计算机科学2002Vol. 29№. 11

基于语言的移动代码安全问题*)

Language-Based Mobile-Code Security

郭 帆 陈意云 胡荣贵

(中国科学技术大学计算机系 合肥230026)

Abstract With the development of Internet, traditional network is converting to active network computing, which makes the problem of mobile-code security emerge as one of the most challenges of research in Computer Science today. The literature introduces one method that solves the problem of mobile-code security: Language-Based Security, and decribes in detail several methods in this research area, such as Proof-Carrying Code(PCC) and Typed Assembly Language(TAL). Our resolvation of mobile-code safety is presented at last.

Keywords Mobile-code, Code safety, Safety policy

1 引言

在我们越来越依赖于全球信息基础构造的同时,我们发现自己对恶意攻击和对有漏洞软件的破坏的抵抗能力也越来越弱。虽然类似梅莉莎和爱虫之类的程序已经在世界范围造成了很大的灾难,但是,人们依然在不顾后果地下载并运行plug-in 程序。主动网络概念的出现,使传统网络正快速从被动地传送字节流向更一般的网络计算转变,移动代码的安全问题越来越成为计算机科学领域要面对的挑战之一。

如果本地主机允许移动代码被动态安装并执行,主机系统可以提供一种灵活的方式让移动代码访问主机内部资源。但是,为了保证移动代码是可安全执行的,必须确保移动代码不会破坏主机内部的资料,确保移动代码不会耗费主机太多的资源,确保执行该程序不会降低主机的运行性能。如何得到这些保证呢?

在这里,我们介绍一种特殊的方法来解决移动代码的安全问题,叫做基于语言的安全(language-based security)。首先,我们给出该方法的一个概述,然后讨论这一领域中正在研究的几种方法:携带证明的代码(Proof-Carrying Code,简称PCC)[1~9],类型化的汇编语言(Typed Assembly Language,简称 TAL)[10~14,18]和其它一些研究,最后提出了我们的一些想法。

2 移动代码安全问题概述

理想的移动代码是一种具有智能的可执行代码,它具有跨地址空间持续运行的能力,可以在网络中自主迁移到不同主机上去执行代码的不同部分。它具有智能性、持续性和移动性等特点。代码的移动性使代码安全成为一个重要问题。移动代码的安全包括两个方面:(1) 从代码的角度看,当它在网络中不同主机上持续执行时,它会在不同主机之间进行传输,因此它需要防止恶意主机对自己的修改,以免恶意主机利用它对网络中别的主机进行攻击;(2) 从主机的角度看,当移动代码迁移到本地时,如何保证它的执行不会破坏自己的系统资料。安全问题的第一方面涉及到人工智能领域。本文讨论的移动代码安全是指其第二个方面。

2.1 安全策略

若希望从一个未知的不受信任的站点移动一个程序到本地运行,在运行该程序之前,希望能保证该程序是可以安全运行的。但是,怎么样才算是安全的呢?在不同的环境下,安全可能有不同的解释。例如,可能期望程序永远都不会覆盖关键系统资料。否则,程序会导致系统崩溃;可能期望程序不会访问在同一物理地址空间中分配给其它进程的内存资料,否则会造成资料泄密;可能期望拒绝程序所有的磁盘 I/O(这是对付从网上下载的 Java 小程序的缺省安全策略),或者只允许有限的 I/O。在这里,我们将安全策略定义为:描述主机安全的规则集合,每条规则说明了移动代码对各种主机资源的访问权限和限制条件,资源包括内存、磁盘、CPU 等。

为了保证可疑代码在本地安全执行,制订的安全策略至少要保证如下的基本安全性质:(1)控制流安全。程序的跳转指令和调用指令所指向的地址,必须是在程序代码段的地址之中,而不能跳到别的位置上。所有的调用都应指向合法的函数入口并且所有的返回地址都处于调用该函数时的内存位置;(2)内存安全。程序只能访问其静态资料区范围内的资料,或者是系统分配给它的堆上的资料,或者是合法的栈帧上的资料,而不能随意访问内存中的其它位置;(3) 栈安全。对于基于栈式分配的语言实现,运行时栈应该在函数调用的过程中被保存下来。这三个基本的安全性质是相互独立的。它们构成了最基本的安全策略,任何有意义的安全策略都会包括这三个基本性质。在这些基本安全策略之上,可以进一步扩展安全策略。例如,可以设置来自某些 IP 地址的程序不能读写磁盘,来自某些地址的程序可以使用打印机等方法,对移动代码读写磁盘和使用外部设备的权限做出限制。总之,安全策略是由代码接收方定义,它的复杂性取决于接收方对安全的需求程度。

许多文章还提到了类型安全,它的前提是要存在一种定型规则。定型规则用来确定代码和资料的类型。前面所提到的控制流安全,内存安全和栈安全都可以包含在类型安全中,因为这些性质都可以编码到定型规则中。TAL 使用这种方法(见第5节)。

2.2 信任关系

*)本项目得到国家自然科学基金资助(项目编号60173049)。郭 帆 胡荣贵 博士研究生,主要研究方向为程序设计语言理论和实现技术、软件系统结构。陈意云 教授,博士生导师,主要研究方向为形式化方法、程序设计语言理论、软件系统结构。

代码可以分成两类，可信代码和可疑代码，用一种虚拟的信任边界分开，如图1所示。所有的软件安全机制都是建立在可信代码集之上的。可信代码集至少包括本地操作系统内核和编程语言的一些运行时的支持(前提是它们没有被破坏)。应采用尽可能小的可信代码集，因为需要信任的代码越少，抵抗力就会越强。

用一种类型安全的语言编写程序并使用可信编译器，可以保证控制流安全、内存安全以及栈安全等。可以由发送方把源码发给接收方并且让接收方去编译该源码，或者是接收方必须信任发送方编译器以及传送目标代码的信道，这两种方法都不能令人满意，前者是性能上的问题，后者是安全上的问题。另外它们的共同缺点是，编译器必须属可信代码集。而本文讨论的基于语言的方法，可以在无需信任编译器的情况下达到同样的目标。

2.3 传统方法

传统的解决安全问题的方法包括内核监督，代码检测以及加解密方法等。这些方法给出了基本安全策略的一个固定集合，缺乏灵活性。

内核监督可能是软件系统中最广泛使用的安全机制。它把访问关键系统部件和资料的操作独立开来，放在系统内核中。所有的进程只能通过内核以有限的方式访问这些关键资料。这不仅可阻止可疑代码对系统的破坏，而且允许内核监视所有的访问、完成验证以及强加其它的安全策略等。但是，通过内核调用，访问被限制在内核接口提供的几种高级抽象操作中。若允许所有的非内核进程直接访问关键的系统部件和资料可以极大地提高性能，例如，可以使用许多利用低级数据结构的优秀算法，而且，内核调用通常有上下文切换的开销，而直接访问不需要。人们期望找到一些方法使得在不牺牲安全性的前提下，允许所有的非内核进程直接访问关键资料。

代码检测是指对机器码进行修改使关键操作可以在执行时被监视。它使用如下策略：(1)若源代码没有违背安全策略，修改后代码的函数行为与源代码的一致；(2)如果源代码违背了安全策略，修改后代码运行时将检测到对安全策略的违背，让系统截获控制权并终止该错误进程，或者阻止该违背在系统其它部分产生不良后果。

代码检测的优点是，接收方可以在不对代码做任何假设以及无需有关代码的任何附加信息的情况下，独立地执行代码检测。但是，对任意代码通过代码检测方式强加安全策略，代价将非常高。在每个敏感操作之前都必须执行运行时检查，会使得运行时开销过大。对程序进行分析后可以删除一些不必要的检查，但是这种分析的代价也不小，还会增加装入代码时的开销。此外，即使是最优秀的分析技术也是不完善的，因为安全性质通常是不可判定的。

加解密方法是指在数据传输过程中对资料进行加密，在接收时对其进行解密，从而防止资料的泄露。但是，当前加密协议的安全性依赖于一些尚未证明的复杂性理论的假设。另外，加解密方法是建立在代码可信的基础上。

2.4 基于语言的安全

Schneider[24]将基于语言的安全定义为使用基于编程语言的理论和实现的技术，包括语义、类型、优化和验证等，来解决安全问题。

编译器通常会收集许多关于程序的信息，比如变量的类型信息或对变量值的其它限制，还有结构信息、命名信息等。这些信息在分析程序时得到，并被用来检查类型正确性和进行代码优化等。编译后这些信息往往被丢弃，只剩下指令序列。然而，这些信息在考虑目标代码的安全时可能会很有用处。例如，用类型安全的语言编写的程序在编译时必须成功通过类型检查，若编译器可信的话，任何成功通过类型检查的源程序所编译成的目标代码必定是内存安全的。如果代码接收方可以得到这些信息，那么判定移动代码是否可安全执行就有可能变得相对容易。

我们用“基于语言的安全”来表示这样一种方法：从编译后的目标代码中获取前面所说的有关源程序的信息。这些信息，称之为“证书”，是在编译时被创建，附属在目标代码之中。当目标代码被移动时，证书也被移动。接收方则运行一个验证器，它检查代码和证书，验证代码是否符合某种安全策略。若验证成功，那么代码可安全执行。验证器必须是接收方可信代码集的成员，编译器、目标代码以及证书可以不是(见图1)。

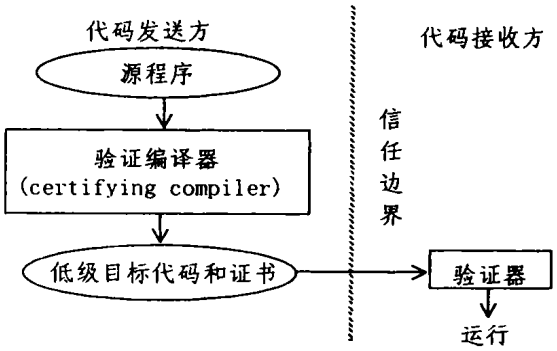

图1 基于语言的方法

这种方法最大好处在于保证代码满足安全策略的责任从代码接收方转移到发送方。发送方必须提供一个证书，该证书给出足够信息使得验证目标代码是否满足安全策略成为可能。而接收方从证明代码是否安全简化到检查代码是否安全，其工作量极大降低。

证书可以采用不同的形式。例如，PCC 的证书是验证条件的一阶逻辑证明，其验证过程是检查该证书是否为有效的一阶证明；TAL 的证书是类型注解，验证过程是类型检查。虽然各种方法在表达能力、有效性和灵活性上有差别，但它们有共同的目标：利用编译时产生的额外信息来帮助解决可疑的移动代码在本地安全而有效执行的问题。

3 携带证明的代码

携带证明的代码[1~9]是指允许安全条件形式证明在代码运行前产生和验证的一种方法。PCC 的关键思想在于代码发送方创建一个形式化的“安全证明”来证实代码是否遵照预先定义的安全条件。然后，代码接收方使用一个简单快速的证明验证器来检查证明过程是否正确，从而判断代码是否可以安全执行。

通常，PCC 的运行包括四个阶段[1]：

1. 发送方产生一个带有注解的目标代码并发送给接收方，注解包括循环不变式和函数的前后条件等。这些注解帮助产生关于代码安全的证明。

2. 接收方(已经定义了一个安全策略)，根据接收到的代码产生验证条件并发还给发送方。验证条件是一系列的一阶逻辑公式，它们蕴涵程序满足安全策略。安全条件可依据不同的语言结构和机器结构做相应的扩展。

3. 发送方综合得到的验证条件，运行一个定理证明器，产

生关于该程序是安全的形式证明，再把该证明发送给接收方。

4. 接收方依据已有的验证条件，运行一个证明检查器，检查该证明是有效的。

相应地，一个 PCC 的实现至少包括4个部分[6]：(1)用于表达安全策略的形式规范语言；(2)可疑代码使用的语言的形式语义，指源程序与安全策略的形式规范之间的关系逻辑(就是如何产生验证条件)；(3)用来表示证明的语言；(4)验证证明是否正确的算法。

第一阶段通常是由一种称为验证编译器的工具来执行的。编译器使用分析程序得到的信息来构造循环不变式和函数的前后条件。该过程大部分是自动的，偶尔需要人工干预，取决于安全策略的复杂性。函数的前条件应该提供足够的信息使得下一阶段可以为该函数提供相应的验证条件。验证条件蕴涵函数满足安全策略并且满足其后条件。

TouchStone 编译器[3]是一个验证编译器，实现了一个类型安全的 C 语言子集。它的一个主要优点是容许许多普通的优化，如死代码删除、公共子表达式删除、复写传播、指令调度、寄存器分配、循环不变计算外提、冗余代码删除以及数组越界检查的删除等。SpecialJ[5,8]是实现 Java 子集的验证编译器，它把 Java 的字节码转换为 IntelX86体系结构下的机器码。目前，SpecialJ 仍然处于开发之中。

第二阶段产生的验证条件是安全策略的形式描述在特定程序上的实例化，即验证条件是程序相关的。独立于程序来表达安全策略是极其困难的，也没有必要。George C. Necula 和 Peter Lee[1]提出使用一阶逻辑规范来表达安全策略，并且使用 Floyd 风格的验证条件产生器来产生验证条件。接收方扫描待验证代码的每一条指令来产生一阶逻辑的谓词作为验证条件，并且还提供一些证明公理给发送方。该谓词随后由发送方利用给定的定理证明器和接收方提供的公理给出该谓词的证明，然后将证明发回给接收方。SpecialJ 正是假设接收方是这样做的[9]，并且它自己定义了相应的与接收方相同的安全策略和相同的验证条件产生器，从而可自己产生验证条件和相应的证明过程，省略了第二阶段。如果双方的安全策略并不一致的话，那么接收方会发现证明过程与自己的验证条件并不匹配，从而拒绝执行该代码。

第三阶段所做的事情已经和编程语言及源程序无关了，而仅与表达安全策略的规范有关。目前，Edinburg 逻辑框架(简称 LF)[21]被用做表示验证条件及其证明的语言。LF 是一种元语言，用作高级的逻辑规范，本质上它是一种类型化 λ 演算，它把证明表示成表达式，把谓词表示为类型。使用 LF 的一个优点是可以使用 LF 的类型检查机制，通过检查表达式的类型来验证证明是否正确。TouchStone 采用 LF 来表示证明，它扩展了 Nelson 和 Oppen 提出的组合判定过程[22]的方法来实现定理证明器，增加了处理相等性的同余闭包判定过程和处理线性计算的 Simplex 判定过程[7]。

第四阶段的验证可以利用 LF 的类型检查机制，如果采用 LF 来表示证明的话。LF 的类型检查机制非常简单，通常只需要一到二百行程序就可以解决问题，这得益于 LF 类型系统的简单。

在证明产生阶段必须考虑证明的长度，在验证阶段必须考虑证明的验证时间。这两者是密切相关的，如果证明太长的话，验证时间相应地加长，这是难以接受的。因为，验证过程是在代码向接收方移动时执行，如果等待时间太长，代码移动可能会被终止。不幸的是，虽然 LF 比较简单，但是由它表示的证明中，冗余非常多。把 LF 扩展成 LF$_i$，用以处理隐式子项，把 LF 类型检查算法扩展到能合成缺少的子项，可以得到较好的结果[4]。

PCC 使得移动代码安全的责任从接收方转移到了发送方。接收方只需做少量工作即可认定移动代码是否安全，这有利于那些内部资源不很充足的便携式设备下载代码。PCC 的另一个优点是，它不受干扰的影响。也就是说，如果代码、证明或验证条件在传输过程中被恶意改动或由于网络故障而丢失部分资料，也不会影响代码的安全执行。因为，接收方只根据自己产生的验证条件和发送方发来的证明进行验证，以判断代码是否满足安全策略。如果代码改动导致不满足安全策略的话，则拒绝执行；如果仍满足，则说明相应改动并未破坏安全策略，仍然可安全执行，只是代码的语义与原来的可能不一样。PCC 的缺点是证明的产生和验证相对 TAL 以及其它一些方法来说代价稍微大一些，但是可以忍受，实际上 SpecialJ 就是为了商业用途而开发的。

4 类型化的汇编语言

类型化的汇编语言是一种基于语言的系统，在编译强类型语言的程序时，程序中的类型信息通过一系列的转换依附在目标代码之中。产生一种带类型注解的目标代码。同时，必须有一个类型检查器对这种目标代码进行静态检查，判断目标代码的语义与施加的类型限制是否一致来决定目标代码是否安全。

TAL 的理想目标[13]是提供一种低级的、静态的类型化目标语言，使之比 Java 字节码更加适合于支持众多的源语言以及大量的优化。

最初，TAL 是通过对一种类 ML 的多态类型抽象语言进行一系列变换而得到的结果[10]，这些变换被称为保类型变换，最终形成的 TAL 是基于传统的 RISC 汇编语言。但是，TAL 的静态类型系统加强了高级的语言抽象，例如闭包、元组以及用户自定义的抽象数据类型等。TAL 的类型系统保证类型正确的程序不会违反这些抽象。而且，TAL 的类型系统不会阻碍如寄存器分配，指令选取和指令调度之类的低级优化。

产生早期 TAL 版本的变换中包括了后续传递语义(CPS)变换，CPS 使用堆分配的活动记录取代栈分配的活动记录。堆分配与传统的栈分配的实现相比有不少优点[11]：易于实现一些控制原语如异常，一阶后续[15]等；Appel 和邵中[16]指出堆分配可以节省空间，因为它更容易共享环境；堆分配有统一的内存管理机制(垃圾回收)来分配和释放各种对象，包括活动记录。但是有两个重要的因素使得提供对栈的支持很有必要[11]：基于栈的活动记录所占用空间比基于堆的活动记录要小；许多芯片的体系结构设计要求应用程序使用基于栈的方法，例如，奔腾 Pro 芯片内部有一个用来预测过程返回地址的内部栈[17]，它来源于使用标准控制栈的调用/返回原语。因此，TAL 的后续版本添加了栈，它的文法请参见文[11]。

Neal Glew 和 Greg Morrisett[12]指出将 TAL 扩展成 MTAL(modula TAL)，并在每个目标文件的开始处添加引入和导出接口，使得连接时的类型检查与目标文件自身的类型检查分离开来，如果检查得出各个文件的接口是良类型的，则可以保证目标代码在连接时的良类型性质。

如果不采取任何优化手段的话，TAL 中类型注解的大小

超过实际执行代码的大小，这使得 TAL 还不能适合大规模工业应用。Dan Grossman 和 Greg Morrisett[14]提出使用 Bake it in 方法，“不优化”方法，以及重构和压缩算法等，可以得出相当精简的注解，最后形成的注解大小大约为代码总量的 10%。

目前，TAL 已实现的版本叫做 TALX86软件包[13]。它定义了类似于 C 的高级语言 Popcorn。我们首先使用 Popcorn 编译器将 Popcorn 源程序编译成带有注解的汇编码文件（以 .tal 为后缀）以及相应的引入和导出接口文件（后缀名分别为 ._i_tali 和 ._e_tali）；然后使用类型检查器 talc 检查目标文件的良类型性质，以及目标文件连接的良类型性质；如果目标文件是良类型的，则 talc 除去目标文件中的注解，并调用 MS 的 MASM6.0产生可执行文件；否则，报错。

但是，TAL 还不完善，还需要进一步的扩展。目前 TAL 类型系统的一个重要缺点是，不能有效地释放内存对象，而是依靠一个垃圾回收程序来回收对象，从而阻止了对内存的及时重用，而这种重用在类型化低级语言中是很关键的。Frederick Smith 和 David Walker[18]提出使用指针别名的方法，将指向内存位置的指针也作为一种类型加入类型系统，可以保证在释放内存对象时不会破坏 TAL 代码的良类型性质。但是，这样做需对现有的类型系统做很大的变动，因此目前还没真正使用这一扩展。其它处于研究之中的对 TAL 的扩展还包括，删除数组越界检查[19]和运行时的代码生成[20]。

TAL 与 PCC 相比，不如 PCC 表达得那么清楚，但是，TAL 可以处理所有可用类型系统的术语表示的安全策略，包括了内存安全，栈安全和控制流安全等基本安全策略。并且，TAL 在面对编译优化时非常健壮，因为它的类型注解是与代码一起变换的。最后，Dan Grossman 和 Greg Morrisett[14]证明了 TAL 是完全适合于大规模工业应用的。

5 其它研究

Dextor Kozen[24]提出一种有效代码验证（ECC）的方法，它在保证类似小应用程序这样的可疑的并只运行一次的应用程序的安全的同时，改善运行时的效率。它给代码添加的注解提供了代码的结构和意图，以及一些基本的类型信息。ECC 的注解易于产生，易于验证，并且它在目标代码级进行操作，因此效率较高。其缺点是依赖于平台和依赖于编译器的实现。

基于语言的方法能够用于互不信任的机器之间传递控制信息流。这里，安全策略是基于信息流模型的[26]，用户在高级语言中增加注解来限制信息在程序内和程序间流动。加注解的程序在编译时被检查，保证程序遵循流规则。目前已经有一个原型实现 Jflow，它给 Java 增加了流原语。Jflow 是一个源语言到源语言的翻译器，它使用类型检查机制来检查信息流安全，然后丢弃注解，还原为普通的 Java 程序。如果这些控制信息传递到目标代码，则可以用与 TAL 和 ECC 类似的方法进行验证。

FLINT 项目[25]致力于为高阶的类型化语言构造一个通用的编译器架构。他们希望使用一个类型富有的类型化中间语言，作为编译其它语言的目标语言。目前，FLINT 已经作为 SML/NJ 编译器的中间语言而广泛使用。

Scheider 对传统的代码检测的方法进行了扩展，提出“安全自动机”的概念[24]。可以处理所有可以由有限自动机表示的安全策略。代码检测使得任何可能影响安全自动机状态的指令都必须在对该自动机调用之后执行。安全自动机使安全策略的规范变得很灵活，并且允许按照代码接收方的特定需要来构造安全策略。它的主要缺点是在运行时对自动机的模拟存在一定的开销。

6 我们的方法

虽然，目前在使用基于语言的技术解决移动代码的安全问题上展开了众多的研究，并且形成了以 PCC 和 TAL 为代表的两种解决方案，但是，这些研究所面向的代码安全策略还处于比较低级的层次上面。目前都仅仅能够处理类型安全和内存安全等策略。那么，如何实现一种安全策略来满足各个层次上的安全要求呢？我们可以采用一种三层体系结构（如图2所示）来实现这个目标。

整个安全策略由高中低三部分安全策略组成。低级安全策略主要是解决基本的代码安全，包括类型安全，内存安全和控制流安全，这里可以采用 PCC 的方法来实现。中级安全策略，可以针对每条具体机器指令来做不同的安全动作，主要是解决类型安全所不能解决的一些安全策略。例如，要控制每个程序的总的指令数，或者说不许过程中的指令访问过程栈帧之外的栈空间（即防止栈溢出）之类的安全策略。我们可以事先确定每条指令所要做的动作，也可以提供一种策略描述语言，使得可以依据不同的需要来制定不同的安全策略，从而实现极大的灵活度。通过中级安全策略，我们可以限制一个程序的最大指令数，以及可使用的堆栈空间的上限等等一些定量的限制。然后，综合分析用户定制的安全策略和接收到的移动代码，看是否违背安全策略。高级安全策略主要是用来决定代码对于一些设备和资源的访问是否是安全的。我们可以通过修改机器中的资源访问的 API 来达到这个目标，通过提供一种资源描述语言，让用户来确定哪些资源的哪些操作是不安全的，以及一旦出现这些操作，应该做出什么样的动作。例如，对于文件读操作，我们可以在高级策略中定义某个文件名是不可读的，如果读该文件则弹出一个警告提示。高级安全策略会产生一个有关文件读写的新的库，用来替换原来的 API，则当做文件读写时，调用的 API 是新的经过修改的 API，它已经满足了用户制定的安全策略。

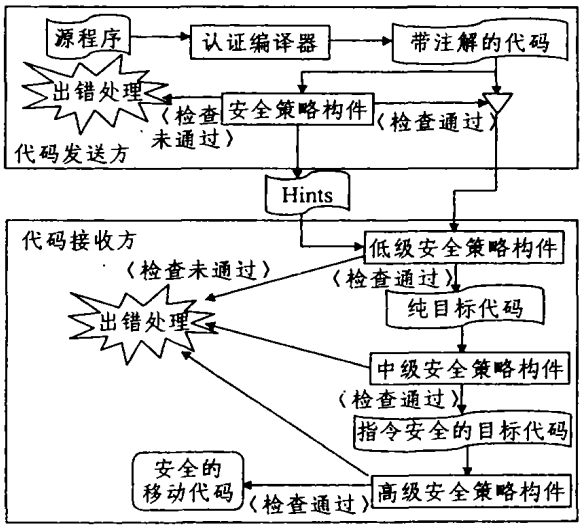

图2 三层体系结构的安全策略

我们的方法采用这样的三层体系结构，可以解决从低到高的各种安全策略。而且，由于中高级策略，是通过描述语言的形式提出的，给了用户很大的灵活性，用户可以轻松地制定

自己所需要的安全策略，从而可以满足各人的不同需要。

结束语 在使用基于语言的技术解决移动代码的安全问题上已经展开了众多的研究，但是该领域中还存在着大量的问题和挑战。

当前有不少用于表示证书信息的高级的语言构造，如PCC使用一阶逻辑证明，TAL使用类型信息等等，但除此之外是否还有其它的高级语言结构更适合用于表示证书中的信息？

当前这些研究方法所处理的安全策略仍局限于基本的安全策略，如何扩展这些方法使它们可以处理更高层次的安全策略？我们已经提出了一个解决方法的雏形，但是有无更好的方法存在？

解决代码安全问题的一个挑战在于证书的庞大，它往往是目标程序的几倍甚至是几十倍。因此，如何减少证书的大小成为一个不可回避的问题。它包括了如何有效地构造证书，以及一些压缩算法。

另外，代码接收方如何有效地验证收到的证书是和代码匹配的，也是需要研究的课题。

参 考 文 献

1 Necula G C, Lee P. Safe Kernel Extensions Without Run-Time Checking. In: Proc. of the 2nd Symposium Operationing System Design and Implementation(OSDI'96), Seattle, Oct. 1996. 229～243

2 Necula G C, Lee P. Proof-carrying code. In: Neil D. Jones, ed. Conf. Record 24th Symposium on Principles of Programming Languages(POPL'97), Paris, Jan. 1997. 106～119

3 Necula G C. Compiling with Proofs: [PhD thesis]. Carnegie Mellon University Oct. 1998. Available as Technical Report CMU-CS-98-154

4 Necula G C, Lee P. Efficient representation and validation of proofs. In: Proc. 13th Symp. Logic in Computer Science, IEEE, June. 1998, 93～104

5 Necula G C, Lee P. The design and implementation of a certifying compiler. In: Proc. Conf. Programming Language Design and Implementation, ACM SIGPLAN, 1998. 333～344

6 Lee P, Necula G C. Research on Proof-Carrying Code on Mobile-Code Security. In: Proc. of the Workshop on Foundations of Mobile Code Security, Monterey, 1997

7 Necula G C, Lee P. Proof Generation in the Touchstone Theorem Prover. In: Proc. of the 17th Intl. Conf. on Automated Deduction, Pittsburgh, 13 June 2000

8 Colby C, et al. A Certifying Compiler for Java. In: Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI00), Vancouver, British Columbia, Canada, 2000. 18～21

9 Colby C, Lee P, Necula G C. A Proof-Carrying Code Architecture for Java. In: Proc. of the 12th Intl. Conf. on Computer Aided Verification (CAV00), Chicago, July 2000

10 Morrisett G, Walker D, Crary K, Glew N. From System F to Typed Assembly Language. In: the Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, Jan. 1998. 85～97

11 Morrisett G, Crary K, Glew N, Walker D. Stack-Based Typed Assembly Language. In: the 1998 Workshop on Types in Compilation, Kyoto, Japan, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, Lecture Notes in Computer Science, volume 1473, Springer-Verlag, 1998. 28～52

12 Glew N, Morrisett G. Type-Safe Linking and Modular Assembly Language. In: the Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, Jan. 1999. 250～261

13 Morrisett G, et al. TALx86: A Realistic Typed Assembly Language. In: the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta, GA, USA, May 1999. 25～35

14 Grossman D, Morrisett G. Scalable Certification for Typed Assembly Language. In: the 2000 ACM SIGPLAN Workshop on Types in Compilation, Montreal, Canada, Sep. 2000

15 Appel A W. Compiling with Continuations. Cambridge University Press, 1992

16 Appel A, Shao Z. An empirical and analytic study of stack vs. Heap cost for language with closures. Journal of Functional Programming, 1993, 1(1)

17 Corporation I. Intel Architecture optimization Manual. Intel Corporation, P. O. Box 7641, Mt. Prospect, IL, 60056-7641, 1997

18 Smith F, et al. In European Symposium on Programming, Berlin, Germany, March 2000

19 Xi Hongwei, Pfenning F. Eliminating array bound checking through dependent types. In: Proc. Conf. Programming Language Design and Implementation, ACM SIGPLAY, June 1998. 249～257

20 Hornof L, Jim T. Certifying compilation and runtime code generation. In: Proc. Workshop on Partial Evalition and SemanticsBased Program Manipulation, ACM, Jan 1999. 60～74

21 Harper R, Honsell F, Plotkin G. A framework for defining logics. J. Assoc. Comput. Mach, 1993, 40(1): 143～184

22 Nelson G, Oppen D C. Fast decision procedures based on congruence closure. Journal of the Association for Computing Machinery, 1980, 27(2): 356～364

23 Erlingsson U, Schneider F B. SASI enforcement of security policies: A retrospective, Preprint, April 1999

24 Kozen D. Efficient Code Certification: [Technical Reporrt 98-1661]. Computer Science Dept, Cornell University, Jan. 1998

25 Shao Z. An overview of the FLINT/ML Compiler. In: Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation, June 1997

26 Andrew C. Myers. Jflow: Practical static information flow control. In: Proc. 26th Symp. Priniples of Programming Languages, ACM, Jan. 1999