ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.21, No.2, February 2010, pp.305−317
doi: 10.3724/SP.J.1001.2010.03794

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

字节码虚拟机的构造和验证*

董 渊+, 任 恺, 王生原, 张素琴

(清华大学 计算机科学与技术系,北京 100084)

Construction and Certification of a Bytecode Virtual Machine

DONG Yuan+, REN Kai, WANG Sheng-Yuan, ZHANG Su-Qin

(Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

+ Corresponding author: E-mail: dongyuan@tsinghua.edu.cn

Dong Y, Ren K, Wang SY, Zhang SQ. Construction and certification of a bytecode virtual machine. *Journal of Software*, 2010,21(2):305−317. http://www.jos.org.cn/1000-9825/3794.htm

Abstract: This paper presents a method to build and verify bytecode virtual machine. The formal definition and the operational semantics of a bytecode virtual machine (BVM) are given. CertVM (certified virtual machine) is implemented with X86 assembly code. It is proved in this paper that the CertVM is satisfied with the formal definition of the bytecode machine with simulation relation. The virtual machine implementation program is certified in the Coq proof assistant. The proof is machine checkable. This method guarantees that a certified bytecode program will run on the certified virtual machine without stuck unless hardware faults. This work does not only provide a solid theoretical foundation for reasoning about virtual machine, but also makes an important advance toward building the trustworthy software.

Key words: certified VM; modular certification; bytecode; Hoare-style logic

摘 要: 提出一种虚拟机构造和验证方案.给出字节码程序运行环境 BVM(bytecode virtual machine)的形式化定义;采用 X86 机器语言构造虚拟机 CertVM(certified virtual machine);并证明该虚拟机实现符合相应程序规范并和 BVM 之间具有模拟关系.利用辅助工具 Coq 给出证明,所有证明均可机器自动检查.CertVM 确保在硬件环境满足其语义规范的情况下,已验证的字节码程序能够在给定虚拟机环境中正常运行.给出的方案不仅为虚拟机验证提供理论基础,而且为可信软件构造提供了一种有益的尝试.

关键词: 已验证虚拟机;模块化验证;字节码;类 Hoare 逻辑

中图法分类号: TP316 **文献标识码**: A

互联网和智能终端设备日益深入地渗透到日常生产和生活中的各个环节,如何验证这样的软件,确保程序正确、顺利地执行,成为大家普遍关注和深入研究的一个热点问题[1].字节码(bytecode)及其虚拟机(virtual machine)运行环境是其中最为重要的技术,很多现有研究工作集中在字节码程序验证上.但是,虚拟机实现的可

* Supported by the National Natural Science Foundation of China under Grant Nos.90818019, 90816006 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant Nos.2008AA01Z102, 2009AA011902 (国家高技术研究发展计划(863))

Received 2009-06-15; Revised 2009-09-11; Accepted 2009-12-07

信问题也同样重要,因为即便是已验证字节码程序,一旦虚拟机出错依然无法正确运行.而现实中虚拟机通常采用C/C++等语言实现,其中的缺陷屡见不鲜[2].本文深入探讨字节码虚拟机的可信问题,给出一种可信虚拟机构造和验证方案,为可信软件运行环境的构造提供理论和技术支持.

1 虚拟机验证的价值

本文研究字节码程序运行环境——语言级虚拟机的验证问题.这类虚拟机通常解释执行以JAVA字节码(Java bytecode)[3]和微软.NET CIL[4]为代表的字节码程序,是目前大多数网络应用程序的运行环境.

随着程序形式化验证研究的深入开展,字节码程序验证成为一个得到广泛关注的话题.从高级语言[5]和汇编语言[6−9,10]两个方面的验证研究发展趋势来看,都需要在中间表示层面进行深入、细致的研究工作,而字节码是最佳选择之一.字节码验证可以大大提高相关软件的可信程度,具有相当大的实用价值.将验证语言提高到字节码将带来验证效率方面的显著提升,同时可以为未来从高级语言到汇编语言的证明保持编译器构造提供中间语言的支持,为可信软件构造提供有力的理论和工具支持.

最初,针对JAVA字节码验证的主要研究工作集中于JVM内部检查器,目标是类型正确性,从而保证存储安全性(verifier)[11].随着研究工作的深入,人们逐步认识到仅有类型安全检查远远不够,还需要进一步研究其他更多程序的性质.近年来,人们提出大量用于字节码程序验证的逻辑系统,代表性研究工作包括:Quigley设计了一个用于字节码程序的类Hoare逻辑系统,证明了含有循环的程序片段[12];MRG项目主要着眼于程序资源,提出一种针对字节码的具有资源感知特征的操作语义,以及相应的逻辑系统[13];Bannwart和Muller提出一种支持对象、引用、方法和继承等面向对象特征的逻辑系统,用来证明类JAVA字节码[14];Benton提出一种组合逻辑系统,用于一种.NET CIL的命令式语言子集,并给出纯手工证明[15].特别值得一提的是字节码建模语言BML (bytecode modeling language)的研究工作[16].该语言作为一种可理解的字节码程序规范,应用开发人员可用它在字节码层面书写程序标注,以规范字节码程序的行为功能;与之对应,JML语言提供JAVA源语言层面的规范描述.该研究同时包含一个并不完整的JML到BML编译器,其局限性在于,证明检查仍然需要借助一个复杂的验证器.此前我们提出一系列针对字节码程序的验证方法,实现了程序的模块化验证、对嵌套程序的支持[17]等.

虽然上述工作取得了大量成果,但是这些研究工作通常都没有考虑虚拟机运行环境实现本身的可信问题,所以即便是验证过的字节码程序,一旦虚拟机出错都将不能正确运行.而现实中虚拟机实现中的错误屡见不鲜,因此,开展可信虚拟机构造研究是一个非常迫切的需求.

本文深入探讨虚拟机本身的验证问题,基于FPCC方法[18],采用汇编代码构造出一个可真实运行于Bochs模拟器[19]的虚拟机系统原型,使用证明辅助工具Coq[20]给出该虚拟机符合规范的证明.所有证明均可以自动检查[21],为可信虚拟机的构造问题提供了一种良好的解决思路.

2 字节码虚拟机 BVM(bytecode virtural machine)

本节定义字节码虚拟机 BVM,支持类似于 JAVA 字节码和.NET CIL 的字节码子集 BC/0(bytecode zero).图 1 给出该机器可运行的程序实例.本节给出虚拟机定义、BC/0 指令操作语义.

```
;int factor(){r=1; while (n!=0){r=r*n; n=n−1;}}
;method factor: Factorial, while loop with specification
−{(p0,g0)}      ;I1 (instruction sequence 1), entry point
0 pushc 1       ;push immediate data 1          |8 pushc 1       ;push immediate data 1
1 pop r         ;r=1                            |9 binop         ;n−1
2 goto 11       ;jump to the end of while loop  |10 pop n        ;save variable n
−{(p3,g3)}      ;I2, loop start here            |−{(p11,g11)}    ;I3
3 pushv r       ;push variable r                |11 pushv n      ;push var n
4 pushv n       ;push variable n                |12 pushc 0      ;push imm 0
5 binop*        ;r*n                            |13 binop#       ;n#0?
6 pop r         ;save variable r                |14 brture  3    ;conditional goto
7 pushv n       ;push variable n                |15 ret          ;function ret
```

Fig.1 Stack-Based bytecode program with source code (function factor)

图 1 基于栈的字节码程序及其源代码(函数 factor)

2.1 虚拟机BVM的定义

图 2 给出了虚拟机的定义.BVM 采用类似于 JAVA 虚拟机的双栈结构.整个机器配置(machine configuration)称为“世界”$\mathbb{W}$(world),包含只读的代码堆$\mathbb{C}$(code heap)、可修改的状态$\mathbb{S}$(state)、函数调用栈$\mathbb{K}c$(call stack)和程序计数器 pc(program counter).代码堆是代码标号 f(labels)到指令序列$\mathbb{I}$(instruction sequences)的部分映射,状态$\mathbb{S}$包含内存堆$\mathbb{H}$(memory heap)和计算栈$\mathbb{K}$(evaluation stack),函数调用栈$\mathbb{K}c$(call stack)存放函数调用的返回地址,程序计数器 pc 指向代码堆中的当前指令.

(World)	$\mathbb{W}::=(\mathbb{C},\mathbb{S},\mathbb{K}c,pc)$	(State)	$\mathbb{S}::=(\mathbb{H},\mathbb{K})$
(CodeHeap)	$\mathbb{C}::=\{f\to\mathbb{I}\}^*$	(ProgCounter)	$pc::=n$
(CStack)	$\mathbb{K}c::=nil\|f::\mathbb{K}c$	(EStack)	$\mathbb{K}::=nil\|w::\mathbb{K}$
(Memory)	$\mathbb{H}::=\{k\to w\}^*$	(Word)	$w::=i$(integers)
(Labels)	$f,k::=n$(nat nums)	(OprNum)	$m::=\{+,\ldots/,\ldots,\ldots+\}$
(Command)	$c::=\iota\|ret\|$goto f	(InstrSeq)	$\mathbb{I}::=\iota;\mathbb{I}\|ret\|$goto f
(Instr)	$\iota::=$pushc w|pushv k|pop k|binop m|unop m|btrue f |call f		

Fig.2 Definition of BVM

图 2 BVM 定义

指令序列$\mathbb{I}$定义为由跳转和返回指令结尾的一系列指令组成的片段,$\mathbb{C}[f]$表示$\mathbb{C}$中由 f 开始的一个指令序列.用“点”来表示多元组中的组成部分,如$\mathbb{S}.\mathbb{K}$表示状态$\mathbb{S}$中的栈$\mathbb{K}$.$(F\{a\to b\})(x)$表示对映射 F 赋值,使得 $F(a)=b$,所有其余映射关系保持不变.函数 $length()$和 max()用来获取栈$\mathbb{K}$,$\mathbb{K}c$ 的有效长度和最大长度,$valid$ 表示有足够计算栈和函数调用栈空间,$validRa$ 表示函数调用栈中保存了合法的返回地址.

$$\mathbb{C}[f]\triangleq\begin{cases}c, & c=\mathbb{C}[f] \text{ and } c=\text{goto } f' \text{ or } ret\\ \iota;\mathbb{I}, & \iota=\mathbb{C}[f] \text{ and } \mathbb{I}=\mathbb{C}[f+1]\end{cases}\qquad (F\{a\to b\})(x)\triangleq\begin{cases}b, & \text{if } x=a\\ F(x), & \text{otherwise}\end{cases}$$

$$validK\ n\ \mathbb{K}\triangleq length(\mathbb{K})+n\le\max(\mathbb{K})\qquad validK\ n\ \mathbb{K}c\triangleq length(\mathbb{K}c)+n\le\max(\mathbb{K}c)$$

$$validRa\ \mathbb{K}c\triangleq\exists f,\exists\mathbb{K}c',\mathbb{K}c=f::\mathbb{K}c'$$

Fig.3 Definition of representation

图 3 表示符号定义

2.2 指令操作语义

图 4 定义指令的操作语义,这里,$Enable(c,\mathbb{K}c,\mathbb{S})$是每一条指令$c$可以执行的最弱前条件,$NextKc(c,pc,\mathbb{S})$关系

定义pc所指指令执行后的函数调用栈$\mathbb{K}c$的变化,$NextS(c,pc,\mathbb{K}c)$关系定义pc所指指令执行后的状态$\mathbb{S}$的变化,$NextPC(c,\mathbb{S},\mathbb{K}c)$表示状态$\mathbb{S}$、调用栈$\mathbb{K}c$时$c$指令执行导致的$pc$变化.程序执行通过机器配置$\mathbb{W}$的逐步转化来刻画,即$\mathbb{W}\rightarrow\mathbb{W}'$是通过$pc$所指指令的执行而实现.

$NextS(c,pc,\mathbb{K}c)\ \mathbb{S}'\ \mathbb{S}''$ where $\mathbb{S}'=(\mathbb{H},\mathbb{K})$

if c=	if $Enable(c,\mathbb{K}c,\mathbb{S})$=	then $\mathbb{S}''$=
pushc w	$validK\ 0\ \mathbb{K}$	$(\mathbb{H},w::\mathbb{K})$
pushv k	$validK\ 0\ \mathbb{K}$ and $\mathbb{H}(k)=w$	$(\mathbb{H},w::\mathbb{K})$
pop k	$\mathbb{K}=w::\mathbb{K}'$	$(\mathbb{H}\{k\rightarrow w\},\mathbb{K}')$
binop bop	$\mathbb{K}=w_1::w_2::\mathbb{K}'$, $w=bop(w_1,w_2)$	$(\mathbb{H},w::\mathbb{K}')$
unop uop	$\mathbb{K}=w_1::\mathbb{K}'$, $w=uop(w_1)$	$(\mathbb{H},w::\mathbb{K}')$
brtrue f	$\mathbb{K}=w::\mathbb{K}'$, w=True or False	$(\mathbb{H},\mathbb{K}')$
call f	$validKc\ 1\ \mathbb{K}c$	$(\mathbb{H},\mathbb{K})$
ret	$validRa\ \mathbb{K}c$	$(\mathbb{H},\mathbb{K})$
…	…	$(\mathbb{H},\mathbb{K})$

$NextKc(c,pc,\mathbb{S}')\ \mathbb{K}c\ \mathbb{K}c'$ where $\mathbb{S}=(\mathbb{H},\mathbb{K})$

if c=	if $Enable(c,\mathbb{K}c,\mathbb{S})$=	then $\mathbb{K}c'$=
call f	$validKc\ 1\ \mathbb{K}c$	$(pc+1)::\mathbb{K}c$
ret	$validRa\ \mathbb{K}c$	$\mathbb{K}c'$
…	…	$\mathbb{K}c$

$NextPC(c,\mathbb{S},\mathbb{K}c)\ pc\ pc'$ where $\mathbb{S}=(\mathbb{H},\mathbb{K})$

if c=	if $Enable(c,\mathbb{K}c,\mathbb{S})$=	then pc'=
brtrue f	$\mathbb{K}=w::\mathbb{K}'$ w=True	f
	$\mathbb{K}=w::\mathbb{K}'$ w=False	$pc+1$
call f	$validKc\ 1\ \mathbb{K}c$	f
ret	$validRa\ \mathbb{K}c\wedge\mathbb{K}c=f::\mathbb{K}c'$	f
goto f		f
…	…	$pc+1$

$$\frac{c=\mathbb{C}(pc)\ Enable(c,\mathbb{K}c,\mathbb{S})\ NextS(c,pc,\mathbb{K}c)\mathbb{S}\ \mathbb{S}'\ NextKc(c,pc,\mathbb{S})\mathbb{K}c\ \mathbb{K}c'\ NextPC(c,\mathbb{S},\mathbb{K}c)\ pc\ pc'}{(\mathbb{C},\mathbb{S},\mathbb{K}c,pc)\rightarrow(\mathbb{C},\mathbb{S}',\mathbb{K}c',pc')}(pc)$$

Fig.4 Operational semantics of BVM

图 4 BVM 机器操作语义

3 虚拟机构造和验证

本文主要工作是采用形式化验证的软件构造方案,给出一种CertVM虚拟机的实现和证明.CertVM采用X86汇编代码实现,可运行于Bochs模拟器,采用SCAP逻辑系统[6]证明该虚拟机符合上一节所给出的语义,证明实现代码则使用Coq完成.本节首先给出CertVM虚拟机的实现,接着讨论运行环境——X86 实模式的形式化定义,然后简单介绍证明CertVM所采用的逻辑系统,最后着重讨论如何证明模拟关系.

3.1 虚拟机的构造

我们采用 X86 实模式汇编实现了虚拟机 CertVM 字节码解释执行和加载等通常语言级虚拟机最为核心的功能部件,暂未考虑垃圾回收、即时编译等运行时优化功能,也不考虑检查器.

内存分配:字节码虚拟机 CertVM 的只读代码堆($\mathbb{C}$)、内存堆($\mathbb{H}$)、计算栈($\mathbb{K}$)和函数调用栈($\mathbb{K}c$)在 X86 汇编实现中均采用数组表示,即内存中一段连续的空间.我们分别用符号 $M_{\mathbb{C}}$,$M_{\mathbb{H}}$,$M_{\mathbb{K}}$ 和 $M_{\mathbb{K}c}$ 来表示这些数组.我们定义函数 *base*()和 max()来描述这些数组基地址和数组上界.定义 *top*()函数来获取 $\mathbb{K}$ 和 $\mathbb{K}c$ 的栈顶指针(*sp*,*csp*)的位置.栈顶指针(*sp*,*csp*)和程序计数器(*pc*)等在 X86 汇编中实现为变量.

字节码加载:字节码加载器(loader)是用来加载一段字节码程序的.它进行如下一系列的操作:把字节码程序加载到只读代码堆 $M_{\mathbb{C}}$;初始化内存堆 $M_{\mathbb{H}}$(即清零),将计算栈 $M_{\mathbb{K}}$ 清空(栈指针置底);将当前程序作为顶层函数,其返回值的地址设为−1(0*xFF*),因此函数调用栈 $M_{\mathbb{K}c}$ 被清空,压入−1 到栈底;最后将 *pc* 指向字节码程序的入口处,程序转入取值执行阶段.

执行机制:字节码程序的每条指令都是通过一系列的 X86 汇编代码模拟执行.每条指令的模拟执行都需要经过 4 个阶段:取指、译码、分配和解释运行.虚拟机在取指阶段主要利用 *pc* 值在 $M_{\mathbb{C}}$ 得到当前待模拟的字节码指令.在译码阶段,虚拟机分析并判断待模拟指令的类型.然后在分配阶段,通过查询分配表跳转到该指令类型对应的解释代码.最后在解释运行阶段,完成这条指令的解释运行.需要注意的是,对于不同的字节码指令,其取值、译码和分配阶段相应的 X86 代码完全相同,只有在解释运行阶段的 X86 代码才有区别.图 5 给出 goto 指令相应的 X86 解释代码.其中,标号 fetch,decode,dispatch 和 goto 分别对应了字节码 goto 指令的取指、译码、分配和解释运行 4 阶段的 X86 实现.

```
;Part of the implementation of BVM
−{(p_fetch,g_fetch)}          ;bytecode fetch
1 fetch:
2     movw(pc), %ax           ;bytecode program counter
3     cmpw $0xFF, %ax         ;ra of top level function
4     je fetch                ;loop after top function
5 decode:                     ;bytecode decode
6     movw $code, %bx         ;bytecode base address
7     addw %ax, %bx           ;current bytecode address
8     movw (%bx), %ax         ;bytecode fetch i.f
−{(p_dispatch,g_dispatch)}
9 dispatch:                   ;push variable n
10    movw $table, %bx        ;base of dispatch table
11    addw %ax, %bx           ;offset for current bytecode
12    addw %ax, %bx           ;entry point is 2 word long
13    movw (%bx), %ax         ;get entry point
14    jmpw *%ax               ;jump to entry point
−{(p_goto,g_goto)}
15 goto:                      ;bytecode execution
16    movw(pc), %ax           ;code point
17    movw $code, %bx         ;bytecode base address
18    movw 2(%bx), %cx        ;operand fetch i.a
19    addw %cx, %cx           ;each instruction is 4bytes
20    addw %cx, %cx
21    jmp fetch
```

Fig.5 Fragment of CertVM implementation, instruction goto

图 5 CertVM 实现代码片段,goto 指令

3.2 X86机器定义和SCAP逻辑系统

本文采用 SCAP 逻辑系统来证明字节码虚拟机 CertVM 的汇编代码实现,本节给出 SCAP 系统的简单介绍.该系统采用(*p*,*g*)规范描述函数功能,支持 X86 汇编代码的模块化验证,具有很强的表达能力.这里简单给出 SCAP 系统相关的 X86 机器定义、指令操作语义、推理规则和合理性证明.SCAP 系统给出一种形式化的保证,硬件满足其语义规范的情况下,通过证明的程序将不会进入滞留状态,而且符合其相应的程序规范.

3.2.1 X86 机器定义

图 6 给出了对 X86 机器状态的形式化定义.类似于 BVM,整个机器配置(machine configuration)也称为“世界”$\mathbb{W}$(world).不同的是,世界中主要包含只读的代码堆 $\mathbb{C}$(code heap)、可修改的状态 $\mathbb{S}$(state)和程序计数器 *pc*(program counter).其中,可修改的状态 $\mathbb{S}$ 由内存堆 $\mathbb{H}$(memory heap)、通用寄存器 $\mathbb{R}$(general register)和标志寄存器 *zf*(flag register)组成.此定义对 X86 进行了简化,只包含虚拟机代码中使用的 4 个通用寄存器.

类似于 BVM,图 7 定义了 X86 指令的操作语义.*NextS*(*c*,*pc*)关系定义 *pc* 所指指令执行后的状态变化,*NextPC*(*c*,$\mathbb{S}$)表示 *c* 指令执行导致的 *pc* 变化.其中,*eval*(*o*)表示对操作数 *o* 求值.程序执行通过机器配置 $\mathbb{W}$ 的逐步转化来刻画,即 $\mathbb{W}\rightarrow\mathbb{W}'$ 是通过 *pc* 所指指令的执行而实现.

(World)	$\mathbb{W} ::= (\mathbb{C}, \mathbb{S}, pc)$		(Flag)	zf	::= false | true
(CodeHeap)	$\mathbb{C} ::= \{f \to \mathbb{I}\}^*$		(Label)	l, f, pc	::= n (nat nums)
(State)	$\mathbb{S} ::= \{\mathbb{H}, \mathbb{R}, zf\}$		(Word)	w	::= i (integers)
(Heap)	$\mathbb{H} ::= \{l \to w\}^*$		(Register)	r	::= $ax \mid bx \mid cx \mid dx$
(RegFile)	$\mathbb{R} ::= \{r \to w\}^*$		(Address)	a	::= $l \mid l(r)$
(Command)	$c ::= \iota \mid$ jmp $f \mid$ jmpw r		(Operator)	o	::= $w \mid r$
(InstrSeq)	$\mathbb{I} ::= \iota; \mathbb{I} \mid$ jmp $f \mid$ jmpw r				
(Instr)	$\iota ::=$ mov $o, r \mid$ ld $a, r \mid$ st $o, a \mid$ cmp $o, r \mid$ add $o, r \mid$ sub $o, r \mid$ je f				

Fig.6 Definition of X86 machine

图 6 X86 机器的定义

NextS(c,pc) $\mathbb{S}\ \mathbb{S}''$ where $\mathbb{S}=(\mathbb{H},\mathbb{R},zf)$

if c=	if *Enable*(c,$\mathbb{S}$,pc)=	then $\mathbb{S}''$=
mov o, r	w=*eval*(o)	($\mathbb{H}$,$\mathbb{R}${$r\to w$},zf)
ld a, r	l=*eval*(a)$\wedge l\in$*dom*($\mathbb{H}$)$\wedge\mathbb{H}[l]=w$	($\mathbb{H}$,$\mathbb{R}${$r\to w$},zf)
st o, a	l=*eval*(a)$\wedge l\in$*dom*($\mathbb{H}$)$\wedge w$=*eval*(o)	($\mathbb{H}${$l\to w$},$\mathbb{R}$,zf)
cmp o, r	w=*eval*(o)	($\mathbb{H}$,$\mathbb{R}$,true) if $w=\mathbb{R}(r)$; ($\mathbb{H}$,$\mathbb{R}$,false) others
add o, r	$w=\mathbb{R}(r)$+*eval*(o)	($\mathbb{H}$,$\mathbb{R}${$r\to w$},zf)
sub o, r	$w=\mathbb{R}(r)-$*eval*(o)	($\mathbb{H}$,$\mathbb{R}${$r\to w$},zf)
…		($\mathbb{H}$,$\mathbb{R}$,zf)

NextPC(c,$\mathbb{S}$) pc pc' where $\mathbb{S}=(\mathbb{H},\mathbb{R},zf)$

if c=	then pc'=
je f	f, if zf =true; pc+1, others
jmp f	f
jmpw r	$\mathbb{R}(r)$
…	pc+1

$$\frac{c=\mathbb{C}(pc)\ \textit{Enable}(c,\mathbb{S},pc)\ \textit{NextS}(c,pc)\ \mathbb{S}\ \mathbb{S}'\ \textit{NextPC}(c,\mathbb{S})\ pc\ pc'}{(\mathbb{C},\mathbb{S},pc)\to(\mathbb{C},\mathbb{S}',pc')}(pc)$$

Fig.7 Operational semantics of X86 machine

图 7 X86 机器操作语义定义

3.2.2 SCAP 规范介绍

SCAP 直接使用 Coq 证明辅助工具的内嵌逻辑作为程序规范书写语言.该逻辑是一种使用归纳定义的高阶谓词逻辑.SCAP 系统是基于程序规范(program specification,又称断言)进行推理的.程序员在每个指令序列开始处插入断言(程序规范)s 来规定程序执行需要满足的条件.SCAP 程序规范是谓词二元组(p,g),插入断言之后的代码如图 5 所示,图中的大括号即给出程序规范.谓词 p 描述程序入口状态$\mathbb{S}$的性质,谓词 g 描述两个程序状态之间的关系,Coq 中的 p,g 均定义为返回值为命题的函数,分别以$\mathbb{S}$和两个$\mathbb{S}$为参数.我们使用 p 来描述当前状态的前条件,使用 g 来描述程序当前点与函数返回点(出口)之间的状态变化(即程序行为).图 8 给出 SCAP 规范的形式定义.图 7 中 *Enable*(c,$\mathbb{S}$,pc)就是一个谓词 p,而 *NextS*(c,pc)就是一个谓词 g.

(Pred)	$p \in State \to Prop$	(Guarantee)	$g \in State \to State \to Prop$
(Spec)	$s ::= (p, g)$	(MPred)	$m \in Memory \to Prop$
(CdHpSpec)	$\Psi ::= \{(f_1, s_1), ..., (f_n, s_n)\}$		

Fig.8 Specification constructors for SCAP

图 8 SCAP 的规范构造符

同时,我们定义基于 SCAP 规范的一系列运算,如图 9 所示.这些运算能够从简单的程序规范出发,构造更复杂、语义更丰富的程序规范.

$$p \Rightarrow g \triangleq \forall \mathbb{S}\ p\ \mathbb{S} \to \exists \mathbb{S}', g\ \mathbb{S}\ \mathbb{S}' \qquad p \triangleright g \triangleq \lambda \mathbb{S}\ \exists \mathbb{S}_0,\ p\ \mathbb{S}_0 \wedge g\ \mathbb{S}_0\ \mathbb{S}$$

$$p \Rightarrow p' \triangleq \forall \mathbb{S}\ p\ \mathbb{S} \to p\ \mathbb{S}' \qquad p \circ g \triangleq \lambda \mathbb{S}, \mathbb{S}'\ p\ \mathbb{S} \wedge g\ \mathbb{S}\ \mathbb{S}'$$

$$g \Rightarrow g' \triangleq \forall \mathbb{S}, \mathbb{S}'.g\ \mathbb{S}\ \mathbb{S}' \to g'\ \mathbb{S}\ \mathbb{S}' \qquad g \circ g' \triangleq \lambda \mathbb{S}, \mathbb{S}'.\exists \mathbb{S}'', g\ \mathbb{S}\ \mathbb{S}'' \wedge g'\ \mathbb{S}''\ \mathbb{S}'$$

Fig.9 Constructors for p and g

图 9 p,g 构造符

3.2.3 SCAP 系统介绍

SCAP系统类似于Hoare逻辑[22],通过一系列的推理规则来证明每个程序点满足相应的程序规范.能够满足所有的规范的程序被称作良型程序或良型世界,表述如下:

$\Psi \mapsto \{s\}\ \mathbb{W}$ (WLD,良型世界)

$\Psi \mapsto \mathbb{C} : \Psi'$ (CDHP,良型代码堆)

$\Psi \mapsto \{s\}\ \mathbb{I}$ (良型代码序列)

证明程序良型性的推理规则如图 10 所示.

$$\frac{\Psi \mapsto \mathbb{C} : \Psi'\ \Psi \subseteq \Psi'\ \Psi \mapsto \{s\}pc : \mathbb{C}[pc]\ \ \{s\}\Psi'\mathbb{S}}{\Psi \mapsto \{s\}(\mathbb{C}, \mathbb{S}, pc)} \quad \text{(WLD)}$$

$$\frac{\forall (f, s) \in \Psi' : \Psi \mapsto \{s\}\ f : \mathbb{C}[f]}{\Psi \mapsto \{s\}(\mathbb{C}, \mathbb{S}, pc)} \quad \text{(CDHP)}$$

$$\frac{\Psi \mapsto \mathbb{C}_1 : \Psi'\ \ \Psi \mapsto \mathbb{C}_2 : \Psi'\ \ \mathbb{C}_1 \# \mathbb{C}_2}{\Psi \mapsto \{s\}(\mathbb{C}, \mathbb{S}, pc)} \quad \text{(LINK)}$$

$$\frac{\begin{array}{l}\iota \notin \{\text{jmp}, \text{jmpw}, \text{je}\}\ \ \Psi \mapsto \{(p', g')\}pc+1 : \mathbb{I} \\ p \Rightarrow g_\iota\ \ (p \triangleright g_\iota) \Rightarrow p'(p \circ (g_\iota \circ g')) \Rightarrow g\end{array}}{\Psi \mapsto \{(p, g)\}pc : \iota; \mathbb{I}} \quad \text{(SEQ)}$$

$$\frac{\begin{array}{l}(f, (p', g')) \in \Psi\ \ \Psi \mapsto \{(p'', g'')\}pc+1 : \mathbb{I} \\ (p \triangleright g_{\text{je}T})\ \ p'\ \ (p \circ (g_{\text{je}T} \circ g')) \Rightarrow g \\ (p \triangleright g_{\text{je}F})\ \ p''\ \ (p \circ (g_{\text{je}F} \circ g'')) \Rightarrow g\end{array}}{\Psi \mapsto \{(p, g)\}pc : \text{je}\ \ f; \mathbb{I}} \quad \text{(JE)}$$

$$\frac{(f, (p', g')) \in \Psi\ \ \ p \Rightarrow p'\ (p \circ g') \Rightarrow g}{\Psi \mapsto \{(p, g)\}pc : \text{jmp}\ f} \quad \text{(JMP)}$$

$$\frac{\mathbb{R}(r) = f\ \ (f, (p', g')) \in \Psi\ \ \ p \Rightarrow p'\ (p \circ g') \Rightarrow g}{\Psi \mapsto \{(p, g)\}pc : \text{jmpw}\ r} \quad \text{(JMPW)}$$

$g_{\text{je}T} \triangleq \lambda \mathbb{S}, \mathbb{S}'.\ NextS_{(\text{je}, \_)}\mathbb{S}\ \ \mathbb{S}'$ (where $\mathbb{S}'.zf$ = true)

$g_{\text{je}F} \triangleq \lambda \mathbb{S}, \mathbb{S}'.\ NextS_{(\text{je}, \_)}\mathbb{S}\ \ \mathbb{S}'$ (where $\mathbb{S}'.zf$ = false)

$g_c \triangleq \lambda \mathbb{S}, \mathbb{S}'.\ NextS_{(c, \_)}\mathbb{S}\ \ \mathbb{S}'$ (for all other c)

Fig.10 SCAP inference rules

图 10 SCAP 的推理规则

程序不变量:WLD 规则中描述一个程序满足良型性(well-formedness)的所有前提:

- 根据 CDHP 规则,代码堆$\mathbb{C}$是良型的(well-formed).
- 当前模块对内规范Ψ是对外规范Ψ'的子集,Ψ中包含$\mathbb{C}$所使用到的位于其他代码块的被调用接口规范,Ψ'还包含本模块定义以及将被其他模块调用的接口说明.
- 当前pc所对应的规范s在Ψ中,因此当前指令序列$\mathbb{C}[pc]$关于规范s是良型的.
- 给定外部规范集Ψ',当前状态$\mathbb{S}$应满足断言 s.

推理规则所用符号定义如图 9.谓词$p\triangleright g_\iota$描述p满足初始状态并经过状态转移g_ι后的结果,它是状态转移g_ι的最强后条件.两个连续状态转移g和g'的组合用$g\circ g'$来表示,$p\circ g$表示在g的基础上已知p得到满足.

程序模块:在 CDHP 规则中,每个模块是由至少 1 个指令序列组成的小代码堆,每个模块的规范不仅包含当前模块内部代码块的规范,还包含了所有可能被该模块调用的其他代码块的规范,因此,SCAP 逻辑支持各个程序模块的独立验证.每个独立模块的良型性通过 CDHP 规则定义,多个互不重叠的良型模块通过 LINK 规则链接起来.WLD 规则保证所有良型模块能够链接成为一个全局良型代码堆.

指令序列:类似于传统Hoare逻辑,SCAP采用前、后条件作为程序规范.SEQ规则是对每个以顺序指令ι(不含条件跳转和函数调用指令)开始序列的判定形式.它描述在给定内部规范Ψ和满足前条件(p,g)的情况下执行以当前pc开始的指令序列是安全的.程序员需要找到一个中间规范(p'',g''),使剩余指令序列得到满足,该规范同时也作为当前指令的后条件.g_ι用来描述执行指令ι所引起的状态转移,具体定义如图 7 所示.对于顺序指令而言,NextS不依赖于当前pc值,因此用"_"表示任意pc.

找到合适的中间规范(p'',g'')后,检查SEQ规则中的 4 个条件以确定该指令序列的良型性.第 1 个前提表明剩余指令序列在该中间规范下是良型的;第 2 个前提表明,$p\Rightarrow g_\iota$检查只要初始状态满足p,状态转移g_ι就可以完成;第 3 个前提表示,如果当前状态满足p,那么在状态转移g_ι后,新状态满足p''.最后一个前提描述在当前状态满足p的情况下,g_ι和g''的组合能够满足g.

其他指令:条件跳转指令跳转成功与否依赖于其判定条件是否得到满足,因此,JE规则中使用$g_{\mathrm{je}T}$和$g_{\mathrm{je}F}$来表示不同的执行情况.无条件跳转指令可以安全执行的充分必要条件是当前断言能够蕴含目标代码处的断言,它可以看作是条件跳转je的特例.

SCAP 的完备性:基于前进性和保持性引理,SCAP 的完备性保证只要初始状态满足 WLD 规则中定义的程序不变量,整个程序就永远不会陷入滞留状态.程序运行过程中永远满足该不变量.该不变量还可以蕴含部分正确性等更多程序性质.SCAP 推理规则同时保证程序满足 jmp 或 jmpw 指令跳转目标地址处的规范,也满足程序模块边界处的规范.关于 SCAP 更进一步的信息参见文献[6].

引理 2.1(CBP progress). 如果$\Psi\mapsto\{s\}\ \mathbb{W}$,则存在世界$\mathbb{W}'$,使得$\mathbb{W}\to\mathbb{W}'$.

引理 2.2(CBP preservation). 如果$\Psi\mapsto\{s\}\ \mathbb{W}$,并且$\mathbb{W}\to\mathbb{W}'$,则存在$s'$,使得$\Psi\mapsto\{s'\}\ \mathbb{W}'$.

3.3 虚拟机验证

3.3.1 模拟关系定义

对于任意字节码虚拟机 BVM 的世界$\mathbb{W}=(\mathbb{C},(\mathbb{H},\mathbb{K}),\mathbb{K}c,pc)$,模拟运行$\mathbb{W}$的 X86 机器需要将$\mathbb{W}$保存在其内存堆中.保存在内存堆的信息维护了两个层次之间的模拟关系.为了能够正确模拟运行$\mathbb{W}$,X86 模拟程序必须保证在解释运行每一条字节码指令时,内存堆中的模拟关系信息能够完整保持,同时正确执行前文所述的 4 个阶段.令X86 机器在SCAP框架中表示为世界$\mathbb{W}_x=(\mathbb{C}_x,(\mathbb{H}_x,\mathbb{R}_x,zf),pc_x)$,则在每解释一条字节码指令之前,$\mathbb{W}$和$\mathbb{W}_x$之间存在着如下的模拟关系~:

$$\frac{\mathbb{C}_x=\mathbb{C}_{VM}\ \ pc_x=\mathrm{fetch}\ sim(\mathbb{W},\mathbb{H}_x)}{\mathbb{W}\sim\mathbb{W}_x}(\mathrm{SIM}).$$

- 代码堆$\mathbb{C}_x$中的代码必须是CertVM的代码;
- 程序计数器 pc 必须指向 CertVM 取指阶段程序的入口;
- 字节码虚拟机世界$\mathbb{W}$的信息必须被保存到$\mathbb{H}_x$中,维持相应的内存模拟关系(详细内容见下文);
- 对于通用寄存器$\mathbb{R}_x$和标志寄存器zf_x没有限制.

3.3.2 内存模拟关系

如前文所述,在X86 机器的内存堆$\mathbb{H}_x$中,数组$M_{\mathbb{C}},M_{\mathbb{H}},M_{\mathbb{K}}$和$M_{\mathbb{K}c}$分别保存了BVM的$\mathbb{C},\mathbb{H},\mathbb{K}$和$\mathbb{K}c$.除此之

外,我们还用数组M_p保存指针pc,cps和sp.因此,内存模拟关系$sim(\mathbb{W},\mathbb{H}_x)$形式的定义为

$$sim(\mathbb{W},\mathbb{H}_x) \triangleq \mathbb{H}_x = Mp \uplus M_{\mathbb{K}} \uplus M_{\mathbb{K}c} \uplus M_{\mathbb{C}} \uplus M_{\mathbb{H}} \uplus M_{\mathbb{O}},$$

其中,$\uplus$表示内存之间非重叠的合并关系,即当且仅当 dom $H1\cap$dom $H2=\varnothing$,才有 $H1 \uplus H2=H1\cup H2$.

而 $M_{\mathbb{O}}$表示M中其他剩余内存空间.字节码指令保存在 $M_{\mathbb{C}}$中,每条指令长度为 4 字节,$M_{\mathbb{C}}$定义为$\mathbb{H}_x$中一段连续内存.对于$\mathbb{C}$的任意合法标号f $(\forall f\in dom(\mathbb{C}))$,指令$\mathbb{C}[f]$保存在 $M_{\mathbb{C}}[f\times4]=\mathbb{H}_x[base(M_{\mathbb{C}})+f\times4]$的内存块.类似地,计算栈$\mathbb{K}$、内存堆$\mathbb{H}$和函数调用栈$\mathbb{K}c$的每一项在$\mathbb{H}_x$中保存时,每项占 2 字节.其中,$M_p$中保存$pc$,$cps$和$sp$,每个占 2 字节,分别保存在$M_p[0]$,$M_p[2]$以及$M_p[4]$起始的内存中.下面给出这些数组的具体定义:

$$M_{\mathbb{C}} \triangleq \mathbb{H}_x[base(M_{\mathbb{C}}),base(M_{\mathbb{C}})+\max(\mathbb{C})\times4],$$
$$M_{\mathbb{K}} \triangleq \mathbb{H}_x[base(M_{\mathbb{K}}),base(M_{\mathbb{K}})+length\ \mathbb{K}\times2],$$
$$M_{\mathbb{K}c} \triangleq \mathbb{H}_x[base(M_{\mathbb{K}c}),base(M_{\mathbb{K}c})+length\ \mathbb{K}c\times2],$$
$$M_{\mathbb{H}} \triangleq \mathbb{H}_x[base(M_{\mathbb{H}}),base(M_{\mathbb{H}})+\max(\mathbb{H})\times2],$$
$$M_p \triangleq \mathbb{H}_x[base(Mp),base(M_p)+6].$$

3.3.3 虚拟机良型性定义

通过模拟关系给出良型虚拟机(well-formed virtual machine)的定义:一个良型的虚拟机$WFVM(\mathbb{W},\mathbb{W}_x)$,对于任意字节码世界$\mathbb{W}$,存在对应的X86 世界$\mathbb{W}_x$满足$\mathbb{W}\sim\mathbb{W}_x$;并且若有字节码世界$\mathbb{W}'$且$\mathbb{W}\rightarrow\mathbb{W}'$,那么存在$\mathbb{W}'_x$满足$\mathbb{W}'\sim\mathbb{W}'_x$且$\exists n,\mathbb{W}_x\rightarrow n\,\mathbb{W}'_x$.图 11 表述了这一定义.

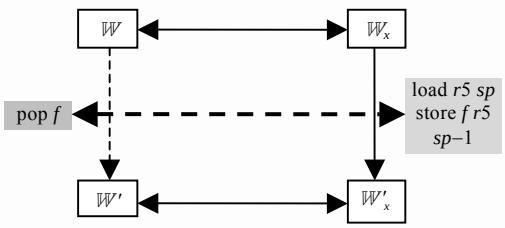


Fig.11 Illustration of well-formed virtual machine

图 11 良型虚拟机的表示

从良型虚拟机的定义可知,任意一条字节码指令的执行必须严格对应一段X86 程序,并且在这段X86 程序的入口和出口都必须严格满足模拟关系.为了利用SCAP框架证明虚拟机的良型性,我们需要利用SCAP的程序规范来描述良型虚拟机需要满足的性质.从前文的虚拟机执行机制可知,这段X86 程序恰好就是虚拟执行的 4 个阶段.因此,每一条字节码指令对应的X86 程序片段的入口一定是fetch(如图 5 所示的实例).定义fetch处程序规范$(p_{\text{fetch}},g_{\text{fetch}})$为

$$p_{\text{fetch}} \triangleq \lambda\mathbb{S}_x,\exists\mathbb{W},sim(\mathbb{W},\mathbb{S}_x.\mathbb{H}_x)\wedge Enable(c,\mathbb{S},\mathbb{K}c)\ \text{where}\ \mathbb{W}=(\mathbb{C},\mathbb{S},\mathbb{K}c,pc)\wedge c=\mathbb{C}(pc),$$
$$g_{\text{fetch}} \triangleq \lambda\mathbb{S}_x,\mathbb{S}'_x,\exists\mathbb{W},\mathbb{W}',\mathbb{W}\rightarrow\mathbb{W}'\wedge sim(\mathbb{W},\mathbb{S}_x.\mathbb{H}_x)\wedge sim(\mathbb{W}',\mathbb{S}'_x.\mathbb{H}_x).$$

其中,p_{fetch}描述在进入fetch之前,X86 世界的内存堆中必须维持某个字节码世界的模拟关系,并且要求这个字节码世界是可以运行的(也就是存在下一个状态).第 1 次进入fetch时,X86 内存堆中必定保存着字节码程序的最初始状态;g_{fetch}保证执行完解释程序后,其出口状态$\mathbb{S}'_x$能由入口状态$\mathbb{S}_x$严格推出,即$\mathbb{S}'_x$保存了$\mathbb{S}$所模拟字节码世界后续状态的模拟关系.一旦解释执行完一条字节码指令时,程序就返回到fetch入口,此时的状态$\mathbb{S}_x$必又能满足p_{fetch}.

因此,要完成整个虚拟程序的验证工作,我们只需要利用 SCAP 框架对每一条字节码指令分别证明其对应的 CertVM 实现代码满足如下性质:

$$\Psi_{\text{cvm}} \mapsto \{(p_{\text{fetch}}, g_{\text{fetch}})\}\,\text{fetch}: \mathbb{C}_{VM}[\text{fetch}].$$

4 程序实例和实现

本文第 3 节图 5 介绍的 X86 程序实例是对字节码指令 goto 进行解释执行.本节将以这段程序为例,展示 CertVM 的证明过程.

4.1 验证步骤介绍

划分指令序列:划分指令序列是验证工作的第 1 步,正确的划分才能够确保证明过程.根据图 6 中对指令序列$\mathcal{D}$的定义,整个代码片段能够自然地化成两个部分:fetch 和 goto 标号所指向的程序块(fetch 段从第 1 行开始到第 14 行结束,而 goto 段为第 15 行~第 21 行).其中,fetch 段代码是不同字节码指令解释程序所共用的,所以一旦证明,就可以被复用.

程序规范建立:程序员需要给出代码堆规范 Ψ,即从指令标号 f 到代码断言 s(即(p,g)二元组)的映射,我们将 SCAP 规范内嵌在 X86 汇编代码中,即见图 5 中花括号部分.

根据推理规则,需要给出以下位置的代码断言:每一个指令序列的开头、跳转指令(包括 jmp 和 je)的目标位置.在此段程序中,我们只需要在 fetch 和 goto 标号处加入程序规范.由前文分析可知,fetch 处的程序规范主要保证模拟关系,这里不再赘述.goto 入口处需要满足当前字节码世界 pc 所指的指令为 goto 类型即可.

程序规范的形式定义见图 12.定义中,默认 $\mathbb{W}=(\mathbb{C},(\mathbb{H},\mathbb{K}),\mathbb{K}c,pc)$且$\mathbb{S}_x=(\mathbb{H}_x,\mathbb{R}_x,zf)$.其中,$p_{\text{dispatch}}$为中间断言,后文将解释.type函数是一个把字节码指令类型映射到自然数的函数.而T_{goto}则为字节码goto所映射的自然数.

$$p_{\text{goto}} \triangleq \lambda\mathbb{S}_x, \exists\,\mathbb{W}, sim(\mathbb{W}, \mathbb{S}_x.\mathbb{H}_x) \wedge Enable(c, \mathbb{S}, \mathbb{K}c) \wedge type(c) = T_{\text{goto}},\ \text{where } c = \mathbb{C}(pc),$$

$$p_{\text{dispatch}} \triangleq \lambda\mathbb{S}_x, \exists\,\mathbb{W}, sim(\mathbb{W}, \mathbb{S}_x.\mathbb{H}_x) \wedge Enable(c, \mathbb{S}, \mathbb{K}c) \wedge \mathbb{R}_x(ax) = type(c),\ \text{where } c = \mathbb{C}(pc),$$

$$g_{\text{goto}} = g_{\text{ftech}},\ g_{\text{dispatch}} = g_{\text{ftech}}.$$

Fig.12 Specification for the X86 implementation of bytecode instruction goto

图 12 字节码 goto 指令 X86 解释执行代码的示例规范

验证并连接:为了检查一个以指令ι开始序列的良型性,程序员需要寻找恰当的中间断言(即是当前指令ι的后条件,又是后续指令序列的前条件),并选用相应推理规则完成指令ι的证明.一个简单的例子是,在dispatch标号处可以加入图 12 中所定义的断言$(p_{\text{dispatch}}, g_{\text{dispatch}})$.这个断言表示在取指译码完成之后,X86 程序正确地获得字节码世界 $\mathbb{W}$中 pc 所指向指令的类型,并将类型信息保存在寄存器 ax 中.

完成指令证明之后,程序员使用 CDHP 规则建立各个独立指令序列的良型性证明,多个互不重叠的良型代码片段可以用 LINK 规则连接在一起,最后,应用 WLD 规则把所有代码片段连接并构成全局良型代码堆.

4.2 验证实现

本文采用 Coq 证明辅助工具来实现逻辑系统和上述实例程序的证明,所有的定义和证明都可以机器自动检查.给出了 BVM 及其操作语义、X86 机器及其操作语义、SCAP 逻辑系统和 CertVM 实现及其证明的形式化表示.字节码和 X86 语法采用 Coq 的归纳定义给出,相应的操作语义和所有的 SCAP 推理规则都定义为归纳关系,逻辑系统的合理性证明则根据语法方式进行形式化和证明,同时还给出 CertVM 实现程序实例的形式化描述和证明.

所有证明实现约 14 000 行的 Coq 代码,花费数个人月.其中,接近 1/6 分的工作在于一些基本工具的实现,包括关于映射与分离逻辑的引理和策略.这些通用工具独立于本逻辑系统以及验证实例,部分内容重用自以往相关项目中并加以适当修改.其中,3 200 多行定义字节码机器及其操作语义,2 700 多行定义 X86 机器及其操作语义、SCAP 推理规则定义及其完备性证明,另外约 1 800 行是 CertVM 实现的内存映射和机器状态模拟关系

定义.本文中图 5 对应实例的证明超过 3 200 行,且 CertVM 实例直接由 X86 汇编代码开发.事实上,SCAP 系统同样可用于人工优化过的程序以及优化编译工具自动生成代码的证明.程序验证实践表明,编写程序规范依赖于程序员,其难度取决于所感兴趣的程序性质以及算法本身的复杂性.只要给出程序规范,程序证明书写就相对比较简单.对于本文虚拟机 CertVM 的证明,其程序规范由字节码操作语义和状态之间的模拟关系共同确定,而字节码的操作语义比较容易给出.因此,只要确定字节码状态和相应 X86 机器状态之间的模拟关系,就可以很方便地给出程序规范,进而完成证明实现.

Table 1 Coq proof code statistic

表 1 Coq 证明代码统计

Number	Type	Lines of proof code	
		Value	Percent
1	Basic coq lactic library	2 354	17.5
2	Bytecode virtual machine definition	3 285	24.4
3	X86 machine definition and SCAP logic	2 706	20.1
4	CertVM memory layout and SPEC relations	1 864	13.8
5	Proof code of CertVM example	3 258	24.2
	Total	13 467	100

4.3 未来扩展研究

初步完成虚拟机的构造和验证之后,还将进一步结合以往字节码程序验证的工作基础,建立一套完整的逻辑系统,将已验证的字节码程序和已验证虚拟机有机地结合起来,确保硬件环境满足其语义规范的情况下,已验证的字节码程序能够在已验证虚拟机环境中正常运行,建立完整的已验证计算环境.

目前,本文的字节码子集只针对最关键的控制栈和非结构化控制流,因此只包含其中的关键指令,后续将扩展支持更多的语言特征.系统首先要扩展的是对象、引用、方法和继承等面向对象特征,以便更好地体现当今字节码程序的发展趋势和使用情况.另外,提供例外处理支持是一个直接的扩展工作,采用类似于函数调用/返回的思路,加入例外支持应该比较容易.并发程序验证支持是又一个值得深入探讨的热点问题,随着多核处理器的发展而更加突出,即便是广泛使用的并发程序库,比如JDK同步类等,也存在一系列缺陷[23].引入上述特征之后,将进一步完成对应特征的X86 汇编实现和对应的证明实现.这些扩展工作的主要难度将在于相应的操作语义定义以及X86 实现的模拟关系建立.此外,证明过程中引入更多的自动技术以提高验证效率,也是一个需要深入研究的方向.

5 结束语

与本文CertVM验证工作相关的研究包括一系列针对字节码、虚拟机的形式化描述和检查方面的工作.近年来提出的大量用于字节码程序验证的逻辑系统[12,14−16]以及我们提出的字节码程序模块化验证和嵌套字节码程序的验证[17]等工作,都没有考虑虚拟机运行环境实现本身的可信问题.本文则深入探讨虚拟机本身的可信问题,采用汇编代码构造出一个可真实运行的虚拟机系统原型,并利用FPCC方法证明该实现符合字节码操作语义.Fong等人的可运行FJVM系统模型[24]等工作提供了一系列可用于新语言特性研究的VM模型,我们则不仅形式化地给出一个虚拟机模型,而且构造相应的可执行代码实现并给出完整证明.Klein和Nipkow深入研究了一种类JAVA语言及相应虚拟机、编译器的模型[25],给出了类JAVA语言和相应Jinja VM执行之间的模拟关系证明,但是没有回答如何确保Jinja VM实现符合其规范这一重要问题,本文的CertVM则通过形式化证明的方案给出了“VM实现是否符合规范”这一问题的确切回答.

本文基于FPCC方法构造出一个符合字节码语义规范的虚拟机原型系统.给出字节码操作语义和运行环境X86 机器的形式化定义,利用 SCAP 逻辑系统证明虚拟机实现和字节码程序之间模拟关系证明,并利用辅助工具Coq给出证明,所有证明均可机器自动检查.本系统能够确保只要字节码程序能够在给虚拟机CertVM中正常运行,其执行结果就符合字节码操作语义定义.

本文的工作采用程序验证手段构造字节码程序的可信运行环境,为广泛使用的一类复杂网络应用程序的

深刻理解、深入分析和准确验证提供理论帮助,同时为可信软件构造问题的解决提供了一种良好的思路.

致谢 在此,我们向对本文工作给予建议的同行,特别是 Yale 大学的 Zhong Shao 教授、TTI-Chicago 的 Xinyu Feng 博士和 Lehigh 大学的 Gang Tan 博士等人表示感谢.感谢论文评审专家的宝贵意见,感谢北京大学梅宏教授和国防科学技术大学王戟教授在 NASAC 2009 会议上关于可信的讨论和建议.

References:

[1] Hoare T. The verifying compiler: A grand challenge for computing research. In: Proc. of the 2003 Int'l Conf. on Compiler Construction (CC 2003). LNCS 2622, Heidelberg: Springer-Verlag, 2003. 262−272.

[2] http://bugs.sun.com/bugdatabase/top25_bugs.do

[3] Lindholm T, Yellin F. The Java Virtual Machine Specification. 2nd ed., Boston: Addison-Wesley Publishing Company, 1999.

[4] Information Technology—Common Language Infrastructure (CLI). ISO/IEC 23271:2006, 2006. http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html

[5] von Oheimb D. Hoare logic for Java in Isabelle/HOL. Concurrency and Computation: Practice and Experience, 2001,13(13): 1173−1214.

[6] Feng XY, Shao Z, Vaynberg A, Xiang S, Ni ZZ. Modular verification of assembly code with stack-based control abstractions. In: Proc. of the Programming Languages Design and Implementation (PLDI 2006). New York: ACM Press, 2006. 401−414.

[7] Feng XY, Shao Z, Dong Y, Guo Y. Certifying low-level programs with hardware interrupts and preemptive threads. In: Proc. of the Programming Languages Design and Impl. (PLDI 2008). New York: ACM Press, 2008. 170−182.

[8] Wang SY, Liang YY, Dong Y. Verification of concurrent assembly programs with a Petri net based safety policy. Tsinghua Science and Technology, 2007,12(6):684−690.

[9] Zhu YM, Zhang LW, Wang SY, Dong Y, Zhang SQ. Verifying parallel low-level programs for multi-core processor. Acta Electronica Sinica, 2009,37(z1):1−6 (in Chinese with English abstract).

[10] Guo Y, Chen YY, Lin CX. A method for code safety proof construction. Journal of Software, 2008,19(10):2720−2727 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/19/2720.htm

[11] Leroy X. Bytecode verification for Java smart card. Software Practice & Experience, 2002,32(4):319−340.

[12] Quigley CL. A programming logic for Java bytecode programs. In: Proc. of the 16th Int'l Conf. on Theorem Proving in Higher-Order Logics (TPHOLs 2003). Berlin: Springer-Verlag, 2003. 41−54.

[13] Aspinall D, Beringer L, Hofmann M, Loidl HW, Momigliano A. A program logic for resource verification. In: Proc. of the TPHOLs 2004. Berlin: Springer-Verlag, 2004. 411−445. http://portal.acm.org/citation.cfm?id=1321978

[14] Bannwart F, Muller P. A program logic for bytecode. Electronic Notes in Theoretical Computer Science, 2005,141(1):255−273.

[15] Benton N. A typed, compositional logic for a stack-based abstract machine. In: Proc. of the 3rd Asian Symp. on Programming Languages and Systems (APLAS). LNCS 3780, Berlin: Springer-Verlag, 2005. 364−380.

[16] Burdy L, Pavlova M. Java bytecode specification and verification. In: Proc. of the SAC 2006. New York: ACM Press, 2006. 1835−1839. http://portal.acm.org/citation.cfm?id=1141277.1141708

[17] Dong Y, Wang SY, Zhang LW, Yang P. Modular certification of low-level intermediate representation programs. In: Proc. of the 33rd Annual IEEE Int'l Computer Software and Applications Conf. (COMPSAC 2009). Washington: IEEE Computer Society Press, 2009. 563−570. http://dblp.uni-trier.de/rec/bibtex/conf/compsac/DongWZY09

[18] Appel AW. Foundational proof-carrying code. In: Proc. of the 16th IEEE Symp. on Logic in Computer Science. Washington: IEEE Computer Society Press, 2001. 247−258. http://portal.acm.org/citation.cfm?id=871860

[19] Lawton K, Denney B, Guarneri ND, Ruppert V, Bothamy C. Bochs user manual. 2009. http://bochs.sourceforge.net/

[20] Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.1, 2006.

[21] http://soft.cs.tsinghua.edu.cn/dongyuan/~dongyuan/verify/certvm.html

[22] Hoare CAR. An axiomatic basis for computer programming. Communications of the ACM, 1969,26(1):53−56.

[23] Sen K. Race directed randomized dynamic analysis of concurrent programs. In: Proc. of the Programming Languages Design and Implementation (PLDI 2008). New York: ACM Press, 2008. 11−21.

[24] Fong PWL. Reasoning about safety properties in a JVM-like environment. Science of Computer Programming, 2007,67(2-3): 278−300.

[25] Klein G, Nipkow T. A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Trans. on Programming Languages and Systems, 2006,28(4):619−695.

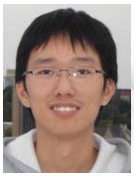
附中文参考文献:

[9] 朱允敏,张丽伟,王生原,董渊,张素琴.面向多核处理器的低级并行程序验证.电子学报,2009,37(z1):1−6

[10] 郭宇,陈意云,林春晓.一种构造代码安全性证明的方法.软件学报,2008,19(10):2720−2727. http://www.jos.org.cn/1000-9825/19/2720.htm

董渊(1973－),男,内蒙古和林格尔人,博士,副教授,CCF 会员,主要研究领域为操作系统,编译系统,基于语言的可信软件.

王生原(1964－),男,博士,副教授,CCF 高级会员,主要研究领域为程序设计语言与系统,Petri 网应用.



任恺(1987－),男,主要研究领域为系统软件,程序设计语言.

张素琴(1945－),女,教授,CCF 高级会员,主要研究领域为语言与编译技术.