第5章 软件防篡改技术

5.1 引言

软件防篡改技术是保护软件安全的途径之一，其基本目标在于阻止软件程序被非法修改，检测篡改并作出适当的响应。它有两种实现策略：

（1）采取相应机制增大篡改程序的难度，提高篡改者攻击程序的成本。

（2）检测篡改事件并采取应对措施，比如，导致程序非正常运行。

本章在总结软件防篡改领域国内外研究现状及进展的基础上分析其未解决的问题，并讨论其研究前景方向。本章按照如下结构组织：首先描述防篡改技术的分类方式，对介绍防篡改机制的文献进行分类，并设计软件防篡改技术的准则；然后详细介绍各种软件防篡改技术方法的实现模式，分析其优缺点及相应的应用场景；最后结合软件防篡改领域的研究现状，对其研究前景作出展望。

5.2 软件防篡改技术的分类方式和设计准则

本节在讨论软件防篡改技术之前先就攻击者攻击软件的方式进行概述，接着讨论防篡改技术的分类方式及其设计准则。

5.2.1 攻击类型

一般而言，按照攻击源的位置，攻击方式可以划分为以下3类：

（1）第一类攻击。该类攻击来源于软件外部，属于典型的黑客攻击方式。攻击者绕过网络的访问控制层从而入侵系统，但无法直接访问软件及其运行所依赖的平台。健壮的访问控制机制可以阻碍第一类攻击，防止攻击者从外界入侵系统。防篡改软件并不用来抵制该类攻击。

（2）第二类攻击。该类攻击同样来源于软件外部，代表类型有病毒和木马等。攻击者将非法代码嵌入目标系统，非法代码本身并不在攻击者的直接控制下，而是由操作系统运行。恶意代码的嵌入导致程序完整性被破坏。第一类攻击的目的在于发起第二类攻击，即攻击者首先侵入系统，接着向软件中嵌入恶意代码。

（3）第三类攻击。该类攻击方式无任何安全限制，攻击者完全控制软件及其运行所依赖的平台，这种攻击方式严重破坏了软件的知识产权。攻击者可以使用软硬件分析工具、调试工具和系统诊断工具等实施攻击。该类攻击的攻击力度最强，抵抗其的难度最大。

如第 5.1 节所述，软件防篡改技术阻止软件被非法修改及检测非法修改，可以抵抗第二类和第三类攻击。

5.2.2 分类方式

1. 按实现模式划分

按实现模式，软件防篡改技术分为硬件与软件两种实现方式：

(1) 硬件方式。利用安全可靠的硬件设备来防范软件被篡改。硬件负责检测软件程序的完整性，保证程序只有在运行时处于解密状态，其他时刻都是被加密的，以抵挡攻击者的篡改攻击。

(2) 软件方式。依赖软件技术抗击非法篡改行为，它可以验证被保护软件的完整性，以软件方式对被保护软件进行加解密等。

2. 按检测程序完整性的手段划分

软件防篡改技术的任务之一即检测被保护程序是否被篡改。目前有 3 种机制实现这一目标，它们之间可以相互组合以提高程序抵御篡改行为的力度。

(1) 检查程序代码本身的完整性。在程序运行之前或运行过程中检验代码的某一个特征值与原始值是否一致。

(2) 检查程序运行中间结果的正确性。这是动态行为，在程序执行过程中发生。

(3) 对程序进行加密。攻击者如果篡改被加密的程序代码，将导致其解密失败，进而无法正常运行。

3. 按实现防篡改功能的主体划分

按实现防篡改功能的主体划分，可将软件防篡改技术分为外部式和自我式。

(1) 外部式。通过采用软件本身之外的手段来实现软件的防篡改。这种技术的实现不需要被保护软件本身参与计算。典型的外部式软件防篡改技术主要用在可控的运行环境中，比如可以在可控的 Java 程序运行环境中(Java 虚拟机)，通过向其添加完整性验证模块来实现软件的合法运行。

(2) 自我式。在被保护软件中嵌入相关模块以实现防篡改功能。自我式分为静态与动态两种模式。静态模式是指程序在启动时检查其完整性，只执行一次；动态模式是指程序在运行过程中反复地验证其完整性。自我式的防篡改机制分为 3 个步骤：

① 检测。检测被保护的软件程序是否被篡改。

② 触发。若检测模块验证程序未被篡改，则程序正常运行，否则触发响应步骤。

③ 响应。使程序无法正常运行，即扰乱或中止程序的部分或所有功能。

4. 辅助手段

此外，结合其他技术方案可以增强软件防篡改的力度，如混淆技术和软件定制技术等。

5.2.3　设计准则

一个好的软件防篡改方案至少需要满足下面几条设计准则。

1. 隐秘性

隐秘性是指防篡改机制不易被攻击者探测。比如，在自我式方案中，隐秘性原则要求嵌入被保护软件的防篡改代码应该与原有代码具有高度相似性，避免被攻击者定位从而移除或绕过防篡改代码。加密方案使用密钥加解密程序代码，密钥的安全存储也是需要慎重考虑的。在基于检测-响应的防篡改机制中，响应(response)与程序故障(program failure)在时间和空间上应该分离，进一步说，在空间上，响应代码与使程序出故障的代码位于程序当中不同的模块，攻击者无法通过程序故障点反跟踪至响应位置；在时间上，程序作出响应后不立即发生异常情况，攻击者不易感知程序对其篡改行为给出的应对措施。

2. 可预测性

防篡改机制需要保证任何方式的篡改行为都被成功检测。比如，有一些高级的篡改是临时的：攻击者篡改程序导致其异常运行后，将恢复程序至原始版本状态。一旦程序被篡改，那么防篡改机制就会导致程序非正常运行，且程序发生异常的时间和位置都是可控的。在基于检测-响应的防篡改方案中，检测模块检测到程序被篡改后，响应模块要作出合理的响应，改变程序正常的执行轨迹。在利用加密保护程序被篡改的机制中，如果攻击者非法修改程序，则程序在运行过程中因无法被成功解密而不能执行原有功能。

3. 低开销

设计软件防篡改方案需要考虑实施防篡改机制的开销，比如通过硬件方式实现的防篡改机制往往需要较高的成本；防篡改机制本身应该与被保护软件合理结合；需要考虑实施防篡改机制后被保护软件的执行性能。

5.3　软件防篡改技术

本节在详细介绍各种软件防篡改技术方法之前，根据不同的分类方式及设计准则，对其进行比较，如表 5-1 所示。

表 5-1　各类防篡改技术比较

防篡改技术	分类方式[①]							设计准则		
	实现模式		完整性检测手段			实现主体		隐秘性	可预测性	低开销
	A1	A2	B1	B2	B3	C1	C2			
校验和		√		√			√		√	√
IVK		√			√		√	√	√	√
哨兵		√	√				√	√	√	√

续表

防篡改技术	分类方式①							设计准则		
	实现模式		完整性检测手段			实现主体		隐秘性	可预测性	低开销
	A1	A2	B1	B2	B3	C1	C2			
断言检查		√		√			√			
隐式哈希		√		√			√		√	√
Tester-Corrector		√	√				√	√	√	√
控制流图检测		√	√				√		√	
基于分支函数的检测		√	√				√			√
联机检测		√	√			√			√	
指针置空响应法②		√					√	√	√	√
加密		√			√		√	√	√	
硬件方式	√					√		√	√	

注：① 分类方式下各代号的含义如下：
A1：硬件方式；A2：软件方式
B1：检查程序代码本身的完整性；B2：检查程序运行中间结果的正确性；B3：对程序进行加密
C1：外部式；C2：自我式
② 按实现防篡改功能的主体划分，该防篡改技术方法属于自我式中的响应手段，而不属于检测程序完整性的手段。

5.3.1 校验和

校验和(checksum)是广泛应用于通信领域中冲突检测的一种技术，现在也被应用于软件的完整性验证。这种技术的优势是能够在被保护程序内部进行完整性验证，而不需要外界运行环境的支持，并且可以在程序编译/链接阶段自动地集成校验和计算比较代码。

在程序发布之前，向其中嵌入校验和计算比较的代码以及原始的校验和。在程序运行前，程序利用报文摘要算法计算其校验和并与其原始的校验和进行比较，如果发现不一致，则判定程序被篡改，从而触发相应的响应模块。

这种方法的缺点是隐秘性比较差，如果检测部分被攻击者探测，则很容易被删除或者绕过。因此，校验和机制通常不会被单独使用，一般作为其他防篡改手段的一个中间步骤。Wurster 等人提出了针对基于校验和的防篡改技术的攻击方式，其主要思想是：这种基于校验和的防篡改机制建立在下面的假设上，即代码和数据共用相同的内存空间。然而，利用代码和数据间的地址转换机制的不同，特制的处理器能够使得被检验的代码与实际执行的代码没有任何关系。当程序运行时，处理器执行的是攻击者修改过的代码；当程序运行到校验和检验这一步骤时，程序读取原始的代码并计算其校验和，从而可以顺利地通过完整性验证而无法探测篡改行为。

5.3.2 多块加密

Aucsmith 在[3]中第一次提出多块加密思想，介绍了一种保护软件代码被分析篡改的

技术。防篡改领域中许多论文都运用了其设计思想。它设计了一种能够进行自我修改、自我解密的结构体，称为完整性验证模块(Integrity Verification Kernel，IVK)。IVK 嵌入被保护程序中，执行两类任务：

(1) 验证程序的完整性，即验证代码块的数字签名与预存值是否一致。

(2) 与其他 IVK 通信共同完成相关功能以加强防篡改机制的力度。

图 5-1 描绘了 IVK 的结构，它由若干个等大小的代码单元(cell)组成。在 IVK 被执行之前，除了第一个单元，其他单元均处于加密状态。每一个单元均包含一个“解密跳转”模块。在整个 IVK 的运行过程中，同一时刻有且只有一个单元处于解密状态。IVK 的入口位于第一个单元。IVK 的执行过程如下：

(1) IVK 在第一个单元接受相关参数后开始运行。

(2) 第一个单元利用输入参数设置 IVK 的初始状态。

(3) IVK 执行第一个单元的“解密跳转”模块，并跳转至解密的单元执行，记作 C_i。

(4) C_i 执行完自身功能后，执行自己的“解密跳转”模块，解密并跳转至另一个单元 C_j。

(5) 重复步骤(4)，直至所有单元被执行。

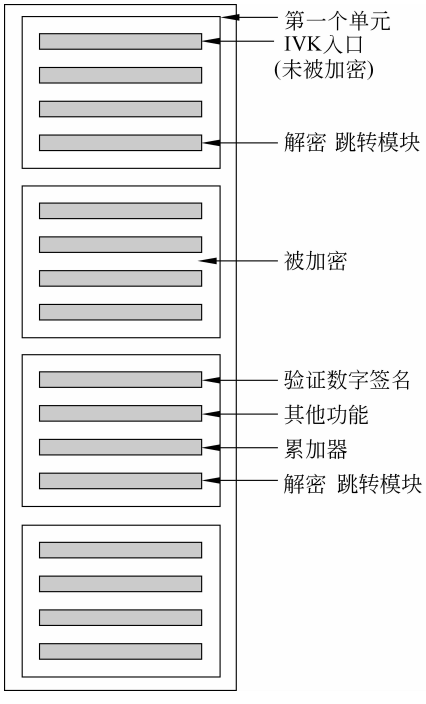

图 5-1　IVK 结构示意图

每一个单元完成的功能如下：

(1) 计算验证待保护程序代码段的数字签名的完整性。

(2) 检查程序是否被非法调试。

(3) 结合哈希函数，通过累加当前已执行过单元的特征值，检查当前单元之前的所有单元是否被正常运行。

为增强防篡改能力，多块加密技术提出了一套多个 IVK 模块联锁信任机制。根据设计需要，每一个 IVK 可以验证其嵌入宿主程序代码段的完整性，也可以验证其他程序代码段的完整性；一个程序当中可以嵌入多个 IVK。这种机制保证遭受篡改的程序必然非正常终止。IVK 机制的局限性在于它不适合于类型安全型程序语言，如 Java。

5.3.3　哨兵

Hoi Chang 和 Mikhail Atallah 等人提出了基于多点设置的防篡改技术。该技术利用称为哨兵(guard)的单元(一个 guard 即一段代码)来保护程序，哨兵的功能主要有两类：

(1) 对代码求校验和。哨兵使程序具有“完全自我感知”的能力。它在程序运行时刻计算其保护代码的校验和以检查其完整性。

(2) 修复代码。哨兵使程序具有“自我修复”的能力。当发现代码的完整性被破坏后，哨兵修复被篡改的代码至初始版本，使程序就像未被篡改一样正常运行。

若干哨兵被嵌入待保护程序中，组成一个哨兵网络，如图 5-2 所示，它们对程序的不同代码段执行上述安全相关的任务，增强保护力度。同时，哨兵互相保护彼此的完整性，攻击者必须绕过或移除所有哨兵才能顺利实施攻击。哨兵机制有如下优势：

（1）分布性。哨兵在程序中的多点分布远远增大了攻击者定位的难度。程序在不同位置处运行这些哨兵以保护程序自身。

（2）多样性。几个哨兵可以共同保护同一段代码。它们在不同时刻执行相关的安全任务。

（3）动态性。配置哨兵网络的方式是可变的。即使攻击者明确程序使用了哨兵机制作为保护手段，但在不了解其部署机制的前提下，仍然无法成功篡改程序以实现自己的目的。

（4）可扩展性。保护程序的级别是可以扩展的。比如，针对规模大或安全级别高的程序，可以适当增加哨兵的数目。

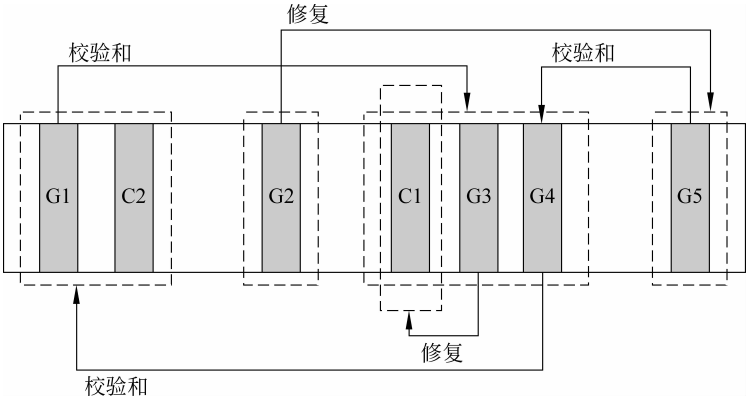

图 5-2　使用哨兵机制保护程序

5.3.4　断言检查

断言检查(assertion checking)检查程序中间执行结果的有效性。断言是一种逻辑表达式，用于说明一个条件或程序变量之间的关系。断言检查即检验断言是否正确。例如，程序的一个变量 i 在某个特定的位置的值应该是正数，但是在这之前被赋予负值，那么断言检查将失败，认为发生篡改行为，导致程序出错。这种方法存在几个缺陷：

（1）一个非预期的变量值可能来自程序错误，而不是由非法篡改造成的。程序应该具备自我恢复能力，例如输出一个错误的提示信息。但是这种防篡改的方法最终会导致程序提前终止，降低了程序的容错能力和可用性。

（2）在程序中嵌入过多的中间结果检查代码会降低程序的执行效率，并且这样很容易被攻击者定位。

（3）这种方法的应用很难实现自动化。程序员需要复查代码，决定插入断言检查语句的位置与数量。

（4）这种机制极有可能出现漏检现象，无法保证程序被篡改后就一定会产生错误的中间结果。

5.3.5　隐式哈希

Yunqun Chen 等人提出了一种程序完整性验证的原型——隐式哈希(oblivious hashing)。该原型在程序运行时计算其执行轨迹的哈希值,并与原始预存哈希值进行比较以验证程序的完整性。

作为一种程序完整性验证的原型,其软件实现方法是将计算哈希指令插入待验证的程序代码中。与传统哈希和校验和相比,隐式哈希依赖于程序运行时而非静态特征计算代码的哈希值;并且其操作具有隐秘性,即哈希指令与程序指令具有相似性,能够与待验证的程序无缝融合,不易被攻击者定位为特殊指令。

Chen 等将隐式哈希原型应用于 Java 程序:在 Java 程序运行时,计算栈顶元素的哈希值,将其与预存值进行比较以判断程序是否被篡改。M. Jacob 等利用代码重叠技术将隐式哈希指令嵌入目标代码,以验证程序运行时状态和指令的完整性。

隐式哈希涉及与原始值的比较,不合理的原始值预存方案将成为攻击者的突破口,比如攻击者替换原始值从而成功通过完整性验证。

5.3.6　Tester-Corrector

Horne 等人提出了一种动态自我式防篡改的解决方案。该机制由两类组件构成:tester 和 corrector。tester 以源代码的形式被嵌入待保护程序中。在其运行过程中,tester 计算代码区段(code interval)的哈希值,与预存的原始哈希值进行比较以判断程序是否被篡改。corrector 巧妙地解决了原始哈希值的存取问题,它是一个字节,可被设置为任意值,存储的载体称作 corrector slot。corrector 被放置在待保护程序的基本块之间。代码段实际上除了自身以外,还包括 corrector slot。tester 使用迭代式线性哈希函数对这两部分进行哈希计算,如图 5-3 所示。corrector slot 存取适当的值使得代码区段被哈希为事先约定的固定值。tester 在程序运行时对代码区段应用哈希函数,若程序完整性被破坏,则无法得到预计的固定值。Bill Horne 等人提出了将 tester 与水印相结合的策略来增强抗攻击性;利用多点设置 tester-corrector 的思想和其他手段来提高该防篡改机制检测代码的覆盖率;增强它的隐蔽性;增大攻击者的攻击成本。然而,Bill Horne 尚未提出健全的响应机制,它也将其作为未来研究工作的一部分。

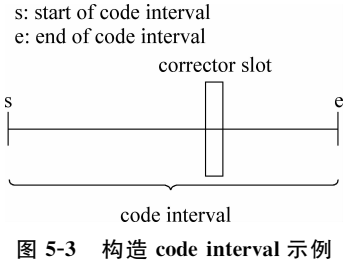

图 5-3　构造 code interval 示例

5.3.7　控制流图检测

Brian 在其硕士毕业论文中描述了一个双进程模型:将被验证程序编译成两个进程:监视进程(monitor process)和主进程(main program process)。前者检查后者控制流的正确性,即实现了完整性验证的功能。该模型认为所有的篡改模式(如数据篡改、内存篡改等)都将导致程序控制流图发生变化。主进程周期性地向监视进程发出完整性验证请求,监视进程则将主进程当前的控制流图与编译期获得的原始结果进行比较,若检测到篡改,则作出响应,如中止主进程。该模型依然未详细给出合理的响应机制;而且,它应用于工程的可行性值得进一步研究。M. Abadi 等人也提出了控制流完整性验证的实现。

5.3.8 基于分支函数的检测

Hongxia Jin 等人设计了一种基于分支函数的防篡改检测机制。分支函数将程序的控制流转移给非条件分支的目标指令，其思想如图 5-4 所示：条件转移指令转换成对分支函数的调用，分支函数返回条件转移指令的目标地址。Hongxia Jin 等人将分支函数扩展为完整性验证分支函数（Integrity Check Branch Function，ICBF）。它执行完整性验证和关键码演化等任务，可以检测程序是否被篡改：当完整性验证的结果或关键码被修改时，分支函数返回错误的目标地址，导致程序异常运行。该机制不需要与原始结果进行比较从而检测程序篡改行为，但它有 3 个缺陷[29]：

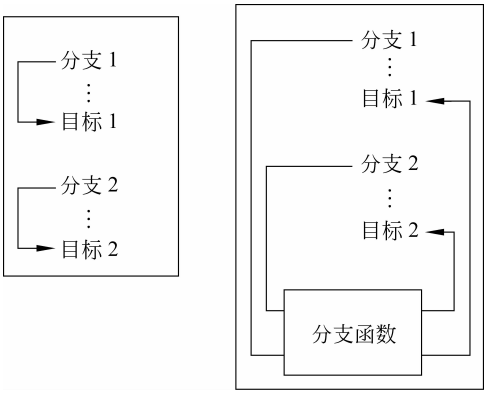

图 5-4 分支函数图示

（1）该机制假设完整性验证模块能够成功检测攻击者的篡改行为；并在攻击者绕过 ICBF 之前导致程序异常终止。

（2）初始关键码的存储可能成为攻击者的一个突破点。

（3）较难确定分支指令集合的选择。关键码演化过程能够代表程序的运行行为，位于指令函数执行时必经路径上的分支是合适的候选对象。

5.3.9 联机检测

Hongxia Jin 等人提出了一套基于事件日志的联机检测机制以验证程序是否被篡改，捕捉攻击者对程序实施逆向工程时留下的证据。假设被验证的软件被嵌入完整性检查模块，位于客户端；验证端（clearing house）位于服务器端。在软件执行过程中，完整性验证模块计算的结果被周期性地记入日志文件中。每一条日志记录都加上时间戳，时间戳用随着时间演化的关键码（key）表示。软件周期性地与验证端进行网络联接，日志文件被传送至验证端以验证程序是否被篡改。关键码的演化过程中用到单向（不可逆）函数，有 3 种记录日志的方法（见图 5-5）：

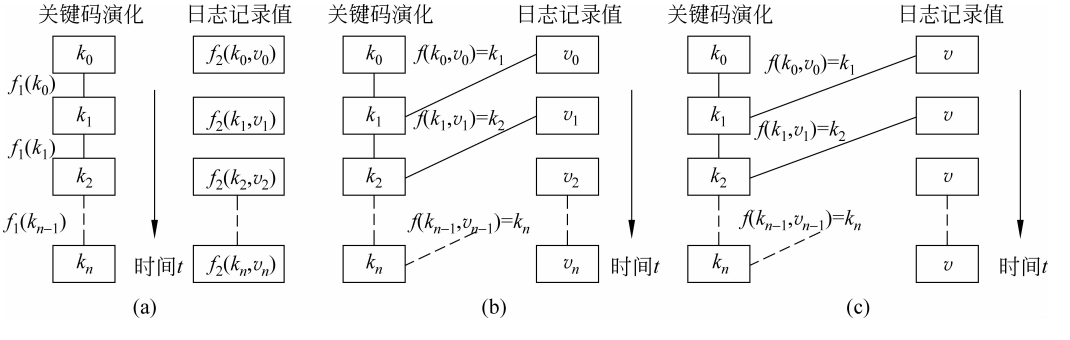

图 5-5 3 种关键码的演化过程与日志记录的关系

(1) 图 5-5(a)中密钥的演化过程不依赖于记入日志中的完整性验证的结果值。在每一轮回的计算中,函数 $f1$ 以当前密钥为参数,生成下一轮的密钥;$f2$ 以密钥和完整性验证的结果作为参数,计算的结果被记入日志文件中并被传送至验证端。验证端再次应用函数 $f2$,将结果与预存值比较,以判断篡改行为是否发生。

(2) 图 5-5(b)中密钥的演化过程依赖于记入日志中的完整性验证的结果值。在每一轮回的计算中,函数 f 以当前密钥和完整性验证的结果为参数,计算下一轮的密钥。每一轮的完整性验证的结果值和最后一轮的密钥被记入日志文件,并传送至验证端。与第(1)种方法相比,本方法的日志文件存储的是明文,即完整性验证的结果值未经处理。

(3) 图 5-5(c)中密钥的演化过程不依赖于记入日志中的完整性验证的结果值。该方法采取合适的完整性验证策略,使得每一次完整性验证产生相同的结果。在每一轮回的计算中,函数 f 以当前密钥和完整性验证结果(常量值)为参数,计算下一轮的密钥。只有最后一轮的密钥和轮数记入日志文件,被传送至验证端。与前两种方法相比,这种方法减小了记录日志文件内容的大小。

Hongxia Jin 等提出的联机机制可以在攻击者成功篡改软件之前检测出其篡改行为。但它有 3 个缺陷:

(1) 该机制要求软件周期性地联接网络,它只适合顾客与服务提供商保持长期联系的商业场景,而不适合那些只在本机运行的软件。

(2) 该机制假设攻击者篡改软件一定被验证端成功检测。在实际情况下,如果攻击者能够完全控制软件(属于第 5.2.1 节提到的第三类攻击),则其可以成功地通过验证端的完整性验证,如伪造校验和值等。

(3) 验证完整性的代码段的位置不易确定。随意选择会导致传送至验证端的日志文件过大;限制验证完整性的代码段位于程序执行必经路径上也无完全可行性:程序运行必经路径不一定足够多;确定程序运行必经路径是 NP 难问题。

前面主要介绍了几种程序自我检测的方案。事实上,检测机制与响应机制两者相辅相成,均是自我式防篡改技术不可或缺的一部分。即使检测机制十分健全,脆弱的响应机制也可能导致整套防篡改机制无效。在目前的防篡改领域中,对响应机制的研究十分有限。下面介绍一种针对 C 语言程序被篡改后的响应方案。

5.3.10　指针置空响应法

Gang Tan 等人提出了一种针对 C 语言程序的篡改响应机制。当程序检测到被篡改后,将程序的指针变量置空;程序解引用被置空的指针将导致程序崩溃。如图 5-6 所示,模块 B 和 C 使用全局指针 p,模块 A 不使用 p。当模块 A 检测到程序被篡改后,将指针 p 置空。接着,程序保持正常运行,直至执行模块 B、C 解引用指针 p 时,程序崩溃。该种响应机制具有如下优点:

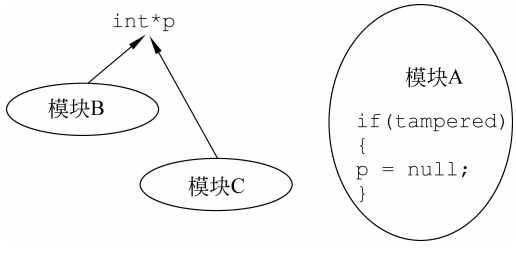

图 5-6　指针置空响应法

(1) 从空间与时间上将响应与程序故障分离。在空间上,模块 A 响应,模块 B、C 导致程序崩溃;在时间上,模块 A 作出响应后,要

过一段时间，直至模块 B、C 被执行时，程序才崩溃。

（2）响应代码具有隐秘性。将指针置空的程序语句是简单的赋值语句，不易被定位。

（3）程序崩溃的可预测性。程序经检测被篡改后，将指针置空，程序一定出现异常。

同时，该响应机制也有如下的局限性：

（1）未考虑不使用指针及全局变量的程序语言，如 Java。

（2）忽略了一条通用的编程规则：C 语言程序在解引用指针之前会做安全检查，即只解引用非空指针。

5.3.11 加密

近年来，许多研究者提出将加密技术应用于软件保护领域。加密技术可以阻碍攻击者探查程序代码。

Cappaert 等人提出了一种基于密码技术的软件实现方案，阻止恶意的软件宿主篡改程序。该解决方案中，对程序中的部分代码进行加密，密钥由程序中另一部分代码应用校验和或哈希函数计算获得。在程序运行过程中，只有需要运行的代码才被解密，当其执行完毕后，又再次被加密，称该机制为按需解密（on-demand decryption）。整个方案以函数为操作粒度，利用函数调用图（call graph）建立代码之间的依赖关系，调用函数方（caller）生成密钥，对被调用函数方（callee）进行加密和解密。生成密钥，加解密的任务由加密哨兵（crypto guard）完成，它可以以内联函数的形式嵌入被保护程序中。Jaewon 等人也描述了类似的机制。该方案的优势如下：

（1）与其他通过条件判断语句实现的防篡改方案相比，Cappaert 等人提出的方案不需要存取原始哈希值，也不需要运用条件判断指令，更不容易被攻击者定位进而移除。

（2）若被解密的代码或生成密钥的代码被篡改，则解密无法成功，程序异常退出。此特性加大了攻击者跟踪至程序退出位置的难度。

“哨兵”思想可以解释为，将若干个加密哨兵组织成一个网络散布在被保护程序当中，增大攻击者分析和修改程序代码的难度。图 5-7 展现了该解决方案的基本思想。

Michiels 和 Gorissen 针对白盒攻击，利用白盒加密技术使程序软件能够抵抗非法篡改。白盒攻击（white-box attack）和白盒加密（white-box cryptography）的概念如下：白盒攻击者可以完全访问软件及控制其执行环境，包括跟踪执行轨迹，查看中间执行结果，捕捉内存中的密钥信息，执行静态代码分析等。白盒加密方案即为阻碍白盒攻击的手段。在该解决方案中，程序的密钥无法被攻击者定位提取（程序中部分代码是被加密的，需要经解密才能正常运行）。该解决方案的主要优点如下：

（1）它加大了篡改程序的难度。篡改用于生成密钥的代码或被加密的代码将导致解密生成不正确的指令，进而引起程序崩溃或不确定的运行行为。

（2）它加大了静态分析程序的难度。由于密钥是在程序运行时生成的，攻击者无法通过静态分析手段获得密钥。

该解决方案有一个缺陷：在程序运行时计算密钥以及加解密将产生不可忽视的开销，进而影响程序性能。借助一些启发式方法，合理选择生成密钥的函数体，在程序防篡改力度与其运行性能之间作出平衡折中。