

动态存储管理安全验证的 Coq 实现

项 森 陈意云 林春晓 李 隆

(中国科学技术大学计算机科学技术系 合肥 230027)

(sengx@mail.ustc.edu.cn)

Safety Verification of Dynamic Storage Management in Coq

Xiang Sen, Chen Yiyun, Lin Chunxiao, and Li Long

(Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027)

Abstract The increasing scale and complexity of software makes the software safety and security critical. Thus more and more research focuses on the development of high-assurance software. Since type system is not expressive enough, the existing research does not touch the verification of infrastructure codes. In this paper, a certified dynamic storage management library is built using Hoare-logic style reasoning at the assembly level with the assistance of a theorem formalization and proof tool called Coq, since Hoare logic is more expressive. This work is a significant application of program verification technique. The experiment shows that program verification can be applied in high-assurance software development.

Key words formal method; high-assurance software; program verification; proof carrying code

摘 要 随着软件规模和复杂度的日益提升,软件安全的问题变得越来越严峻,同时有越来越多的研究工作集中在高可信软件的开发上。由于类型系统表达能力的不足,现有的研究不触及底层软件的验证。由于 Hoare 逻辑更好的表达能力,采用 Hoare 逻辑风格的推理,在汇编语言级别,使用 Coq 形式化与定理证明工具可以实现一个经过安全验证的动态存储管理函数库,这是程序验证技术一次有意义的实践。实践表明,程序验证技术可以应用到高可信软件的开发上。

关键词 形式化方法;高可信软件;程序验证;携带证明的代码

中图法分类号 TP301.2

随着软件规模和复杂度的提高,软件安全的问题变得越来越严峻,所以如何开发高可信软件成了计算机科学研究领域一个研究的热点。在代码安全的研究领域,存在基于程序分析的方法和程序验证的方法,其中程序验证是一种严格和可靠的方法。程序验证的框架主要有携带证明的代码 PCC^[1-2]和类型化的汇编语言 TAL^[3-4]等。PCC 这个通用框架最早由 Necula 和 Lee 提出,它提供了一种通过运行前的证明自动检查保证移动代码安全性的机制。现存的 PCC 系统的研究集中在那些安全证明可以自

动产生的程序(如 TAL 程序),由于类型自身表达能力上的不足,使得现有的 PCC 系统只能验证程序一些比较简单的属性,如类型安全。

为了解决 PCC 表达能力受类型的限制的问题, CAP^[5]为 PCC 提供了一种补充的途径。在 CAP 中,程序最一般的属性包含部分正确性都能得到半自动(借助于 Coq^[6]证明助理)的验证。从根本上说, CAP 是 Hoare 逻辑^[7]风格的汇编级别程序的验证方法,所以它完全保有 Hoare 逻辑风格推理的表达能力。但是由于 Hoare 逻辑对一类(first class)代码指针

(如函数返回地址)推理支持的不足,使得 CAP 必需借助于全程序(如枚举函数所有调用点)的推理,这使得 CAP 的模块化很差,在 CAP 的框架内进行代码加安全证明的开发非常困难和不实用. Yu 在 CAP 上实现了动态存储管理库例程(malloc/free)的安全验证,但是其中每个库函数的验证都不能单独进行,这一点严重削弱了它的现实意义.

我们的工作是在 CAP 的扩展 SCAP^[8](stack-based CAP)上进行堆管理库例程(malloc/free)的模块化验证,模块化意味着独立构造每个函数的安全性证明,不再依赖于全程序的推理. 我们的研究工作主要包含以下 2 点:

- 1) 使用 Coq^[6]进行 SCAP 的形式化,包括抽象机,推理规则以及推理规则的可靠性定理;
- 2) 在 SCAP 下使用 Coq 构造动态存储管理函数的安全证明,包括规范标注和半自动的定理证明.

上述工作的意图都是为了最大程度地、方便地将程序员编程时的推理形式化为机器可检查的证明,以借助机器检查推理过程是否正确. 本工作所有的实现可以自由获得. 本文将在以下各节分别介绍这些工作,其中部分内容由于篇幅的原因只做简单叙述.

1 SCAP 简介及其 Coq 的实现

SCAP^[8]是 Hoare 逻辑风格的验证方法,但是与 Hoare 最先提出的方法^[7]是有区别的. Hoare 逻辑是一个公理系统,它的良型性公式为 $\{P\}S\{G\}$,其中 P 和 G 分别为前后断言, S 是源程序. Hoare 逻辑给出良型性公式的一组结构化的推理规则,然后基于语言的操作语义去证明该公理系统对语言的语义是可靠的. 而 SCAP 首先描述一个操作模型(抽象机)和一组静态推理规则,然后遵循类型系统可靠性证明的语法方式证明推理规则对此抽象机的可靠性. 另外,SCAP 针对低级语言,而不是源语言级别的程序. SCAP 是 CAP 的扩展,目的是为了支持函数调用的 Hoare 逻辑风格的模块化推理. 它首先由耶鲁大学 flint 研究组提出,我们使用 Coq 对它进行了形式化,严格证明了它的可靠性,并在此基础上(使用 SCAP 的推理规则)实现了动态存储管理库的模块化安全验证. 本节首先简单介绍 SCAP,然后描述如何使用 Coq 对它进行形式化.

1.1 SCAP 的抽象机和静态推理规则

SCAP 的抽象机基于一个类 MIPS 的机器. 它

的语法和指令的操作语义由图 1 描述. 此抽象机上的程序是一个三元组,包括一个代码堆 C , 一个存储状态 S 和一个当前正执行的指令序列 I . 代码堆将指令序列的标签映射到相应的指令序列;存储状态包括堆(随机存储器的抽象)和寄存器文件(寄存器组状态的抽象).

$$\begin{aligned}
 (\text{Program}) \quad P & ::= (C, S, I) \\
 (\text{CodeHeap}) \quad C & ::= (f \rightarrow I)^* \\
 (\text{State}) \quad S & ::= (H, R) \\
 (\text{Heap}) \quad H & ::= (l \rightarrow w)^* \\
 (\text{RegFile}) \quad R & ::= (r \rightarrow w)^* \\
 (\text{Registers}) \quad r & ::= (r_k)^{k \in \{0, \dots, 31\}} \\
 (\text{Nat}) \quad w, f, l & ::= i \text{ (nat nums)} \\
 (\text{InstrSeq}) \quad I & ::= l; I \mid j, f \mid \text{jal } f, f_{\text{ret}} \\
 & \quad \mid \text{jr } r_s \\
 (\text{Command}) \quad \tau & ::= \text{addu } r_d, r_s, r_t \\
 & \quad \mid \text{addiu } r_d, r_s, i \\
 & \quad \mid \text{subu } r_d, r_s, r_t \\
 & \quad \mid \text{beq } r_s, r_t, f \\
 & \quad \mid \text{lw } r_t, i(r_s) \\
 & \quad \mid \text{sw } r_t, i(r_s)
 \end{aligned}$$

If $I =$	$(C(H, R), I) \mapsto$
j, f	$(C(H, R), C(f))$ when $f \in \text{dom}(C)$
$\text{jal } f, f_{\text{ret}}$	$(C(H, R \{r_{31} \rightarrow f_{\text{ret}}\}), C(f))$ when $f \in \text{dom}(C)$
$\text{jr } r_s$	$(C(H, R), C(R(r_s)))$ when $R(r_s) \in \text{dom}(C)$
$\text{beq } r_s, r_t, f; I'$	$(C(H, R)I')$ when $R(r_s) \neq R(r_t)$ $C(H, R), C(f)$ when $R(r_s) = R(r_t)$
$\tau; I'$	$(C, i_{\text{Next}}(H, R), I')$

If $\tau =$	$\text{Next}(H, R) =$
$\text{addu } r_d, r_s, r_t$	$(H, R \{r_d \rightarrow R(r_s) + R(r_t)\})$
$\text{addiu } r_d, r_s, i$	$(H, R \{r_d \rightarrow R(r_s) + i\})$
$\text{subu } r_d, r_s, r_t$	$(H, R \{r_d \rightarrow R(r_s) - R(r_t)\})$
$\text{lw } r_t, i(r_s)$	$(H, R \{r_t \rightarrow H(R(r_s) + i)\})$ when $R(r_s) + i \in \text{dom}(H)$
$\text{sw } r_t, i(r_s)$	$(H \{R(r_s) + i \rightarrow R(r_t)\}, R)$

Fig. 1 SCAP syntax and operational semantics.

图 1 SCAP 的语法和操作语义

指令序列以无条件跳转指令结尾,包括无条件直接跳转 j, f , 函数调用 $\text{jal } f, f_{\text{ret}}$ 和间接跳转(可用做函数返回) $\text{jr } r_s$. 其中 jal 与 MIPS 机器有点不同,它显式地指明了返回地址 f_{ret} ,这是因为 SCAP 以当前指令序列代替了物理机器里存在的程序计数器 pc . 指令的操作语义由 P 上的二元关系 \mapsto 给出,每条指令的一步执行都意味着控制流和(或)存储状态的改变,其中我们使用 $R \{r_d \rightarrow _ \}$ 表示寄存器文件 R 的 r_d 映射被改变,其他寄存器不变;同样, $H \{i \rightarrow _ \}$ 表示

堆的 i 位置的值被改变,其他位置不变.与 MIPS 机器一样,函数调用 j 以 r_{31} 保存返回地址.这里只给出有代表性的指令,其他同类型指令如 $bgez$ 请查阅 MIPS 参考手册^[9].

在此抽象机器上对程序进行 Hoare 逻辑风格的推理,首先需要为程序标注形式化的规范,再根据静态推理规则进行定理证明.程序规范的结构定义如下:

(Assertion) $p, q \in State \rightarrow Prop$
 (Action) $g \in State \rightarrow State \rightarrow Prop$
 (CdSpec) $\theta ::= (p, g)$
 (CdHpSpec) $\Psi ::= \{f \rightarrow \theta\}$

在 SCAP 里,程序规范 Ψ 是指令序列标签到指令序列规范标注 θ 的映射,即每个指令序列都有一个规范标注 θ . θ 是一个二元组 (p, g) , p 指明程序运行到此指令序列时存储状态必需满足的条件, g 指明当前指令序列处的存储状态与函数返回时的存储状态之间必需满足的关系,所以 p 是存储状态上的谓词,而 g 是两个存储状态上的谓词.在程序规范确定后,需要一组推理规则去证明程序的行为满足规范,即程序在运行时,必定使得在每个指令序列入口处存储状态满足断言 p . SCAP 通过证明程序的良型性(well-formedness)保证程序的行为满足规范.限于篇幅,程序良型性推理规则参见文献[8].

1.2 SCAP 框架的 Coq 实现

Coq^[6]是基于归纳结构演算 CiC^[6]的高阶逻辑形式化和定理证明工具.本节将介绍如何使用 Coq 形式化 SCAP.图 2 显示了 SCAP 框架的部分 Coq 表示.需要注意的是在参数为类型标识符时, \rightarrow 为函数类型构造符.寄存器文件 R 的类型被形式化为 $rfile$,它是由构造符 $emptyR$ 和 $consR$ 归纳定义的一个表类型;堆 H 的类型被形式化为 $heap$,它是从地址到堆值的映射;存储状态 S 的类型是 $rfile$ 和 $heap$ 的积类型;指令序列 I 的类型被形式化为 $iseq$,此类型的值由构造符 seq 以及终结构造符 j 等(即无条件跳转指令)构造而来;代码堆 C 的类型由 $codeheap$ 归纳定义,它也是一个表类型;程序 P 被形式化为积类型 $world$. $Next_c$ 函数在实现时被形式化为普通指令 $iota$ 和指令执行前后的状态上的谓词 $next$;操作语义中的 \vdash 关系被归纳定义成二元关系谓词 $step$.规范结构的 Coq 表示为 $assert$, $action$, $codespec$ (对应指令规范标注 θ), $codehepty$ (对应程序规范 Ψ).推理规则的形式化类似于操作语义,限于篇幅,图中没有给出它们的 Coq 表示.

```

Inductive rfile : Set :=
| emptyR : rfile
| consR : rfile → reg → nat → rfile.
Definition heap : Set := addr → hvopt.
Notation state := ( heap × rfile )% type.
Inductive iseq : Set :=
| seq : iota → iseq → iseq
| j : lab → iseq
| jal : lab → las → iseq
| jr : reg → iseq.
Inductive codeheap : Set :=
| emptyCH : codeheap
| consCH : lab → iseq → codeheap → codeheap.
Notation world :=
( state × ( codeheap × iseq ) )% type.
Notation assert := ( state → Prop ).
Notation action := ( atate → state → Prop ).
Inductive codespec : Type :=
| cdspec : assert → action → codespec.
Inductive codehepty : Type :=
| emptyCT : codehepty
| consCT :
lab → codespec → codehepty → codehepty.
Inductive next : iota → state → state → Prop :=
| next_addu :
  ∀ rd rs rt : reg | h : heap | rf : rfile ).
  next ( addu rd rs rt )
  ( h rf )
  ( h , updateR rf rd ( lookupR rf rs ) + ( lookupR rf rt ) )
  ...
Inductive step : world → world → Prop :=
| step_j :
  ∀ f : lab | h : heap | rf : rfile | ch : codeheap | is : iseq ).
  lookupCH ch f is →
  step ( ( h rf ) ( ch j f ) ) ( h rf ) ( ch is )

```

Fig. 2 Coq formalization of SCAP.

图 2 SCAP 的 Coq 形式化

SCAP 推理规则是可靠(安全)的,可靠意味着 SCAP 下的良型程序的 Preservation 和 Progress 属性^[10],即良型的 SCAP 的程序 P 一定可以运行一步到 P' (progress),并且 P' 也是良型的(preservation).作者同样使用 Coq 形式化地证明了 SCAP 规则的可靠性.

2 动态存储管理库函数的安全验证

本节将介绍如何使用 Coq 进行动态存储管理库函数的安全验证.限于篇幅,将只讨论 malloc 函数的安全验证.首先介绍我们采用的一个简单的动态存储管理算法,然后是 malloc 的代码堆 C 的 Coq 表示以及代码的规范标注,最后简单介绍如何使用 SCAP 的推理规则进行定理证明.

2.3 规范标注

在如图 3 所示的形式化 malloc 汇编指令序列后,还需要为代码标注规范.回忆上一节中描述的规范结构 (p, g) ,即我们要为每个指令序列入口标注 p 和 g .以 malloc 的 prolog 序列为例,在入口处,机器的存储状态必须满足如下断言 p :

1) 假定栈在地址空间处于区间 $[B, 0]$ 内,并往低地址方向增长,那么栈顶指针 sp 必须大于 4 (SCAP 里以 32b 双字为内存访问的单位),否则栈中没有足够的空间存放被调用者返回寄存器以及局部变量;

2) 帧指针 fp 必须不大于 B ,并且 fp 要大于 sp ,只有这样 fp 和 sp 才确定一个正常的调用者的栈帧;

3) 返回地址 $ra \neq 0$,确保能正常返回到调用者;

4) 参数寄存器 $a_0 > 0$,不支持请求大小为 0 的空闲块;

5) 堆的情况稍微有些复杂.首先,堆中存放了一个全局变量 $flist$,它指向空闲内存块链表的头节点;堆还包含一个空闲块链表;堆的最低地址段 $[B, 0]$ 用做栈;最后堆中还包含在 malloc 里不会用到,我们也不关心的其他数据结构.堆的这几部分互不相交.

关于寄存器的断言使用命题逻辑即可形式化,如前 4 条可以写做 $\mathcal{R}(sp) > 4 \wedge \mathcal{R}(fp) > \mathcal{R}(sp) \wedge \mathcal{R}(ra) \neq 0 \wedge \mathcal{R}(fp) \leq B \wedge \mathcal{R}(a_0) > 0$.堆上的断言我们采用分离逻辑(separation logic)^[11]来编码.分离逻辑是对谓词逻辑的封装(如封装了“ $*$ ”连接词)这种封装使得进行堆的形式化描述变得简单,本文不再介绍关于分离逻辑的更多细节.现在看最后一点,仔细分析堆的每一部分,全局变量 $flist$ 可以使用单值堆谓词 $SigH\ p\ v \triangleq \{p \rightsquigarrow v\}$ 描述, $SigH\ p\ v$ 要描述的是堆只包含一个存储单元,地址为 p ,值为 v .栈是一段连续的地址空间,使用 $Stk\ p\ q \triangleq \{p, p-1, \dots, q+1 \rightsquigarrow \dots, \dots, \dots\}$ 描述, $Stk\ p\ q$ 要描述的是堆包含地址从 $p \sim q+1$ 的存储单元,值任意;图 4 介绍如何形式化描述空闲块链表, $MBlk\ p\ q\ s$ 描述了一个首地址为 $p-2$, $next$ 域为 q , $size$ 域为 s (含 $next$ 域和 $size$ 域)的空闲块(参考 $chunk_t$ 类型); $MBlkLst\ n+1\ p\ q$ 归纳地描述了长度为 $n+1$ 、头节点地址为 p 、尾节点的 $next$ 域为 q 的空闲块链表.

显然,当 $q = 0$ 时,这就是一条完整的链表,正如 Good hd 描述的那样.

$$\begin{aligned}
 MBlk\ p\ q\ s &\triangleq p > 2 \wedge s > 2 \wedge (p-2 \rightsquigarrow q) * (p-1 \rightsquigarrow s) * \\
 &\quad (p, \dots, p+s-3 \rightsquigarrow \dots, \dots) \\
 MBlkLst\ 0\ p\ q &\triangleq emp \wedge (p = q) \\
 MBlkLst\ n+1\ p\ q &\triangleq \exists p'. (MBlk\ p+2\ p' \rightsquigarrow) * (MBlkLst\ n\ p'\ q) \wedge \\
 &\quad (p < p' \vee p' = 0) \\
 Good\ hd &\triangleq \exists n. (MBlkLst\ n\ hd\ 0)
 \end{aligned}$$

Fig. 4 Free block list formalization in separation logic.

图 4 空闲块链表的分离逻辑表示

在确定 p 后,还需要确定 C_malloc_prolog 入口处状态与 malloc 返回时状态之间的关系 g .入口处状态与返回时状态必须满足的关系如下:

1) 所有的被调用者保存寄存器的值保持不变;

2) 帧指针寄存器 fp 、栈指针寄存器 sp 、返回地址 ra 保持不变;

3) 除被调用者本身的栈帧外,其他栈帧的内容保持不变,这一点保证不会覆盖;

4) 其他栈帧中的返回地址和被调用者保存寄存器;

5) 堆的后状态除同样包含全局变量 $flist$ 、空闲块链表、栈外,返回值 v_0 或者指向一个大小;

6) 满足要求的空闲块,或者 $v_0 = 0$;堆的前状态中不被关心的部分完全保留下来.

C_malloc_prolog 处的规范 (p, g) 的 Coq 表示如图 5 所示:

```

Definition P_malloc_prolog `assert ::=
  fun s `state =>
    getValInR s sp < getValInR s fp ^ getValInR s sp > 4 ^

Definition G_malloc_prolog `action ::=
  fun s s' `state =>
    getValInR s' sp = getValInR s sp ^ ...

Notation T_malloc_prolog ::=
  (cdspec P_malloc_prolog G_malloc_prolog).
  
```

Fig. 5 Specification of malloc prolog.

图 5 malloc 的 prolog 指令序列规范

2.4 定理证明

根据 SCAP 的规则要证明的是所有指令序列的良型性.以 malloc 的 prolog 指令序列为例,需要证明 C_malloc_prolog 在一个规范上下文 Ψ 下的良型性^[8].由于 prolog 指令序列只会跳转到 init 指令序列,根据规则 JUMP, prolog 指令序列的良型性证明用到的程序规范上下文 Ψ 仅需包含 init 序列的规范 T_malloc_init .所以 prolog 的程序规范上下文 Ψ

定义如下,其中 L_malloc_init 为 $init$ 序列的标签, $psi_malloc_$ 为程序规范上下文 Coq 标识符的前缀:

Notation $psi_malloc_prolog :=$
 $consCT L_malloc_init T_malloc_init emptyCT.$
 那么 $prolog$ 指令序列的良型性定理为

定理 1. $psi_malloc_prolog ; T_malloc_prolog \vdash$
 $C_malloc_prolog.$

证明. 对 $prolog$ 中的每条指令逐条应用推理规则 CMD 等. 由于代码堆规范上下文只给出序列入口处的规范, 所以应用推理规则时需给出下条指令处的规范. 例如, 应用规则 CMD 需要给出 (p', g') . (p', g') 不是任意给出的, 它要求规则 CMD 上部的条件都能得到证明. 与第 2.3 节中 (p, g) 的设计一样 (p', g') 也不难给出. 证毕.

类似地, 可以证明所有指令序列的良型性. 在 Coq 中, 采用交互式的定理证明方式, 即由用户根据当前要证明的目标输入策略, 这个策略或者直接完成当前目标的证明, 或者将当前目标转化为多个子目标, 直到不存在待证明的目标. 据我们的统计, 对每条指令平均需要输入 100 条证明策略, 而且大部分的证明策略都是针对堆上的推理, 这使得证明的效率很低, 因为用户要花费大量的时间输入策略进行堆操作指令如 sw, lw 的推理. 回忆在第 2.3 节中我们使用分离逻辑^[11]描述堆的状态, 分离逻辑可以使堆状态的描述变得简单, 但是堆上的形式化推理仍然比较复杂(这与拥有复杂的动态数据结构的程序难于理解原因一样). 为了使堆上的推理变得简单, 我们建立了一个 Coq 模块, 它包含自定义的证明策略, 这些策略都基于分离逻辑的常见性质. 实验表明, 自定义的分离逻辑证明策略的使用使得每条指令平均只需输入 40 条策略, 这不仅仅大大减轻了证明的工作量, 而且使得证明变的简洁有条理.

3 相关工作

大部分 PCC ^[1] 系统以及类型系统的研究^[3, 12] 都不会触及系统级别代码的验证, 因为它们采用的自动机制限制了它们对高级属性(如部分正确性)验证的能力, 它们只能保证程序不会访问空指针之类的属性. 我们的工作使用 Hoare 逻辑^[7]风格的推理半自动地构造证明, 这使得我们的实现严格保证程序最一般的属性, 例如我们保证一个块被释放时, 如果它的相邻块为空闲的, 一定会引起合并. 另外大部分 PCC 系统的函数调用都是 CPS (continuation

passing style)风格的, 而现代编译器都是采用栈式的函数调用, 这意味着从高级语言得到 PCC 代码需要进行栈式调用风格到 CPS 的转换. 我们的实现采用栈式的函数调用, 所以无须进行栈式到 CPS 的转换.

CAP ^[5] 同样借助 Coq 实现了动态存储管理库函数的安全验证, 但是 CAP 最致命的缺陷在于不能独立标注每个指令序列的规范, 造成每个指令序列的良型性证明不能独立构造. 它很差的模块性使得基本不能用来开发高安全性要求的软件. CAP 采用证明模板(对函数的每一处调用, 应用此模板构造新的证明)缓解这一重大的缺陷, 但同样使得证明和规范标注很不清晰(特别是函数入口规范). 我们的规范标注增加 g , 使得可以独立标注每个指令序列的规范以及构造证明. 另外, 我们的堆管理库函数 $malloc$ 可能失败, 这比 CAP 的 $malloc$ 实现更符合实际.

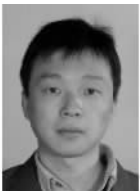
4 结 论

我们利用 Coq 工具开发了一个经过安全验证的堆管理函数库, 这是使用程序验证技术进行高可信软件开发的一次有意义的重要实践, 它说明了借助于定理证明工具进行高可信软件开发是可行的. 但是, 在这项技术的实用化上仍存在很多待解决的问题. 其中最大的问题是开发效率, 程序员用 C 或者汇编可以在 1 天左右的时间完成本文的 $malloc/free$ 函数的编写, 并且可以比较自信地声称它是安全的. 但是构造 $malloc/free$ 的形式化安全证明花了两个星期(不包括 $SCAP$ 框架的形式化), 如果采用高效的动态存储管理算法, 这个时间会更长. 考虑到软件测试等其他方法的安全保证也要花费很大代价并且只能提供不可靠的保证, 此方法仍有比较好的应用前景. 为使这项技术实用化, 未来的工作会集中在开发更智能的定理证明工具, 将用户的策略输入减到最小. 另外, 源语言级别的程序验证技术的研究也将提高高可信软件的开发效率.

参 考 文 献

- [1] G C Necula. Proof-carrying code[C]. In: Proc of the 24th ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages (POPL '97). New York: ACM Press, 1997. 106-119
- [2] A W Appel. Foundational proof-carrying code[C]. In: Proc of Symp on Logic in Computer Society (LICS '01). Los Alamitos, CA: IEEE Computer Society Press, 2001. 247-258

- [3] G Morrisett , D Walker , K Crary , *et al.* From system F to typed assembly language [J]. *ACM Trans on Programming Languages and Systems* , 1999 , 21(3) : 527-568
- [4] G Morrisett , K Crary , N Glew , *et al.* Stack-based typed assembly language [C]. In : *Proc of 1998 Int 'l Workshop on Types in Compilation* , LNCS 1473 . Berlin : Springer-Verlag , 1998 . 28-52
- [5] D Yu , N A Hamid , Z Shao . Building certified libraries for PCC : Dynamic storage allocation [J]. *Science of Computer Program* , 2004 , 5(1-3) : 101-127
- [6] The Coq Development Team . The Coq Proof Assistant Reference Manual , Version V8.0 , <http://coq.inria.fr/> , 2004
- [7] C A R Hoare . An axiomatic basis for computer programming [J]. *Communication of ACM* , 1969 , 12(10) : 576-580
- [8] X Feng , Z Shao , A Vaynberg , *et al.* Modular verification of assembly code with stack-based control abstractions [C]. In : *Proc of ACM SIGPLAN on Programming Language Design and Implementation* . New York : ACM Press , 2006 . 401-414
- [9] MIPS32 Architecture for Programmers [M]. Mountain View , CA : MIPS Technologies , Inc , 2003
- [10] B C Pierce . *Types and Programming Languages* [M]. Cambridge : The MIT Press , 2002
- [11] J Reynolds . Separation logic : A logic for shared mutable data structures [C]. In : *Proc of the 17th Annual IEEE Symp on Logic in Computer Science* . Los Alamitos , CA : IEEE Computer Society Press , 2002 . 55-74
- [12] Chen Hui , Chen Yiyun , Wu Ping , *et al.* A typed low-level language for Java virtual machine [J]. *Journal of Computer Research and Development* , 2006 , 43(1) : 15-22 (in Chinese) (陈晖 , 陈意云 , 吴萍 , 等 . 一种用于 Java 虚拟机的类型化低级语言 [J]. *计算机研究与发展* , 2006 , 43(1) : 15-22)



Xiang Sen , born in 1979 . Received his B.E. 's degree in computer science and technology from University of Science and Technology of China (USTC) , Hefei , Anhui , in 2001 . And he received his Ph. D. degree in computer science from USTC in 2006 . His main research interests include programming language theory , and language-based code safety and security .

项 森 , 1979 年生 , 博士研究生 , 主要研究方向为程序设计语言、理论和基于语言的代码安全 .



Chen Yiyun , born in 1946 . Professor and Ph. D. supervisor at the Department of Computer Science and Technology of USTC , Hefei , Anhui . His main research interests include programming language theory and implementation technique , formal methods , software safety and security , etc .

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理论和实现技术、形式化方法、软件安全等 .



Lin Chunxiao , born in 1981 . Received his B. E. 's degree in computer science and technology from USTC , Hefei , Anhui , in 2003 . Currently , he is a Ph. D. candidate in computer science from USTC . His main research interests include program verification and formal methods .

林春晓 , 1981 年生 , 博士研究生 , 主要研究方向为程序验证和形式化方法 .



Li Long , born in 1979 . Received his B. E. 's degree in computer science and technology from USTC , Hefei , Anhui , in 2003 . Currently , he is a Ph. D. candidate in computer science from USTC . His research interests include program verification and formal methods .

李 隆 , 1979 年生 , 博士研究生 , 主要研究方向为程序验证和形式化方法 .

Research Background

Since types are not expressive enough to encode rich program properties , the existing type-based approach of program verification can't touch the safety and security of infrastructure software . Substituting types with logic predicates can improve the power of languages , just as SCAP does . In this paper , a dynamic storage management library is built in SCAP . The library is verified by constructing proof of program satisfying its logic specification . Since SCAP is a sound language , it is guaranteed that the library routines can be executed as expected . SCAP language , library , and the proof are all encoded in Coq . Our work is supported by the National Natural Science Foundation of China (60473068 , 60673126) and Intel China Research Center (Beijing) .