

一种构造代码安全性证明的新方法^{*}

郭宇^{1,2+}, 陈意云^{1,2}, 林春晓^{1,2}

¹(中国科学技术大学 计算机科学技术系, 安徽省合肥市 230027)

²(中科大苏州研究院 软件安全实验室, 江苏省苏州市 215123)

A New Method for Code Safety Proof Construction

GUO Yu^{1,2+}, CHEN Yi-Yun^{1,2}, LIN Chun-Xiao^{1,2}

¹(Department of Computer Science, University of Science and Technology of China, Hefei, 230026, China)

²(Software Security Lab., Suzhou Institute for Advanced Study, USTC, 215123, China)

+ Corresponding author: Phn: +86-512-87161322, E-mail: guoyu@mail.ustc.edu.cn

Received 2007-00-00; Accepted 2007-00-00

Abstract: Foundational Proof-Carrying Code (FPCC) is a new technique for building high-assurance software and safe mobile code. FPCC is the ordinary code with safety proof attached. The safety of FPCC is ensured by the soundness of a foundational logic system. To produce FPCC, it is an essential step to construct the foundational safety proof. This paper proposes a new method to achieve proof construction, the basic idea of which is to construct proof with auxiliary recursive functions in the foundational logic. In this way, the workload of proof construction and the size of constructed proof can be reduced while maintaining the same trusted computing base. This paper also illustrates how to adapt this method to a type-based FPCC system, where the safety proof can be constructed automatically. All this work is implemented in the proof assistant Coq.

Key words: Type Theory, Software Security, Proof-Carrying Code, Program Verification, Typed Assembly Language

摘要: 基础性携带证明的代码(FPCC)是一种构建高可信软件系统和安全移动代码的新技术。FPCC 是一段附加上安全性证明的可执行代码。此安全性证明是以严格的逻辑系统为基础的, 该基础逻辑系统的可靠性保证了FPCC 代码的安全性。构造代码的安全性证明是产生 FPCC 代码的过程中非常重要的步骤。本文提出一种构造代码安全性证明的新方法。这种方法的基本思想是在基础逻辑中定义辅助递归函数来帮助构造证明。这种构造方法在不增加系统信任计算基础的情况下可以大大减轻构造证明的工作量, 并且减小安全性证明的规模。本文还介绍了该方法在一个 FPCC 系统中的应用。在这个系统中使用该方法使得代码的安全性证明可以自动产生。本文的全部工作的细节已在证明辅助工具 Coq 中实现。

关键词: 类型理论; 软件安全; 携带证明的代码; 程序验证; 类型化汇编语言

中图法分类号: TP301 文献标识码: A

^{*} Supported by the National Natural Science foundation of China under Grant No. 60673126 (国家自然科学基金)

作者简介: 郭宇(1979-): 男, 博士后, 主要研究领域为程序验证、软件安全。陈意云(1946-): 男, 教授, 博士生导师, 主要研究领域为程序设计语言的理论和实现技术、程序验证、软件安全等。林春晓(1981-): 男, 博士生, 主要研究领域为程序验证、软件安全。

1 背景

携带证明的代码(Proof-Carrying Code, PCC)^[1]最早由 Neula 提出,其定义是可执行代码携带上关于自身安全性的形式证明。代码消费方不需要信任 PCC 代码的生产方,即可执行代码与安全性证明不再属于整个系统安全性的信任计算基础(Trusted Computing Base, TCB)。随后 Appel 与 Felty 提出了基础性携带证明的代码(Foundational Proof-Carrying Code, FPCC)^[2]的概念。FPCC 的基本思想是将 PCC 代码与复杂的推理系统都形式化在一个底层的基础逻辑(Foundational Logic)中,于是推理系统的可靠性就可以用基础逻辑保证,复杂的推理系统可以被排除出 TCB。其 TCB 仅包括安全策略、机器模型、基础逻辑以及基础逻辑的证明检查器。

随后的各种 FPCC 实现采用了不同的推理系统来构造代码的安全性证明。例如某些实现采用类型系统,如 LTAL(Low-level Typed Assembly Language)^[3], TALT(Typed Assembly Language Two)^[4];还有一些实现采用逻辑系统,如 CAP(Certified Assembly Programming)^[5,6]等;此外还有一些 FPCC 系统同时采用了类型与逻辑的混合系统作为推理系统,比如 OCAP(Open CAP)^[7]。

1.1 已有的证明构造方法

构造基础的安全性证明是 PCC 的关键步骤。理论上代码生产方可以产生完整的安全性证明并将其附加在可执行代码上发送给代码消费方,代码消费方利用证明检查器来检查这个证明的正确性。对于代码生产方而言,传统的证明生产过程较为复杂与烦琐。另一方面传统的安全性证明往往过于庞大,证明的规模远超代码,并且证明的规模有可能会随着可执行代码的长度的增长以更大的幅度增长。庞大的证明将导致携带证明的代码难以在低速的网络上传输,并且会大大增加证明检查的开销。

针对上面的问题,以往的一些 FPCC 的实现采取了一定的对策。例如在一部分基于类型系统的 FPCC 系统实现中,使用与安全性证明有关的类型标注(Annotation)来代替完整的证明,如图 1(a)所示。例如 LTAL 系统与 TALT 系统将代码的类型信息与类型推理规则附加在代码上,代码消费方接收到代码之后,使用一个 Prolog 风格的解释器^[8,9]检查代码是否满足类型推理规则。这种方法虽然有效地减小了代码携带的信息的规模并降低了代码生产方的负担,但是额外的解释器增加了系统安全性的 TCB。

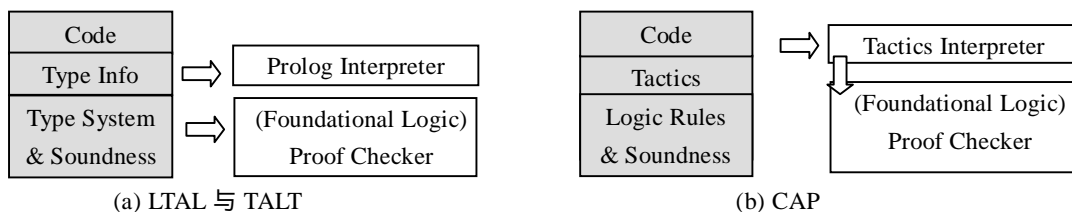


Fig. 1. Traditional FPCC
图 1. 传统的 FPCC

此外在一些基于逻辑系统的 FPCC 系统实现中,使用证明策略(Tactic)来编码证明。证明策略是定理证明器中由程序员输入的面向证明目标(Goal)的证明脚本语言。如图 1(b)所示,CAP 系统将证明策略脚本附加在代码上,代码消费方接收到代码之后,使用证明策略解释器来重新遍历检查证明过程,然后产生完整证明并交给基础逻辑的证明检查器来检查证明的正确与否。使用证明策略来编码并构造证明,可以在一定程度上达到降低的证明复杂度的目的,且不需要信任证明策略解释器。但是这种方法无法避免产生规模庞大的证明。

1.2 本文的证明构造方法

在目前的 FPCC 研究中,基础逻辑通常使用类型化 λ 演算来表示,例如 ELF^[10]、CiC(Calculus of inductive Constructions)^[11]等。根据 Curry-Howard 同构原理,类型化 λ 演算中的类型(Type)可以用来表示逻辑系统中的命题(Proposition),类型化 λ 演算的项(Term)可以用来表示逻辑系统中的证明(Proof)。同时值得注意的是 λ 演算也可以表示计算(Computation),类型化 λ 演算中可以定义递归函数,也具有较强的计算能力。

本文介绍一种基于辅助递归函数的证明构造方式,它利用了1 演算中的递归函数,在不增加系统 TCB 的前提下,增强了证明构造的自动化程度并有效减小了证明的规模。此方法的基本思想如图 2 所示,首先在基础逻辑中定义递归函数,然后使用递归函数来代替推理规则进行证明的构造,代码生产方将这些递归函数以及由递归函数构造的证明附加在可执行代码上交付给代码消费方,代码消费方使用基础逻辑证明检查器对这些递归函数进行计算,完成证明正确性的检查。

因为递归函数不是独立于基础逻辑系统之外,而是定义在基础逻辑系统之中,所以此证明构造方法具有如下优势:(1)由于使用递归函数代替了部分或全部的推理规则,证明构造过程可以(半)自动完成,从而提高了证明构造的自动化程度,易于代码生产方产生 FPCC 代码。(2)利用递归函数构造的证明不会随着可执行代码的长度增长而大幅度增长,证明检查过程中避免了操作规模庞大的证明。(3)定义在基础逻辑系统之中的递归函数不需要被信任,即不属于系统的 TCB。代码消费方在检查安全性证明时,除了基础逻辑证明检查器之外,不再需要信任额外的检查程序。(4)代码携带的安全性证明仍然是严格意义上的基础的安全性证明,不局

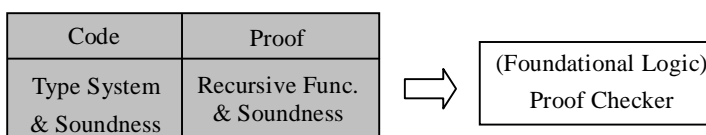


Fig. 2. FPCC built with recursive functions
 图 2. 使用递归函数构造的 FPCC

限于某一种特定的推理系统,因此证明可以基于多态类型系统、一阶算术逻辑系统、或者是多种由不同形式推理系统构成的混合系统。(5)此方法可以与已有的证明构造方法(TALT、CAP 等系统)相结合。

本文在第 2 节使用一个简单的例子来解释此证明构造方法的步骤和原理,然后在第 3 节介绍此方法在一个简化 FPCC 系统中的应用。由于篇幅所限,本文只给出涉及文中内容的部分形式定义与相关定理证明的大概思路。此 FPCC 系统的全部内容都已在证明辅助工具 Coq 中实现,完整的实现代码可以从本文的网站(<http://sbg.ustcsz.edu.cn/~guoyu/proof/>)上下载。此方法已被应用到了垃圾收集器^[12]与多线程库^[13]的证明中。第 4 节简要介绍 Coq 代码实现与实验结论,第 5 节比较相关的工作,最后一节为结束语。

2 利用辅助递归函数构造证明

假设一个定理的证明基于一些推理规则(定义在基础逻辑系统内部),如果由这些推理规则构成的证明可以找到一个算法来机械地完成构造,那么可以在(基础逻辑系统内部)定义递归函数实现这一算法,然后代替原有的推理规则来构造证明。此外,递归函数关于推理规则的可靠性必须在基础逻辑系统内部被证明。利用此递归函数与其可靠性证明,我们就可以构造与原来使用推理规则构造的证明等价的证明。下面将通过一个简单的例子来解释如何利用辅助递归函数构造证明。

2.1 基于推理系统的证明构造

下面定义一个推理系统来推理一个自然数是否是斐波纳奇(Fibonacci)数:

$$\frac{}{0 > 1} \text{ (fibz)} \quad \frac{}{1 > 1} \text{ (fibo)} \quad \frac{m > n_1 \quad m + 1 > n_2}{m + 2 > n_1 + n_2} \text{ (fibp)} \quad \frac{m > n}{\text{Fib } n} \text{ (fib)}$$

其中 $m > n$ 表示斐波纳奇数列的第 m 个数是 n , $\text{Fib } n$ 表示 n 属于斐波纳奇数列。推理规则在 CiC 中定义为:

$$\begin{aligned} \text{Fib } @ \text{ fibz} : 0 > 1 \mid \text{fibo} : 1 > 1 \\ \mid \text{fibp} : (m > n_1) \rightarrow (m + 1 > n_2) \rightarrow (m + 2) > (n_1 + n_2) \\ \mid \text{fib} : (\exists m. m > n) \rightarrow \text{Fib } n \end{aligned}$$

其中 fibz、fibo、fibp 与 fib 是 Fib 谓词的构造符(Constructor), fibp 与 fib 构造符是 λ 函数项。由上面推理规则可知, 命题 Fib 3 与命题 Fib 8 是此推理系统中的定理, 其证明用 λ 项表示如下:

$$\langle 3, \text{fibp fibo (fibp fibz fibo)} \rangle : \text{Fib 3}$$

$$\langle 5, (\text{fibp fibo (fibp fibz fibo)}) (\text{fibp (fibp fibz fibo) (fibp fibo (fibp fibz fibo))}) \rangle : \text{Fib 8}$$

这里的符号 $\langle x, a \rangle$ 表示依赖对, 即命题 $\exists x.T$ 的证明, 其中 a 的类型是 T 。命题 Fib 34 也是此推理系统的一个定理, 但是此定理的证明项的规模接近 1,000 字节。尽管这些证明项中都存在着一定冗余, 但是经过优化之后其规模仍然难以避免地随着输入数值的增加而大幅增长。

2.2 基于递归函数的证明构造

本小节阐释如何利用递归函数代替上述推理规则来构造证明。判断一个自然数是否是斐波纳奇数的存在多种判定过程, 本小节给出一种简单的判定过程, 在基础逻辑 CiC 中定义如下:

```

fix  $f(n)$  @ case n of
  | 0  $\mathbb{P}$  1
  | 1  $\mathbb{P}$  1
  | otherwise  $\mathbb{P}$   $f(n-1) + f(n-2)$ 
end

fix  $g(m, n)$  @ if  $f(m) = n$  then true
  else if  $m = 0$  then false else  $g(m-1, n)$ 

```

$f(n)$ 函数是取斐波纳奇数列中的第 n 个数; $g(m, n)$ 函数从 m 到 0 依次测试 n 是否属于斐波纳奇数列; $isfib(n)$ 函数从 n 开始调用 $g(n, n)$ 来递归测试 n 是否属于斐波纳奇数列; $isfib(n)$ 返回值为布尔值 true 或者 false。此递归函数的定义需要证明对于上面的推理规则是可靠的, 在基础逻辑 CiC 中可以证明下面的定理:

定理 1. 如果自然数 n 输入到函数 $isfib(n)$ 中并返回结果 true, 那么 Fib n 一定在推理系统中存在推导。此定理在 CiC 中表示为:

$$\forall n. (isfib(n) = \text{true}) \rightarrow \text{Fib } n$$

证明: 使用数学归纳法可以证得, 具体细节略。

假定符号 Prf_{TH1} 表示定理 1 的证明项, 那么 Prf_{TH1} 的类型是 $\forall n. (isfib(n) = \text{true}) \rightarrow \text{Fib } n$ 。根据 Curry-Howard 同构原理, Prf_{TH1} 也可以被看作是一个函数项, 它接受两个参数: 第一个参数是自然数 n , 第二个参数是命题 $isfib(n) = \text{true}$ 的证明项。函数的返回结果就是 Fib n 的证明项。

在 CiC 理论中, 如果一个项 A 的类型 T 可以由另一个类型 T' 计算归约得到, 那么 CiC 的类型系统会认为项 A 的类型既可以是 T , 也可以是 T' 。例如上一小节的证明项 $\langle 3, \text{fibp fibo (fibp fibz fibo)} \rangle$ 的类型可以是 Fib 3, 也可以是 Fib (1+2)。又如命题 $isfib(3) = \text{true}$ 的证明项也是命题 $\text{true} = \text{true}$ 的证明项。在 CiC 中, 表达式 $(\text{refl_equal } A)$ 表示命题 $A = A$ 的证明项。

下面我们就可以构造出 Fib n 的证明项:

$$\text{Prf}_{\text{TH1}} n (\text{refl_equal true}) : \text{Fib } n$$

观察上面的证明项, $isfib(n)$ 函数隐含在证明项的类型中, 它的计算过程也隐含在证明项的类型检查过程中。因为 Prf_{TH1} 的类型是一个依赖类型(Dependent Type), Prf_{TH1} 的第二个参数的类型 $isfib(n) = \text{true}$ 中含有一个变量 n , 它必须等于函数 Prf_{TH1} 的第一个参数 n , 所以当 Prf_{TH1} 接受第一个参数 n_0 后, CiC 的类型检查器将要求第二个参数的类型必须为 $isfib(n_0) = \text{true}$, 而此时第二个参数项为 (refl_equal true) , 于是 CiC 的类型检查器会计算等式左边的 $isfib(n_0)$, 假设计算结果等于 b , 那么 CiC 的类型检查器接下来检查 (refl_equal true) 的类型是否是 $b = \text{true}$ 。如果 n_0 确实是斐波纳奇数, 使得函数 $isfib(n_0)$ 计算结果为 true, 那么 $\text{Prf}_{\text{TH1}} n_0 (\text{refl_equal true})$ 将是 Fib n_0 的证明项; 否则 $\text{Prf}_{\text{TH1}} n_0 (\text{refl_equal true})$ 不是一个良类型的 CiC 项。例如:

$$\text{Prf}_1 8 (\text{refl_equal true}) : \text{Fib 8}$$

$$\text{Prf}_1 9 (\text{refl_equal true}) : \text{ill-typed}$$

由上式, 仅输入数值即可构造某个数 n 是斐波纳奇数的证明, 而且构造出证明的规模固定, 即递归函数的大小加上其可靠性证明的大小。

(机器)	$M ::= (C, S)$	(状态)	$S ::= (R, pc)$
(代码堆)	$C ::= \{f a i\}^*$	(寄存器)	$r ::= r_1 r_2 \dots r_8$
(寄存器文件)	$R ::= \{r a w\}^*$	(程序计数器)	$pc ::= f$
(指令)	$i ::= \text{addu } r_d r_s r_t \text{move } r_d r_s \text{li } r_d w \text{bgtz } r_s, f j f \dots$		
(执行)	$M \hat{I}^n M'$	机器 M 执行 n 条指令($n>0$)到 M' 。	

Fig. 3. Machine model definition

图 3. 机器模型定义

2.3 讨论

利用辅助递归函数构造证明方法的关键处是：定义的辅助递归函数并不是用来产生证明的工具，而是作为证明项的一部分。这样构造出的证明项是由(1)递归函数的定义、(2)递归函数的可靠性证明、与(3)参数构成，递归函数被隐藏在可靠性证明的类型中。因此递归函数的计算不是发生在证明的构造期，而是被延迟到了证明的检查期。

即使一个推理系统不存在判定过程，其局部仍然有可能存在部分的推理规则的判定过程，即其中一部分推理规则可以使用递归函数代替。对于这种情况，本节的方法仍然适用。例如 Presburger 算术与集合等价判定等推理系统都存在判定过程，因此也可以应用本节介绍的方法。

3 构造代码的安全性证明

本节介绍如何把利用辅助递归函数的证明构造方法应用到一个 FPCC 系统中。此 FPCC 系统采用类型系统 TAL^[14]作为推理系统。TAL 推理系统的基本思想是使用类型系统为指令以及指令操作数加上静态类型，通过约束指令的操作对象来保证指令操作的安全性，如控制流安全、访问内存安全等等。

TAL 类型系统通常存在可判定的类型检查算法，因此可以在基础逻辑中构造一个类型检查函数并证明此类型检查函数的可靠性，然后利用此类型检查函数来自动地进行安全性证明的构造。为了突出构造证明方法在 FPCC 系统中的应用，本节给出的是一个经过简化的 FPCC 系统，并略去了 FPCC 系统中的部分定义和证明的细节，完整的 FPCC 系统细节参见 Coq 实现代码。

3.1 可执行代码与安全性证明

一个简单的机器模型见图 3。此机器模型略去了数据堆与内存操作的指令，目标机器 M 仅由一个代码堆 C 和一个机器状态的描述 S 组成；代码堆 C 是从地址标号 f 到指令 i 的映射；机器状态 S 由寄存器文件 R 以及程序计数器 pc 组成；寄存器文件 R 是从寄存器名称 r 到数值 w 的映射；程序计数器 pc 是指向代码地址的寄存器。目标机器在运行内存中的代码之前的状态称为初始状态，这时候寄存器文件与程序计数器全部置零，分别用 R_0 与 pc_0 表示。 $M \hat{I}^n M'$ 表示机器 M 向前执行 $n(n>0)$ 步。下面定义代码的安全性 with FPCC 代码。

定义 1(Safety). 如果加载代码 C 的目标机器可以一直不出错地执行下去，那么认为代码 C 是安全的。这是此 FPCC 系统的安全策略，在 CiC 中表示为：

$$\text{Safety}(C) @ \forall n > 0. \exists M \wedge ((C, (R_0, pc_0)) \hat{I}^n M')$$

定义 2(FPCC package). FPCC 代码是一个依赖对，第一个分量是可执行代码；第二个分量是机器安全运行的证明，它的类型是一个依赖和类型(Dependent Sum Type)。在 CiC 中表示为：

$$\text{FPCC} @ \langle C, \text{Proof} \rangle \quad \text{FPCC} : \Sigma C. \text{Safety}(C)$$

3.2 TAL推理系统

本节将要介绍的推理系统是一个经过简化的 TAL 类型系统，它只包含整型与代码指针类型，不包含多态类型以及数据指针类型等等。TAL 的类型语法定义与部分推理规则见图 4。

(值类型)	τ	::=	int code(Γ)
(寄存器文件类型)	Γ	::=	$\{r_1 : \tau_1, r_2 : \tau_2, \dots, r_8 : \tau_8\}$
(代码规范)	Ψ	@	$\{f \ a \ \Gamma\}^*$

$\frac{\Psi \cdot (C, (R_0, pc_0))}{\Psi \cdot \text{Safe } C}$ (safe)	$\frac{\Psi \cdot C \quad \Psi \cdot R : \Gamma \quad \Psi \cdot C \cdot \{\Gamma\} \text{ pc} :: C(\text{pc})}{\Psi \cdot (C, (R, \text{pc}))}$ (mach)
$\frac{\forall r \hat{\Gamma} \text{ dom}(\Gamma). \quad \Psi \cdot R(r) : \Gamma(r)}{\Psi \cdot R : \Gamma}$ (rfile)	$\frac{\forall f \hat{\Gamma} \text{ dom}(\Psi). \quad \Psi \cdot C \cdot \{\Psi(f)\} f :: C(f)}{\Psi \cdot C}$ (cdhp)

Fig. 4 TAL syntax and inference rules (partly)

图 4 TAL 类型语法定义与推理规则(部分)

在 TAL 类型系统中，值类型 τ 包括整数类型 int 与代码指针类型 code(Γ)；寄存器文件类型 Γ 是从寄存器 r 到值类型 τ 的映射；代码规范 Ψ 是从地址标号 f 到寄存器文件类型 Γ 的映射。具有代码指针类型 code(Γ) 的值是指向一段指令序列的地址，而这段指令序列的前条件为 Γ 。 Γ 是某个时刻寄存器文件中所有寄存器类型的集合，可以被看作是 TAL 系统中的前条件(Pre-condition)。代码规范 Ψ 是代码中所有前条件 Γ 的集合。

TAL 类型系统的推理规则(safe)规定了可执行代码是否安全取决于加载指令代码后的初始机器 $(C, (R_0, pc_0))$ 是否是良类型的。规则(mach)规定了一个机器的良类型性需要满足 3 个前提：(1)代码堆 C 的类型为代码规范 Ψ ；(2)当前寄存器文件 R 的类型为 Γ ；(3)即将执行的指令序列的类型为 Γ 。规则(rfile)规定了 R 的类型是 Γ ，当且仅当 R 中的所有值的类型分别对应 Γ 中的类型。规则(cdhp)的含义是 C 中所有指令序列的类型分别对应 Ψ 中的 Γ 。指令序列的类型规则表示为 $\Psi \cdot C \cdot \{\Gamma\} f :: C(f)$ ，这部分类型规则可参考传统的 TAL 系统^[14]或者 Coq 实现代码，这里不再赘述。TAL 可靠性的证明思路是“可靠性=前进性+保持性”。

定理 2(TAL Soundness). 如果一个代码堆 C 是良类型的，那么代码堆 C 满足安全策略，在 CiC 中表示为：

$$\forall \Psi. \forall C. ((\Psi \cdot \text{Safe } C) \rightarrow \text{Safety}(C))$$

证明: 先证明前进性，再证明保持性，最后对机器执行的步长进行归纳。

通过 TAL 类型规则就可以证明可执行代码的良类型性，然后再根据定理 3 就可以得到代码的安全性证明。但是在没有基础逻辑之外的工具的情况下，构造这样的证明需要人工指导给出应用类型规则的步骤；这样最后的证明项的规模也会随着代码的长度而大幅度增大。应用本文的证明构造方法将会避免这些问题。

3.3 TAL类型检查器

上节的 TAL 类型系统存在可判定的类型检查算法，因此在 TAL 中构造安全性证明的过程是可算法化的。根据第 2 节提出的方法，可以使用递归函数 TALCheck 来代替 TAL 类型规则来构造代码的安全性证明。TALCheck 函数需要两个参数：代码规范 Ψ 和代码 C ，函数的返回值是布尔值 true 或者 false，定义如下：

```

fix ick( $\Psi, \Gamma, f, C$ ) @ case  $C(f)$  of
  | addu  $r_d \ r_s \ r_t \ \mathcal{P} \dots$ 
  | move  $r_d \ r_s \ \mathcal{P} \dots$ 
  | ...  $\mathcal{P} \dots$ 
end

fix cck( $\Psi, C, \Psi'$ ) @ case  $\Psi'$  of
  |  $\emptyset \ \mathcal{P} \ \text{true}$ 
  |  $\{f \ a \ \Gamma\} \hat{\Gamma} \ \mathcal{P} \ \mathit{ick}(\Psi, \Gamma, f, C)$  and  $\text{cck}(\Psi, C, \Psi')$ 
end

```

TALCheck(Ψ, C) @ cck(Ψ, C, Ψ) **and** ick($\Psi, \Psi(pc_0), pc_0, C$)

这里的 **and** 表示布尔运算中的“与”运算符。TALCheck 的核心内容是对指令序列进行递归检查。可以看出，TALCheck 函数本身的长度固定，但是它可以检查的任意大小的代码堆。

由第 2 节可知，利用递归函数构造证明项需要递归函数的可靠性证明：

引理 1. 通过 TALCheck 函数检查的代码一定在 TAL 系统中是良类型的：

$$\forall \Psi. \forall C. ((\text{TALCheck}(\Psi, C) = \text{true}) \rightarrow (\Psi \cdot \text{Safe } C))$$

Table 1 Coq implementation code

表 1 Coq 实现代码

模块	代码(行)	定义	引理或定理
机器模型	1,095	26	24
类型系统	8,195	60	279
类型检查器	2,155	14	28
其它辅助	6,303	124	324
合计	18,671	225	666

Table 2 Test results of an example

表 2 一段简单代码的安全性证明的测试数据

	传统方法构造的证明	利用递归函数的证明	
		证明	递归函数
证明项规模	84,906B	766B	63,627B
证明检查时间	5.223s	1.096s	54.339s
证明检查内存	34,948KB	20,089KB	

证明: 展开 TALCheck 函数的定义, 然后对代码规范 Ψ 与代码 C 进行双重归纳。

定理 3(Type Checking Soundness). 通过 TALCheck 函数检查的代码一定是安全的代码:

$$\forall \Psi, \forall C. ((\text{TALCheck}(\Psi, C) = \text{true}) \rightarrow \text{Safety}(C))$$

证明: 由定理 2 和引理 1 立即可得。

3.4 安全性证明的构造

本节利用上一小节得到的 TALCheck 函数的可靠性证明项来构造代码的安全性证明。使用 Prf_{TH3} 表示定理 3 的证明项, Prf_{TH3} 的类型是 $\forall \Psi, \forall C. ((\text{TALCheck}(\Psi, C) = \text{true}) \rightarrow \text{Safety}(C))$ 。 Prf_{TH3} 接受三个参数: 第一个参数是代码规范 Ψ , 第二个参数是代码 C , 第三个参数是代码通过类型检查的证明项。 Prf_{TH3} 函数的返回结果是 $\text{Safety}(C)$ 命题的证明项, 正是代码要携带的证明项。由第 2 节可知, Prf_{TH3} 函数的第三个参数可以是命题 $\text{true}=\text{true}$ 的证明项, 即(refl_equal true)。当 Prf_{TH3} 接受前两个参数后 Ψ 与 C 后, CiC 的类型检查器就能自动推断出第三个参数的类型 $\text{TALCheck}(\Psi, C) = \text{true}$ 。最后可以构造代码的安全性证明项:

$$\text{Prf}_{\text{TH3}} \Psi C (\text{refl_equal true}) : \text{Safety}(C)$$

对于任意的代码 C , 只有它能够通过类型检查函数的检查, 上面的安全性证明项才是有效的。可以看到, 上面的证明项的规模不会跟随代码规范 Ψ 和代码 C 的规模增长。而且此证明项的形式与代码规范 Ψ 和代码 C 无关, 是一个固定且通用的证明项, 因此代码生产方可以自动产生携带证明的代码。此安全性证明项是基础的, 即代码消费方仅使用 CiC 类型检查器就可以检验代码的安全性。

4 实现

我们在 Coq 中实现了一个完整的 FPCC 系统, 包括了本文介绍的所有内容, 其结构与代码量见表 1。Coq 实现代码中定义了一个接近真实 MIPS 架构的机器模型和一个较为复杂的 TAL 类型系统, 它包括多态类型, 指针类型等等; Coq 实现代码中的 TAL 类型检查器的定义与可靠性证明也较为复杂, 它涉及一个多态类型的合一算法(Unification Algorithm)。因此, 这个 FPCC 系统可以用来证明实际的汇编程序。

Coq 针对不同方法构造出的安全性证明进行检查的资源消耗情况比较见表 2。可以看出, 虽然递归函数本身的规模占用的资源并不小, 但是它不会再随着代码的规模而变化。与代码相关部分的证明, 从规模上远小于使用传统方法构造的证明(100:1); 虽然递归函数会在证明检查期带来一些额外的计算开销, 但是检查时间上仍占有优势(5:1), 这是因为这些计算仅涉及较少的内存操作; 从证明检查期的内存占用情况来看, 两者差距不大, 利用递归函数构造的证明稍小一些(3:2)。当代码的规模较大时, 使用传统的方法构造证明极其困难, 因此也难以进行测试对比, 但是我们相信本文提出的方法构造出的证明无论从规模上、构造难易程度上还是从检查开销方面都具有明显的优势。

5 相关工作比较

与本文的 FPCC 的系统相比, Appel 等人的 LTAL 系统多出了额外的没有经过验证的 Prolog 风格解释器。LTAL 的基础逻辑证明检查器是一个 C 语言程序, 而本文中 FPCC 系统的证明检查器是 Coq 系统中的编译器。

本文的 FPCC 系统采用了为代码构造显式的证明项的策略，而不是简单地在代码消费方进行类型检查。这种策略的一个优点是这些显式的基础性证明项可以与其它的 FPCC 模块进行代码与证明的链接，从而构成了完整可执行代码的安全性证明。

Crary 等人的 TALT 系统中的类型系统是一个不可判定的类型系统，因此使用本文的证明构造方法只能进行半自动构造证明，而无法完全自动化，而 TALT 采用 Prolog 风格解释器的优势在于对不可判定的类型系统仍然可以自动化地进行证明构造。TALT 系统采用的 Prolog 风格的解释器理论上也可以产生对应的证明项，从而此解释器可以被排除出 TCB。但是这样产生的证明项将庞大无比且难以检查，因而无法实际应用。

Necula 采用了一种 Oracle-bits^[15]的技术来压缩证明项。Oracle-bits 是一些数字的序列，它记录了在产生证明的过程中哪些中间子目标是一定是失败(fail)的。这些 Oracle-bits 被附加在可执行代码上传输到代码消费方，代码消费方使用同样的一个证明工具来证明代码，通过 Oracle-bits 使得证明器避免多余回溯搜索。研究如何将这种证明压缩的方法与本文的方法相结合是我们未来的工作之一。

6 结束语

本文提出了一种利用辅助递归函数来构造代码的安全性证明的方法，此方法利用递归函数代替推理系统中部分或全部的推理规则来构造代码的安全性证明，在不增加系统信任计算基础的前提下，可以提高证明构造过程的自动化程度并可以降低证明的复杂性。此方法适用于各种 FPCC 系统中，研究如何将此证明构造方法应用在更多的领域也是我们未来工作的一部分。

References:

- [1] Necula G. Proof-carrying code, In: Proceedings of POPL'97. New York: ACM Press, 1997. 106-119.
- [2] Appel AW. Foundational proof-carrying code, In: Proceedings of 16th Annual IEEE Symposium on Logic in Computer Science. Washington: IEEE Computer Society, 2001. 247-258.
- [3] Chen J, Wu D, Appel AW, and Fang H. A provably sound tal for back-end optimization. In: Proceedings of PLDI'03. New York: ACM Press, 2003. 208-219.
- [4] Crary K. Toward a foundational typed assembly language. In: Proceedings of POPL'03. New York: ACM Press, 2003. 198-212.
- [5] Yu DC, Hamid NA, and Shao Z. Building certified libraries for PCC: Dynamic storage allocation. In: Proceedings of 2003 European Symposium on Programming, LNCS 2618. Springer-Verlag, 2003..
- [6] Feng XY, Shao Z, Vaynberg A, Xiang S, and Ni ZZ. Modular verification of assembly code with stack-based control abstractions. In: Proceedings of PLDI'06. New York: ACM Press, 2006. 401-414.
- [7] Feng XY, Ni ZZ, Shao Z, and Guo Y. An open framework for foundational proof-carrying code. In: Proceedings of TLDI'07. New York: ACM Press, 2007. 67-78.
- [8] Wu D, Appel AW, and Stump A. Foundational proof checkers with small witnesses. In: Proceedings of. PPDP'03. New York: ACM Press, 2003. 264-274.
- [9] Crary K and Sarkar S. Foundational certified code in a metalogical framework. Foundational certified code in a metalogical framework. In: Automated Deduction, volume 2741 of LNCS. Berlin: Springer, 2003. 106--120
- [10] Pfenning F. Logic programming in the LF logical framework. In: Girard Huet and Gordon Plotkin, eds, Logical Frameworks. Cambridge: Cambridge University Press, 1991. 149-181.
- [11] Paulin-Mohring C. Inductive definitions in the system Coq-rules and properties. In: Proc. of TLCA, volume 664 of LNCS. Berlin: Springer, 1993. 328-345.
- [12] Lin CX, Chen YY, Li L, and Hua B. Garbage collector verification for proof-carrying code. JCST, 22(3):426-437, May 2007.
- [13] Guo Y, Jiang XY, Chen YY, Lin CX. A certified thread library for multithreaded user programs. In: Proc. of TASE'07. Washington: IEEE Computer Society, 2007, 127-136.
- [14] Morrisett G, Walker D, Crary K, and Glew N. From system F to typed assembly language. In: Proc. of POPL'98. New York: ACM Press, 1998. 85-97.
- [15] Necula G, Rahul S. Oracle-based checking of untrusted software. In: Proc. of POPL'01, New York: ACM Press, 2001. 142-154.