以及它们的可靠性证明也形式地定义在元逻辑中. 本节介绍所采用的元逻辑以及目标机器模型 TM(target machine)的形式化:语法和操作语义。

我们采用 CiC 演算(calculus of inductive constructions)^[13]作为元逻辑. CiC 是构造演算(the calculus of constructions)支持归纳定义的一个扩展. 通过 Curry-Howard 同构,构造性演算对应于 Church 高阶谓词逻辑. 本文提出的汇编程序验证框架使用了支持 CiC 的 Coq 证明辅助工具^[14]实现,所有实现可以自由获得^[15].

图 1 给出了目标机器模型 TM 的语法定义. 一个程序 P 由一个代码堆 C 、程序状态 S 和当前执行的函数 f 的指令块 I 组成. 代码堆 C 是函数名 f 到函数代码 Ψ 的部分映射. 函数代码 F 是标号 l 到指令块 I 的部分映射。

(Program)	$P ::= (C, S, (f, I))$
(CodeHeap)	$C ::= \{f \rightarrow F\}^*$
(Functions)	$F ::= \{l \rightarrow I\}^*$
(State)	$S ::= (M, R)$
(Memory)	$M ::= \{l \rightarrow z\}^*$
(RegFile)	$R ::= \{r \rightarrow z\}^* \{zf \rightarrow z\} \{sf \rightarrow z\}$
(Integers)	$f, l, z ::= \dots \mid -3 \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid 3 \mid \dots$
(Registers)	$r ::= \text{eax} \mid \text{ebx} \mid \text{ecx} \mid \text{edx} \mid \text{esi} \mid \text{edi} \mid \text{ebp} \mid \text{esp}$
(InstrBlocks)	$I ::= c; I \mid \text{jmp } l \mid \text{ret} \mid \text{je } l, l_n \mid \text{jge } l, l_n$ $\mid \text{jg } l, l_n \mid \text{call } f, l_{ret} \mid \dots$
(Commands)	$c ::= \text{addr } r_s, r_d \mid \text{subr } r_s, r_d \mid \text{push } r_s$ $\mid \text{movld } z(r_b), r_d \mid \text{pop } r_d \mid \dots$

Fig. 1 Syntax of the target machine.

图 1 目标机器的语法

在 TM 中,代码的位置并不重要,因此没有指令地址的概念,而只有指令块标号的概念. 所以程序三元组中的 (f, I) 相当于程序计数器 pc. 而代码堆的两级层次便于把一组指令块看做函数来进行模块化验证。

程序状态 S 包含内存 M 和寄存器文件 R 两部分. R 是寄存器名到整数值的映射,其中 zf 和 sf 是用于存放标志位的特殊寄存器. M 是地址到其所存放整数值的部分映射,包括栈和数据堆两部分. 由于 malloc 和 free 被看成是满足基本安全策略的预定义函数,例如 malloc 任何一次调用都能成功并且

所分配空间同尚未释放空间(包括代码区和栈区等)无任何重叠,因此在 TM 中数据堆和栈区不会相交。

从指令块的产生式可以看出,指令块以控制转移指令结尾.这里的指令块亦即编译中代码生成及优化涉及的基本块.限于篇幅,这里只列出了 Intel x86 最常用的指令,而指令集的扩充是直截了当的.有些指令的操作码略有区别是为了使证明简短.另外,在 TM 中没有使用程序计数器,因此条件转移指令和 call 指令都增加了一个标号,例如,call f, l_{ret} 中的 l_{ret} 为返回地址。

TM 指令的操作语义定义如图 2 所示,程序的一步执行被形式化为小步转移 (small-step transition) 关系 $P \mapsto P'$. 图 2 上半部分是指令块的操作语义,其中辅助函数 $Next_c(_)$ 定义了顺序指令的计算副作用。

If $I =$	Then $(C, (M, R), (f, D)) \mapsto$
jmp l	$(C, (M, R), (f, C(f)(l)))$
ig l, l_n	$(C, (M, R), (f, C(f)(l_n)))$ If $R(zf) = 0$ and $R(sf) = 0$; $(C, (M, R), (f, C(f)(l)))$ If $R(zf) = 1$ and $R(sf) = 1$.
call f_1, l_{ret}	$(C, (M\{R(esp) - 4 \mapsto f, R(esp) - 8 \mapsto l_{ret}\},$ $R\{esp \mapsto R(esp) - 8\}), (f_1, C(f_1)(l_{entry})))$ If $l_{ret} \in \text{dom}(C(f))$ and $f_1 \neq \text{main}$
ret	$(C, (M, R\{esp \mapsto R(esp) + 8\}), (f_{ret}, C(f_{ret})(l_{ret})))$ If $l_{ret} = M(R(esp)), f_{ret} = M(R(esp) + 4), f_{ret} \in \text{dom}(C),$ $l_{ret} \in \text{dom}(C(f))$ and $f \neq \text{main}$ $(C, (M, R), (\text{main}, \text{ret}))$ If $f = \text{main}$
$c; I'$	$(C, Next_c(M, R), (f, I'))$

where

If $c =$	Then $Next_c(M, R) =$
addr r_s, r_d	$(M, R\{r_d \mapsto R(r_d) + R(r_s)\})$
subr r_s, r_d	$(M, R\{r_d \mapsto R(r_d) - R(r_s)\})$
movld $z(r_s),$ r_d	$(M, R\{r_d \mapsto M(R(r_s) + z)\})$ If $R(r_s) + z \in \text{dom}(M)$
push r_s	$(M\{R(esp) - 4 \mapsto R(r_s)\}, R\{esp \mapsto R(esp) - 4\})$

Fig. 2 Part of the operational semantics of the target machine.

图 2 目标机器的操作语义(部分)

3 汇编程序验证框架

在第 2 节目标机器形式定义的基础上,本节介绍我们所设计的基于 Hoare 风格的汇编程序验证框架.该框架主要包括断言与规范语言、合式公式、推理规则和可靠性证明.在目标机器 TM 上对程序进行 Hoare 风格的推理,程序员首先要使用断言与规范语言对代码堆(程序)进行标注,然后使用合适

的推理规则进行程序良形性(well-formedness)的定理证明。

1) 断言与规范语言.断言语言直接使用元逻辑.如图 3 所示,断言的类型是 $State \rightarrow State \rightarrow Prop$,即断言是以两个状态为参数的谓词.它们分别是所在函数的初始状态和当前状态.使用函数初始状态便于表达函数返回值和参数初值之间的联系,也便于表达汇编程序的一些特定性质.例如在一个函数运行时,当前活动记录中保存的返回地址和上一个活动记录的基地址都没有被修改,其他活动记录的内容也没有被修改等.元逻辑中的 Prop 在此没有被进一步描述,这是为了便于定制不同的逻辑,使该验证框架能被用于不同场合。

(CdHpSpecCntxt)	$\Psi ::= \{f \rightarrow l \rightarrow a\}^*$
(FunSpecCntxt)	$\Sigma ::= \{f \rightarrow b\}^*$
(Assertion)	$p, q \in State \rightarrow State \rightarrow Prop$
(CmdSpec)	$a ::= p$
(FunSpec)	$b ::= (p, q)$

Fig. 3 Syntax of the assertion and specification.

图 3 断言和规范的语言

函数规范中的 p 和 q 分别对应函数前后条件.指令块规范只需要前条件 p ,而省略了后条件,这是因为可以使用后继指令块的前条件来进行推理(一个指令块的后断言应该蕴涵它后继指令块的前断言)。

使用断言与规范语言对代码堆的每个指令块、函数进行标注,即定义代码堆规范环境 Ψ 和函数规范环境 Σ .由于一个函数满足规范的证明和被调用函数的规范有关, Σ 同时也包含被调函数的规范。

2) 合式公式.它本质上和 Hoare 逻辑的合式公式是一致的.断言加到代码块、函数和程序上,形成 5 个层次的合式公式:合式代码块、合式函数体、合式函数、合式代码堆和合式程序,如图 4 所示.下面使用这些合式公式来定义推理规则。

(Well-formed InstrBlock)	$\Psi(f); \Sigma \vdash \{a\} I$
(Well-formed FuncBody)	$\Psi(f); \Sigma \vdash C(f); \Psi(f)$
(Well-formed Function)	$\Psi(f); \Sigma \vdash \{(p, q)\} C(f)$
(Well-formed CodeHeap)	$\Psi; \Sigma \vdash C; \Sigma$
(Well-formed Program)	$\Psi; \Sigma \vdash \{a\} P$

Fig. 4 Well-formed formulas.

图 4 合式公式

3) 推理规则.图 5 给出了部分推理规则.一个程序 $(C, S, (f, I))$ 是合式的(规则 PROG),仅当存在规范环境 Ψ 和 Σ ,使得代码堆 C 是合式代码堆, a 在当前状态 S 下为真,并且 I 是合式指令块。

$\Psi; \Sigma \vdash \{a\} P$ (Well-formed Program)	
$\frac{\Psi; \Sigma \vdash C; \Sigma \forall S_0. a S_0 S \Psi(f); \Sigma \vdash \{a\} I}{\Psi; \Sigma \vdash \{a\} (C, S, (f, I))}$	(PROG)
$\Psi; \Sigma \vdash C; \Sigma$ (Well-formed CodeHeap)	
$\frac{\forall f \in \text{dom}(\Sigma), \Psi; \Sigma \vdash \{\Sigma(f)\} C(f)}{\Psi; \Sigma \vdash \{f \rightarrow C(f)\}; \{f \rightarrow \Sigma(f)\}}$	(CDHP)
$\Psi(f); \Sigma \vdash \{(p, q)\} C(f)$ (Well-formed Function)	
$\frac{\Psi(f); \Sigma \vdash C(f); \Psi(f) (p_f, q_f) = \Sigma(f) \quad a_0 = \Psi(f) (\text{entry})}{\forall S. p_f S S \rightarrow a_0 S S \quad I = C(f) (\text{exit}) \wedge \text{EndWithRet}(I)}$	
$\Psi(f); \Sigma \vdash \{(p_f, q_f)\} C(f)$	(FUNC)
$\Psi(f); \Sigma \vdash C(f); \Psi(f)$ (Well-formed FuncBody)	
$\frac{\Psi(f); \Sigma \vdash \{a\} I}{\Psi(f); \Sigma \vdash \{1 \rightarrow I\}; \{1 \rightarrow a\}}$	(FBLK)
$\Psi(f); \Sigma \vdash \{a\} I$ (Well-formed InstrBlock)	
$\frac{\Psi(f); \Sigma \vdash \{a'\} I}{\forall S_0, S. a S_0 S \rightarrow a' S_0 \text{ Next}_c(S)}$	(SEQ)
$\frac{\Psi(f)(1) = a' \quad \forall S_0, S. a S_0 S \rightarrow a' S_0 S}{\Psi(f); \Sigma \vdash \{a\} \text{ jmp } 1}$	(JMP)
$\frac{f' \neq \text{main} \quad \Sigma(f') = (p', q') \quad \Psi(f)(l_{\text{ret}}) = a' \quad \forall S_0, S. a S_0 S \rightarrow p' S' S''}{\forall S_0, S, S'. a S_0 S \rightarrow q' S' S' \rightarrow a' S_0 S'}$	
$\Psi(f); \Sigma \vdash \{a\} \text{ call } f', l_{\text{ret}}$	
其中 $S = (M, R) \quad S'' = (M \{R(\text{esp}) \mapsto f, R(\text{esp}) - 4 \mapsto l_{\text{ret}}\}, R \{\text{esp} \mapsto R(\text{esp}) - 8\})$	(CALL)
$\frac{(p, q) = \Sigma(f) \quad f \neq \text{main} \quad \forall S_0, S. a S_0 S \rightarrow q S_0 S' \quad \forall S_0, S. a S_0 S \rightarrow (\forall m. m \geq S_0. R(\text{esp}) \rightarrow S_0, M(m) = S. M(m)) \wedge S_0. R(\text{esp}) = S. R(\text{esp}) \wedge S_0. R(\text{ebp}) = S. R(\text{ebp})}{\Psi(f); \Sigma \vdash \{a\} \text{ ret}}$	(RET)
其中 $S = (M, R) \quad S' = (M, R \{\text{esp} \mapsto S. R(\text{esp}) + 8\})$	

Fig. 5 Part of the inference rules.

图5 推理规则(部分)

规则 FBLK 是指若一个函数的每个指令块是合式的,则该函数所有指令块构成的函数体是合式的. 规则 FUNC 是指若函数体是合式的,函数前断言蕴涵函数入口块的前断言,并且出口块以 ret 指令结尾,则该函数是合式的(函数出口点状态满足函数后断言的要求体现在 ret 指令规则的前提中). 同样,规则 CDHP 是指若构成代码堆的每个函数是合式的,则代码堆是合式的.

合式指令块保证了在满足断言 a 的状态下执行指令块 I 是安全的. 对于指令块 c, I , 在满足 a 的状态下执行 c , 若得到的状态能使断言 a' 为真, 而且 a' 使得 I 为合式的, 那么 c, I 是合式指令块 (SEQ 规则). SEQ 规则的前提使用了操作语义函数 $\text{Next}_c(_)$, 这表明合式指令块的定义依赖于操作语义而不像 Hoare 逻辑那样使用代换. 一个直接跳转是合式的指令块 (JMP 规则), 仅当当前断言 a 蕴含跳转目的指令块对应的断言 a' . 其他跳转和分支指令的规则类似 JMP 规则, 不再一一列出. 因此, 可以看出要证明一个指令块的良好性, 需要应用相应的规则, 并且给出正确的中间断言.

CALL 和 RET 这两条规则反映了 Intel x86 机器上函数调用和返回之间的约定. 例如 RET 规则的前提要求在本函数的执行过程中, 除了当前活动记录外栈的其他地方没有修改, 并且当前活动记录中保存的上一个活动记录基地址和返回地址没有被修改.

4) 可靠性定理. 定义了推理规则之后, 要证明这些静态推理规则对 TM 操作语义的可靠性, 见定理 1. 它保证良形的程序可以在目标机器上一直安全执行.

定理 1. 可靠性(soundness). 如果 $\Psi; \Sigma \vdash \{a\} P$, 那么对任何自然数 n 存在程序 P' , 使得 $P \mapsto^n P'$. 简单地说, $\Psi; \Sigma \vdash \{a\} P$ 是合式的, 则 P 可以一直安全执行. TM 操作语义使用小步归约语义, 根据基于语法的方法 (syntactic approach)^[16] 证明该定理只需要先证明如下两个引理.

引理 1. 前进性(progress). 如果 $\Psi; \Sigma \vdash \{a\} P$, 那么存在 P' , 使得 $P \mapsto P'$.

引理 2. 保持性(preservation). 如果 $\Psi; \Sigma \vdash \{a\} P$ 并且存在 P' , 使得 $P \mapsto P'$, 那么存在 a' , 使得 $\Psi; \Sigma \vdash \{a'\} P'$.

该可靠性证明已经使用定理证明辅助工具 Coq 完成, 并且在该框架下阶乘函数部分正确性的证明也已经给出. 另外, 葛琳等人提出了为该汇编程序验证框架自动翻译生成形式规范的方法.

4 汇编语言指针逻辑

共享易变数据结构 (shared mutable data structure) 上带指针操作 (如链表、树等的操作) 的程序性质是验证的一个难点. 分离逻辑 (separation logic)^[17-18] 通过对 Hoare 逻辑的拓展来证明底层代码的这种性质. 但是分离逻辑并不能保证通过验证的程序没有内存泄露. Reynolds 提出一种精确断言 (precise assertion) 来解决这一问题, 但是没有一种语法的方法来判断一个断言是不是精确^[19].

因此, 在源语言级, 我们拓展 Hoare 逻辑为源语言 PointerC 设计了指针逻辑来验证指针程序的安全性. 指针逻辑可以保证通过验证的程序没有空指针或悬空指针引用、释放悬空指针和内存泄露等安全问题. 它本质上是一种精确指针分析工具, 可以用来从前向后收集程序点各个指针的信息. 这些信息包括指针是否为 NULL 指针、悬空指针还是有效指针 (有指向对象的指针) 的信息以及各有效指针之间

是否相等的信息(在推理规则中有效指针和 NULL 指针集合分别用 Π 和 \mathcal{N} 表示). 使用这些信息可以证明指针程序是否满足有关定型规则的附加条件, 支持同值有关的静态检查.

考虑到断言和证明的翻译问题, 汇编语言级没有采用分离逻辑作为指针程序验证逻辑, 因为从源语言级指针逻辑断言和证明翻译到类似的汇编级指针逻辑断言和证明要容易得多. 在汇编语言上, 我们设计与源语言级类似的指针逻辑, 用以在汇编语言级支持这种指针程序安全性的证明. 限于篇幅, 这里只介绍汇编语言指针逻辑同源语言级指针逻辑的区别和联系, 主要有以下 3 点:

1) 定型规则都并入指针逻辑相应的规则中, 没有一个单独的类型系统. 因为类型系统只关心指针的定型, 而指针逻辑也是有关指针的一些规则, 因此可以把它们合并在一起.

2) PointerC 赋值语句的规则略加修改就可作为汇编语言相应指令的规则, 但还需要为汇编语言特有的一些赋值增加规则.

3) PointerC 调用和返回语句的规则需要由汇编语言几条指令的规则来合成.

首先介绍 1) 区别和联系. 在源语言级, 通过类型系统可以判断一个语句是否涉及指针操作. 而在汇编级, 要知道一条指令是否涉及指针操作也需要设计一个简单的类型系统, 并使用类型断言来描述

$$\Gamma_c(p) = \text{Ptr } t \Gamma_c(q) = \text{Ptr } t p. eq \neq \emptyset q. eq \neq \emptyset,$$

$$p! = \frac{qno_leak(p) \Pi' = (\Pi - \{q. eq\}) / p \cup \{q. eq / p \cup \{p\}\}}{\{\Pi \wedge \mathcal{N} \wedge \Gamma_c\} \text{ movl } q, p \{\Pi' \wedge \mathcal{N} \setminus p \wedge \Gamma_c\}}$$

该规则只在源语言级相应规则^[10, 12]上增加了依据类型断言来判断该指令是否为指针赋值的一个前提(Γ_c 是类型断言). `pushl` 指令也是这种情况, 而且 `pushl` 操作后会使得 Π 和 \mathcal{N} 中以栈顶指针 `esp` 为访问路径起点的指针都需要调整其地址表示. 根据 PointerC 的赋值规则来增加适合这些情况的规则是直截了当的.

再来看 3) 区别和联系. 函数调用语句的实现需要汇编程序有调用指令序列和返回指令序列. 调用序列完成参数压栈、返回地址压栈、基地址压栈、设置新的基地址指针和分配局部变量空间等. PointerC 指针逻辑中函数调用语句规则所体现的指针信息变化要分解到调用序列和返回序列的各条指令中. 例如, 函数入口基本块的前 3 条指令是:

```
pushl %ebp /* 将调用者基地址保留到栈中 */
movl %esp, %ebp /* 建立本函数的基地址指
```

寄存器等的类型. 我们在程序规范中增加一个定型环境, 指明结构类型结构的大小(结构的大小用于防止动态申请空间的访问越界)及哪些偏移地址存放指针; 使用类型断言指出寄存器和活动记录存放数据的类型. 这样, 根据定型环境和类型断言可以判断当前指令是否为指针操作. 这种类型系统的设计比较简单, 限于篇幅, 这里不再介绍.

和源语言级一样, 如果类型系统判断一条指令为指针操作, 那么需要使用指针逻辑根据当前程序点的 Π 和 \mathcal{N} 来判断指针操作是否合法. 若合法, 根据指针逻辑的规则计算下一个程序点的 Π , \mathcal{N} 和类型断言; 若非法则验证失败.

关于 2) 区别和联系, 主要是因为汇编语言的独特指令. 除了函数调用语句, 包括 `malloc` 和 `free`, 其他能引起指针信息发生变化的主要是 `movl`, `pushl` 等指令, 在此也统称它们为赋值指令. 汇编级指针逻辑中有关这些指令的规则同源语言级指针逻辑的赋值规则几乎一样, 只要把那些规则中的变量看成赋值指令中可以出现的地址(内存地址、寄存器、寄存器变址等)即可. 由于汇编程序中不会有复杂的访问路径, 汇编级的别名演算要比源语言级简单得多.

例如, 对于 `movl q, p` 指令, p 和 q 是类型相等、值不相等的有效指针时, 汇编语言指针逻辑的推理规则是:

```
针 */
subl $N, %esp /* 分配局部变量空间, N 为常数 */
```

在第 2 条指令后, 需要把用 `esp` 表示的指针型参数改成用 `ebp` 表示. 在第 3 条指令后需要把指针型局部变量的地址加入悬空指针集合. 因此需要为这些指令设计相应的规则.

前面介绍了汇编级指针逻辑的推理规则, 而汇编程序验证框架之上增加这些指针逻辑规则后, 需要重新定义合式公式. 例如合式指令的形式如下:

$$\Psi(f); \Sigma; \Gamma \vdash \{a_p\} c \{a_q\},$$

其中 Γ 是定型环境, $a_p \equiv (\Pi \wedge \mathcal{N} \wedge \Gamma_c) \wedge a'_p$, Π 和 \mathcal{N} 是指针断言, Γ_c 是类型断言, a'_p 是其他断言. 同样, $a_q \equiv (\Pi' \wedge \mathcal{N}' \wedge \Gamma'_c) \wedge a'_q$. 根据 c 的语法形式, 我们给出指针集合的构造性规则. 对于合式指令块 SEQ 规则, 除了第 3 节要求外, 还需要从当前 Π , \mathcal{N} 和 Γ_c

用指针逻辑规则得到 c 之后程序点的 Π' , \mathcal{N}' 和 Γ'_c , 并且它可以使得后续指令块为合式的, 可以说 $c; l$ 是合式的. 类似地可以得到其他层次上程序结构的合式公式定义. 注意, 汇编级的其他断言都以状态为参数, 但是 Π , \mathcal{N} 和 Γ_c 不需要. 这是因为 Π 和 \mathcal{N} 只关心指针相等与否, Γ_c 记录指针的类型, 而不需要记录指针具体的值. 另外, 因状态变化而引起 Π , \mathcal{N} 和 Γ_c 的变化已经体现在指针逻辑的规则中.

为保证扩展的汇编程序验证框架的可靠性, 需要证明定理 2, 它保证了汇编级指针逻辑规则对操作语义可靠.

定理 2. 指针逻辑可靠性. 指针逻辑规则对 TM 的操作语义可靠.

汇编级指针逻辑的推理规则不是直接用操作语义来设计的, 它的证明同 Hoare 逻辑可靠性的证明方式一致. 该可靠性证明已经在 Coq 中完成, 详见文献[20].

5 实验结果

随着 PointerC 出具证明编译器的不断开发完善, 可以编译并生成证明的程序不断增加. 本节主要介绍部分实验结果, 通过这些数据可以看出代码和证明(包括部分人工交互完成的证明)的大小比例能够自动完成的证明所占总证明大小的比例以及编译(包括验证条件生成和证明生成)时间和证明检查时间的对比.

表 1 中列出了部分测试例的实验数据, 这些例子包括单链表逆转(reversal)、单链表节点插入(insert)、单链表生成(createList)和二叉树前序遍历(preorder). 所采用的实验平台为主频 2GHz 的 Intel 赛扬处理器、1GB 内存的计算机、内核版本为 2.6 的 Linux 的操作系统. 分析表明, 证明大小平均为汇编代码大小的 25 倍, 而 70% 的证明可以自动产生, 证明检查时间与编译时间相比可以忽略不计.

Table 1 The Experimental Results for Part of Our Test Cases

表 1 部分测试例的实验数据

Test Program	Reversal	Insert	CreateList	Preorder
Source Code Size (B)	734	653	504	431
Assembly Code Size (B)	1459	1688	1597	1097
Total Proof Size (KB)	37.1	41.4	39.3	28.2
Auto-Generated Proof Size (KB)	26.4	34.3	27.5	22.1
Percent of Ttotal Proof (%)	71	83	70	78
Compilation Time (ms)	1186	7402	10710	1870
Checking Time (ms)	20	22	21	13

6 相关工作

本文提出的汇编程序形式验证框架与 Shao 等人提出的 SCAP^[6]等都是使用 Hoare 风格的程序验证方法来开发携带证明的代码的通用框架. 不同之处在于我们使用的目标机器 TM 是针对 Intel x86 系列具体机器设计的, 接近实际机器. 考虑到基本块是代码生成和优化中一种重要的汇编程序结构, 在 TM 中我们采用基本块的划分法, 把条件转移和无条件转移指令都作为指令块的结尾, 而 SCAP 等没有把条件转移指令作为指令块的结尾. 另外, 我们按照 Intel x86 体系结构定义了 call 和 ret 指令, 而 SCAP 等所设计的 jal 指令适用于多种实际机器的函数调用方式.

另外, 我们的验证框架也因为设计思路的区别而有所不同. 首先, 我们采用两级层次的代码堆设计, 显式地以函数为单位进行程序验证, 这区别于 SCAP 使用规范隐式地把一组指令块组成一个函数的方法. 使用规范的隐式方法给规范的生成带来困难. 其次, 我们使用的断言与规范语言与 SCAP 显著不同. 对一个函数来说, 入口点要满足的断言一般只和入口点的状态有关, 出口点要满足的断言可能同入口点和出口点的状态都有关. 对于该函数中的任何一点来说, 其断言有两种表示方式 $\lambda S. S_o. g$ 和 $\lambda S_i. S. q$ (其中 S_i 和 S_o 分别代表入口点和出口点状态, S 表示该点的状态). $\lambda S. S_o. g$ 相当于从该点到出口点的程序段的语义, SCAP 使用 $\lambda S. S_o. g$ 方式来隐式地把一组指令块组成一个函数, 该方式源于 CPS (continuation passing style) 风格. 而在我们的汇编程序验证框架中, 我们对函数中的点采用 $\lambda S_i. S. q$ 方式, 它相当于从入口点到该点的程序段的语义. $\lambda S_i. S. q$ 方式便于表示在函数体执行过程中, 保存的返回地址和上一个活动记录的基地址没有被修改, 其他活动记录的内容都没有被修改等安全性质. 另外, 采用 $\lambda S_i. S. q$ 方式, 每个指令块只需要有一个前断言, 这是因为可以使用后继指令块的前条件来进行推理. 而采用 $\lambda S. S_o. g$ 方式时, 每个指令块需要 $\lambda S. p$ 和 $\lambda S. S_o. g$ 两个断言, 其根本原因是指令块和函数的合式定义都是基于正向的控制流, 因此每个指令块需要前断言 $\lambda S. p$ 表示执行该基本块时状态应该满足的条件.

我们设计的指针逻辑和 Reynolds 等人提出的分离逻辑都是通过对 Hoare 逻辑的拓展来证明在共享易变数据结构上带指针操作的程序的性质, 但是它们在设计理念上有较大的区别. 指针逻辑关心

别名,它的出发点是所有不同的访问路径都被认为绑定到不同的地址,除非能证明它们绑定到了相同的地址.因此指针逻辑需要收集有效指针相等信息,以便从中推导出哪些不同的访问路径绑定到了相同的地址.这是指针逻辑不用引入新连接词的原因.分离逻辑面向只有简单访问路径的编程语言,因此它关心指针的指向,它的出发点是所有指针都有可能指向同一个对象,除非它们被显式区分为指向不同的对象,因此分离逻辑需要使用“分离合取”等连接词.关于指针逻辑和分离逻辑关系,我们证明了指针逻辑表达能力不弱于分离逻辑^[21].另外,使用分离逻辑证明汇编程序的部分正确性,通常需要程序员给出40~100倍于程序代码长度的证明^[22],显然这种方法不适合实际应用.而我们设计的汇编语言指针逻辑主要用于证明指针程序的安全性质,如没有空指针引用等等.它是面向自动定理证明工具,不需要程序员干预证明过程,具有一定的实用性.在源语言和汇编语言上都使用指针逻辑有利于断言、规范和证明的翻译.

与TAL相比,我们的类型系统简单得多,最主要的区别是我们没有给保存在活动记录中的基地址和返回地址定型.因为我们用逻辑断言保证了它们没有被修改,因此可以不关心它们的类型.

7 总 结

本文介绍了我们为PointerC的出具证明编译器所设计的汇编程序形式验证框架.该框架基于Intel x86目标机器模型TM,支持Hoare风格的推理.通过出具证明编译器的一个原型,我们在该汇编程序验证框架上生成了有关单链表和二叉树的一些函数的携带证明的代码.我们使用Coq来完成证明的检查.目前在研究指针逻辑的自动定理证明过程中,我们正在设计一个证明检查器来取代Coq,而使得所携带的证明简洁.在下一步工作中,还将考虑在PointerC中允许用户自定义安全策略,并放宽对指针运算的限制,有限制地允许指针算术,以适应编程中经常使用的calloc存储分配.而汇编程序验证框架的下一步研究工作将集中在汇编语言级来支持这些新的特性.

有关目标机器TM和汇编程序形式验证框架的设计、PointerC及其类型系统和指针逻辑系统的设计、证明实例、编译器原型和证明检验器原型等都展示在我们项目网站<http://sbg.ustcsz.edu.cn/lss/>上.

参 考 文 献

- [1] G Morrisett, D Walker, K Crary, *et al.* From system F to typed assembly language [C]. The 25th ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages, New York, 1998
- [2] Y Mandelbaum, D Walker, R Harper. An effective theory of type refinements [C]. The 8th Int'l Conf on Functional Programming, Uppsala, Sweden, 2003
- [3] G Necula. Proof-carrying code [C]. The 24th ACM Symp on Principles of Programming Language, Paris, France, 1997
- [4] A W Appel. Foundational proof-carrying code [C]. The 16th Annual IEEE Symp on Logic in Computer Science, Boston, 2001
- [5] H Xi. Applied type system (extended abstract)[C]. In: Proc of Post-Workshop Proceedings of TYPES 2003, LNCS 3085. Berlin: Springer-Verlag, 2004. 394-408
- [6] X Feng, Z Shao, A Vaynberg, *et al.* Modular verification of assembly code with stack-based control abstractions [C]. In: Proc of the 2006 ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM Press, 2006. 401-414
- [7] Z Z Ni, Z Shao. Certified assembly programming with embedded code pointers [C]. The 33rd ACM Symp on Principles of Programming Languages (POPL'06), Charleston, South Carolina, 2006
- [8] X Y Feng, Z Z Ni, Z Shao, *et al.* An open framework for foundational proof-carrying code [C]. 2007 ACM SIGPLAN Int'l Workshop on Types in Language Design and Implementation (TLDI'07), Nice, France, 2007
- [9] Chen Yiyun, Ge Lin, Hua Baojian, *et al.* Design of a certifying compiler supporting proof of program safety [C]. The 1st IEEE & IFIP Int'l Symp on Theoretical Aspects of Software Engineering (TASE 2007), Shanghai, 2007
- [10] Chen Yiyun, Ge Lin, Hua Baojian, *et al.* A pointer logic and certifying compiler [J]. Frontiers of Computer Science in China, 2007, 1(3): 297-312
- [11] Hua Baojian, Ge Lin. The definition of PointerC programming language [OL]. <http://sbg.ustcsz.edu.cn/lss/doc/index.html>, 2006
- [12] Chen Yinyun, Hua Baojian, Ge Lin, *et al.* A pointer logic for safety and security verification of pointer programs [J]. Chinese Journal of Computers, 2008, 31(3): 372-380 (in Chinese)
(陈意云, 华保健, 葛琳, 等. 一种用于指针程序安全性证明的指针逻辑. 计算机学报, 2008, 31(3): 372-380)
- [13] C Paulin-Mohring. Inductive definitions in the system Coq-rules and properties [C]. Typed Lambda Calculus and Application (TLCA 1993), Utrecht, The Netherlands, 1993
- [14] The Coq Development Team. The Coq proof assistant reference manual ver8.0 [OL]. <http://pauillac.inria.fr/coq/doc/>, 2004

- [15] Li Zhaopeng. An implementation of FCAP using the Coq proof assistant [OL]. <http://zero.ustcsz.edu.cn/svn/projectzero/FCAP/>, 2006
- [16] A K Wright, M Felleisen. A syntactic approach to type soundness [J]. *Information and Computation*, 1994, 115(1): 38-94
- [17] J C Reynolds. Separation logic: A logic for shared mutable data structures [C]. *The 17th Annual IEEE Symp on Logic in Computer Science*, Copenhagen, Denmark, 2002
- [18] M Parkinson, G Bierman. Separation logic and abstraction [C]. *The 32th ACM Symp on Principles of Programming Language*, Long Beach, California, USA, 2005
- [19] J C Reynolds. An introduction to separation logic [OL]. <http://www.cs.cmu.edu/jcr/>, 2005
- [20] Li Zhaopeng, Wang Wei, Tian Bo. A proof of soundness of assembly pointer logic implemented using the coq proof assistant [OL]. <http://zero.ustcsz.edu.cn/svn/projectzero/APL/>, 2006
- [21] Li Zhaopeng, Chen Yiyun. The relations between separation logic and the pointer logic [OL]. <http://ssg.ustcsz.edu.cn/lss/doc/>, 2007
- [22] Xiang Sen, Chen Yiyun, Lin Chunxiao, *et al.* Safety verification of dynamic storage management in Coq [J]. *Journal of Computer Research and Development*, 2007, 44(2): 361-367 (in Chinese)
(项森, 陈意云, 林春晓, 等. 动态存储管理安全验证的 Coq 实现[J]. *计算机研究与发展*, 2007, 44(2): 361-367)



Li Zhaopeng, born in 1978. Received his B. S. degree in computer science from University of Science and Technology of China, Hefei, China, in 2003. Since 2003, he has been a Ph. D. candidate in computer science from University of

Science and Technology of China, Hefei, China. His current research interests include program verification, software safety and security, type system and theory.

李兆鹏, 1978 年生, 博士研究生, 主要研究方向为程序验证、软件安全、类型系统和理论。



Chen Yiyun, born in 1946. Professor and Ph. D. supervisor in Dept. of Comput. Sci. & Technol. at Univ. of Sci. & Technol. of China. He received his M. S. degree from East-China Institute of

Comput. Technol. in 1982. His main research interests include applications of logic (including formal semantics and type theory), techniques for designing and implementing programming languages and software safety and security.

陈意云, 1946 年生, 教授, 博士生导师, 主要研究方向为程序设计语言的理论和实现技术、形式描述技术、软件安全。



Ge Lin, born in 1979. Received her Ph. D degree in computer science from University of Science and Technology of China, Hefei, China, in 2007. Her current research interests include program

verification, software safety and security, type system and theory.

葛琳, 1979 年生, 博士, 主要研究方向为程序验证、软件安全、类型系统和理论。



Hua Baojian, born in 1979. Received his B. S. degree in computer science from Xi'an Institute of Post and Telecommunication, Shanxi, China, in 2002. Since 2002, he has been a Ph. D.

candidate in computer science from University of Science and Technology of China, Hefei, China. His current research interests include program verification, type system, software safety and security.

华保健, 1979 年生, 博士研究生, 主要研究方向为程序验证、类型系统和软件安全。

Research Background

Proof-carrying code brings two grand challenges to the research field of programming languages. One is to study the technology of certifying compilation. The other is to seek more expressive program logics or type systems to specify or reason about the properties of high-level or low-level programs. And safety is an important issue among the properties of high-assurance software. The verification method for software to meet its safety policies is one of the hot researches. In terms of our framework to design and verification of safety programs, and the pointer logic proof system, this paper introduces our research on the formal description of target machine, the formal certifying framework for assembly programs and property proof of assembly pointer programs. We have formalized the whole work in the proof assistant Coq. Our work is supported by the National Natural Science Foundation of China (60473068 and 60673126) and Intel China Research Center (Beijing).