基于压缩的代码保护的低开销策略

陈 勇 何炎祥 石 谦 吴 伟 李清安
(武汉大学计算机学院 武汉 430072) (武汉大学软件工程国家重点实验室 武汉 430072)

摘 要 利用压缩算法及C语言编译器辅助分析,提出了一种用于代码保护的低开销策略。设计了一种基于C语言安全漏洞的安全级别模型,它对不同安全级别的代码采用不同的保护策略,以减少保护开销。同时设计了一种分块二进制压缩算法(BCC压缩算法),对保护前的代码进行了压缩处理,使保护开销进一步降低。软件模拟实验表明,使用该压缩算法可将保护代价降为原代价的80%~90%。

关键词 代码安全保护,安全模型,压缩算法,低存储容量
中图法分类号 TP309.1 **文献标识码** A

Low-cost Protection Strategy Based on the Code Compression

CHEN Yong HE Yan-xiang SHI Qiang WU Wei LI Qing-an
(School of Computer, Wuhan University, Wuhan 430072, China)
(State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China)

Abstract Taking advantage of compression algorithm and C language compiler supporting analysis, we proposed a low-cost protection strategy. To reduce the cost of protection, we designed a security model based on the security problem of C language itself to protect the object code using different security level. Moreover, we invented a new binary compression code algorithm(BCC) based on the blocks to compress the target code before using the protected strategies to further step down protects spending. Through the experiment, we got about 80%~90% cost of the strategy which do not use the compression algorithm.

Keywords Code security, Security module, Compress algorithm, Low storage capacity

1 引言

近年来随着软件规模的不断扩大,软件的应用范围越来越广泛,但由软件病毒、木马带来的损失也越来越大,人们急需获得安全、高效的软件。C语言是目前使用最广泛的编程语言,它的健壮性直接影响其生成的软件的安全性。但为了满足语言灵活和高效的特点,C语言自身存在不少安全缺陷。如文献[1]中所述,它的安全缺陷主要有以下几个方面:类型及转换、宏定义、缺少数组越界检查、缓冲区溢出、指针问题、内存泄露等。因此,本文从可信编译器的设计出发,针对C语言的安全缺陷,设计了一套基于代码压缩的安全保护策略。本策略采用软硬件结合的方式,一方面提高软件的安全性,另一方面最大限度减少标记性代码的长度,以降低安全保护所付出的开销。

本文第2节介绍现有的C语言保护手段;第3节介绍本文采用的安全模型;第4节将详细介绍具体的保护方式以及相应的压缩和加解密算法;第5节是软件模拟的实验结果;最后是相关工作和总结。

2 C语言代码常用保护方法

针对以上C语言的安全缺陷问题,目前主要有3种解决方案:软件审核、缓解易攻击性以及行为管理[2]。

软件审核是在攻击者有机会攻击软件弱点前就防治这些弱点。该方法既可以用于静态分析,也可以用于动态调试。静态分析主要通过扫描源程序,对程序中潜在的易受攻击的部分发出警告。如工具BOON[9]和CQual[8]可以查找C语言中潜在的缓冲区溢出或者发现值的使用不合理;MOPS[14,15]利用有限状态机模型检查由于程序凌乱而导致的格式行漏洞;code obfuscation methods[11,12]和proof carrying code[10]可以用于检查注入代码攻击。但是静态分析可能导致误报,而且有些工具比较耗时。

缓解易攻击性是通过在编译时对易受攻击的地方加入保护机制的代码,在运行时检测这些易受攻击的地方。该类方法在于最大程度降低攻击的伤害。如StackGuard[3]用于检查缓冲区溢出的问题;FormatGuard[13]用于检查printf函数格式匹配的问题。

而行为管理主要是对操作系统特征的描述,如访问控制

到稿日期:2010-12-18 返修日期:2011-03-12 本文受国家自然科学基金可信软件重大研究计划(90818018)资助。
陈 勇(1986-),男,博士生,主要研究方向为可信软件等,E-mail:cyong1000@163.com;何炎祥(1952-),男,教授,博士生导师,主要研究方向为可信软件等;石 谦(1985-),男,硕士生,主要研究方向为可信软件等;吴 伟(1985-),男,博士生,主要研究方向为可信软件等;李清安(1986-),男,博士生,主要研究方向为可信软件等。

策略等。

现有的保护策略虽然在一定程度上加强了软件的安全性，但因其侧重点不同，各种方案均存在一定的缺陷。如静态分析方法虽然速度比较快，但可能导致误报，而且无法防止软件运行时的攻击；动态调试法虽然能够获得比较高的安全性，但其开销比较大。因此，本文结合静态分析和动态调试两种方法，利用编译器前端的分析对目标代码设置安全标记，然后利用 FPGA 硬件根据安全标记采用不同的保护策略。为降低保护代价和代码长度，本文设计了一种简单的二进制压缩方法。编译器生成的目标代码先经过该算法进行压缩后，再对其进行加密和校验处理。当程序运行时，依靠 FPGA 硬件先对要运行的程序进行解密、校验等处理，确认无误后再进行解压缩运行，基本框架如图 1 所示。

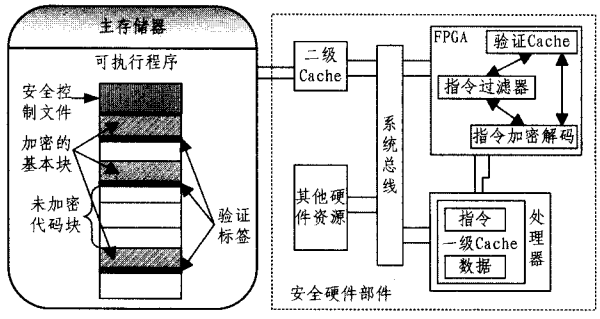

图 1　代码保护策略总体结构

3　代码安全级别模型

根据国家信息安全漏洞共享平台(CNVD)[16]的统计，从 2001 年 1 月到 2010 年 7 月，因输入验证错误产生的漏洞所占比率最多，为 29.82%；其次是设计错误(一般的漏洞都存在设计方面的问题)，为 21.56%；接着是边界条件错误数目，为 19.91%。其中输入验证错误是由于未对用户提供的输入数据做合法性检查，从而引起缓冲区溢出、指针非法等错误。缓冲区错误占整个漏洞比例的 2.4%。由此可见，缓冲区溢出、指针问题是最主要的问题，其次便是数组越界造成的边界条件错误漏洞。因此，本文根据该统计信息，并结合第 2 节有关 C 语言自身缺陷的分析，将 C 语言代码分为 4 个安全级别，如表 1 所列。

表 1　代码安全级别模型

C 语言代码	安全级别
缓冲区溢出、指针问题	3
数组越界	2
条件判断语句、类型转换、内存泄露、void 类型	1
其他	0

4　代码保护策略

4.1　总体设计

本文采用的代码保护策略主要分为 3 个阶段：

第一阶段，在编译器对源程序划分基本块后，本文根据第 2 节的代码安全级别模型，标记各条语句的安全级别，然后根据公式 $S_{bi}=\mathrm{Max}(Sc_{bi,k})$ 计算各个基本块的安全级别(其中 b_i 表示基本块 i，$c_{bi,j}$ 表示基本块 b_i 的第 j 条语句，S_k 表示 k 的安全级别，k 为基本块和基本块中的语句)，最后将标记写入相应的配置文件中。

第二阶段，在编译器的目标代码生成时，先用压缩算法对目标代码进行压缩，然后根据压缩后的实际结果修改相应的安全控制文件，最后按表 2 对目标代码进行加密和验证标记插入的处理，同时对安全控制文件本身进行加密和验证处理。

表 2　安全级别代码及其处理方式

代码级别	处理方式
0	不处理
1	插入验证标记
2	加密处理
3	加密＋验证处理

最后在目标代码执行时，以 FPGA 为依托，对目标代码进行解压缩和解密验证处理后再运行。由于该部分使用硬件实现，因此在解密和解压缩的速度上相对于软件要快得多，而且由于该部分直接与处理器通信，避免了非法入侵者探听总线信号以获得相应指令的危险。图 1 描述了该阶段的总体架构。首先，在程序运行前，先通过解密部件将安全控制文件的数据经过解密和验证处理，无误后加载到 FPGA 的 CBB 模块中，以控制程序的执行，如图 2 所示。每当加载一个块到缓存运行时，先核对该块的安全标记，根据安全标记信息对数据进行解密和验证处理，无误后再将数据传给解压部件解压缩，最后传给处理器进行处理。如果在其中任何一个环节出错，则退出程序，停止执行。

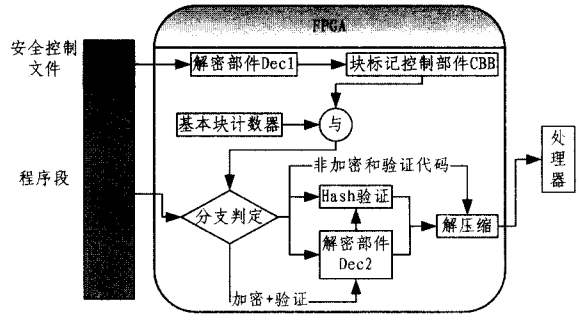

图 2　FPGA 内部模块详细设计图

4.2　压缩算法

为了节省存储空间，适应低存储容量的设备，同时进一步减少加、解密和验证的数据量，本文对目标代码进行了压缩处理。

目前的压缩算法主要分为两类：无损压缩和有损压缩。无损压缩是指将压缩后的信息重构后(即解压缩)能够完全还原为原来信息。而有损压缩是指重构压缩后的数据，其重构数据与原始数据有所不同，但不影响原始数据表达信息。对于程序这样的数据，在编码后必须能够重新还原，故只能使用无损压缩算法。目前的无损压缩算法主要有基于哈夫曼编码的、基于算术编码的、LZW 算法、LZSS 算法等。这些算法的压缩方式一般是根据字符或字符串出现的频率，以字典或映射表的形式设置相应的等长或不等长编码，最终达到对大量数据进行压缩的目的。但是无论是字典还是映射表，它都给程序添加固定开销(即字典和映射表本身的存储空间)，这些开销相对于较大的程序而言可以忽略不计，但对于较小的程序，特别是程序中指定块，其开销比例将大幅度增加。因此，本文设计了一种简单的压缩算法(BCC 算法)，算法采用了直接对二进制数据进行编码的方式，一方面可以省去映射表和字典的开销，适应较小程序的压缩；另一方面算法实现简单，可以用软硬件平台实现，便于降低系统安全加强模块的复杂

度。

本压缩编码由 3 部分组成：

3bit (m 值)	1bit (表示起始 bit 值)	m bit	m bit	……	m bit

第一部分是编码长度位，占用 3 个 bit 位空间。它确定的值 m 表示用 m 位对各个连续出现的 0 或 1 进行编码。

第二部分是起始 bit 值标志位，占用 1 个 bit 位空间。它表示起始 bit 是从 0 开始还是从 1 开始。

第三部分是具体对数据进行编码的部分。由于连续的 0 和 1 是间隔出现的(即连续的 0 后面必然是连续的 1，而连续的 1 后面必然是连续的 0)，因此只要确定了起始 bit 位的值(0 或 1)，就可以依次对后续部分编码。该编码表示连续相同 bit 位的个数，用 m 位进行等长编码。

本压缩算法主要分为以下几个步骤：

1)扫描整个目标文件的二进制代码，记录连续相同二进制位相应的长度的个数，用 N_i 表示连续 i 个相同二进制位出现的个数。如对二进制代码串“0001110011001110000”，$N_1=0, N_2=3, N_3=3, N_4=1$。

2)根据上一步计算的 N_i 值，计算压缩后代码的长度。对于一串目标代码，如果采用固定长度$(m+1)$位对连续的 0，1 串进行编码，对于没有溢出的情况(即没有超过 2^{m+1}个连续数据)，其长度为$(m+1)*\sum_{i=1}N_i$ 个 bit。但通常情况下会出现溢出的情况，因为 m 仅由 3 个 bit 确定，所以$(m+1)$最大值为 8，因此最多只可以表示 255 个连续的数据。而目标程序由于字节对齐等原因经常需用 0 填充部分代码空间，从而导致目标代码中会出现超长连续的 0，所以必须计算溢出情况。当连续数据长度 n 超过 2^{m+1}(即 255)时，本文将其分成 $1+\left\lfloor\frac{n}{2^{m+1}+1}\right\rfloor*2$ 组数据，相对于原来的编码就增加了$\left\lfloor\frac{n}{2^{m+1}-1}\right\rfloor*2*(m+1)$位的数据。例如采用 m 值为 7 对 256 个连续 1 进行压缩编码，则为 111111111 00000000 00000001。因此，当采用 m 位对数据进行编码时，编码后的总长度为：

$$n+1+(m+1)*\sum_{i=1}N_i+\sum_{i=2^{m+1}}\left(\left\lfloor\frac{i}{2^{m+1}-1}\right\rfloor*2*(m+1)*N_i\right) \quad (1)$$

式中，n 为表示用来标记使用多少位对目标代码进行编码(在实验中使用的值为 3)；m 为表示固定编码长度－1，且满足条件 $m<2^n$；N_i 表示连续 i 个相同二进制位出现的个数。

3)使用式(1)，遍历 2^n 次，计算出各个 m 值时编码的长度。选择其中编码最短的 m 值作为最后编码长度，接着对原数据进行压缩。

具体算法如图 3 所示。

```
Input:the source object code
output:the compressed code
algorithm:
1. minvalue=+∞
2. minindex=0
3. traverse the source code to calculate the Ni(i=1,2,3,…)
4. foreach m in {0,1,2,…,2^n-1}
5.     tmpvalue=n+1+(m+1)*∑Ni+∑(⌊i/2^(m+1)⌋*2*(m+1)*Ni)
6.     if tmpvalue<minvalue
7.     then
8.         minvalue=tmpvalue
9.         mininde x=m+1
10.    endif
11. endforeach
12. use m+1 bit to compress the source code
```

图 3　BCC 压缩算法

本算法主要是对目标二进制文件进行分析，统计连续出现的 0 和 1 的个数，然后计算不同编码情况下使用的代码开销，最后选择开销最小的一种编码作为最终压缩方式。由于本压缩方法使用的源码只有 0 和 1，因此可以大大减小固定开销。文件开头部分先采用 n 个 bit 确定编码的长度，后面紧跟着用 1 个 bit 确定起始编码的源码，接着就是正式编码，长度为前 n 个 bit 确定的值。因此采用这种编码方式，固定开销只有 $n+1$ 个 bit 位。

4.3　加密与验证算法

为了加强代码的安全性，同时又不至于给系统增加过大开销，本文根据编译器的分析结果，对部分代码进行加密和验证处理。加密算法可以分为对称加密算法(如 DES，AES 等)和非对称加密算法(如 RSA)。目前较常用的加密算法是 DES 和 AES，也有将这两种加密算法整合起来的。但总的来说，AES 的加密强度高于 DES，因此本文主要采用了文献[6，7]中的 AES 对称加密算法对数据进行加密处理，一方面可以更好地保障代码的安全性，另一方面便于硬件实现。在验证中，先采用奇偶校验位对需要验证的块进行验证，最后对整个代码进行验证。一方面采用分块以及双重验证，可以获得较大的安全保障；另一方面奇偶校验，可以最大程度地减少校验位的开销。

5　软件模拟实验及分析

因为保护的开销与被保护的代码量有关，保护的代码量越小，其保护的开销也就越小，所以本文主要采用被保护的代码量作为评估标准，进行了软件模拟实验。本文选择了 5 个用 gcc 4.4.2 版本编译的 C 语言目标代码以及 3 个 Windows 平台的二进制文件作为测试用例，使用自己编写的 BCC 压缩程序以及修改的 LZW 压缩程序进行测试。运行环境为：Pentium(R) Dual-Core CPU E5300 的处理器，2GB 内存，Windows 操作系统。

实验过程主要包括以下几个步骤：

1)使用安全控制文件 Config1 随机添加安全标记信息，用以模拟编译器分析结果。Config1 的结构如图 4 所示(每个数据块的大小为 64B)。

1. 起始数据块	结束数据块	标记数据
2. 起始数据块	结束数据块	标记数据
……		

图 4　Config1 结构

2)以块为单位，使用相应的压缩算法对目标数据进行分块压缩，并将压缩后的数据信息写到安全控制文件 config2 中，供加密和验证处理。Config2 的结构如图 5 所示。

1. 起始字节	结束字节	标记数据
2. 起始字节	结束字节	标记数据
……		

图 5　Config2 结构

表3中记录整个测试结果。由该结果可以看出，经过压缩，总体的数据量将定量地减少。特别是当原始文件较小时，压缩率可以达到10%～20%。但对于较大的程序，如测试用例7，其压缩程度不到1%，但是其字节减少量也达到了1000字节左右，对于存储空间极其有限的嵌入式环境而言也是比较可观的。

表3 压缩处理前后代码长度对照表

目标文件名	原始文件大小(B)	经压缩处理后文件大小(B)	压缩比
迷宫问题	7919	5891	74.39%
骑士遍历回逆	5991	5076	84.73%
推箱子算法	5445	4527	83.14%
常用图算法	12697	11565	91.08%
简化的汇编翻译器	10663	9719	91.15%
IPMSG	159744	143124	89.60%
InsPro Disk 2.0	241920	240985	99.02%
WinGraphviz_v1.02.24.msi	437760	423044	96.64%

图6、图7显示了本文所使用的BCC压缩算法与常用压缩算法LZW的比较结果。可以看出，BCC压缩算法在基于块的小数据量的压缩上略好于LZW。尤其是当目标文件比较大时，由于分块数的增加，LZW算法在各个分块的固定开销所占比例大幅度增加，从而导致了它压缩后的数据量反而远超过原始数据量。而BCC算法能够保持在与原始数据量相当的位置，甚至还有一定数据量的压缩，如表4最后两个测试所示。

表4 BCC和LZW压缩算法比较

目标文件名	原始文件大小(B)	BCC压缩后(B)	LZW压缩后(B)
迷宫问题	7919	5891	5864
骑士遍历回逆	5991	5076	5052
推箱子算法	5445	4527	4511
常用图算法	12697	11565	11709
简化的汇编翻译器	10663	9719	9466
IPMSG	159744	143124	146163
InsPro Disk 2.0	241920	240985	280463
WinGraphviz_v1.02.24.msi	437760	423044	485002

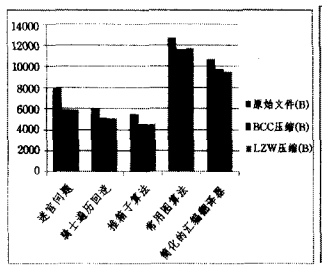

图6 BCC压缩与LZW压缩对比结果1

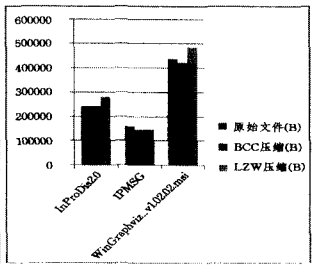

图7 BCC压缩与LZW压缩对比结果2

结束语 如何保证一个程序没有被修改并且不被其他程序入侵，是当前代码安全保障的一个重要课题。Vijay Kongubangaram等人在文献[4,5]中设计了一个与本文类似的编译器框架模型CODESSEAL框架，该工具将程序和数据都以加密的方式存储，在编译时将静态完整性和控制流信息嵌入到可执行代码中。它分为静态验证和动态验证两个部分。静态验证主要是检查程序执行的整体完整性以及其中的签名(SPEE)；动态验证就利用编译时嵌入的静态完整性和控制流信息，防止运行时攻击。该工具验证的手段主要从两个方面进行：1)检查代码和数据块没有被修改过(利用的是静态完整性信息)；2)检查控制流图没有被修改。随后，他们为了减少开销，也将代码分为了4个等级进行加密和验证处理。

Vijay Kongubangaram等人的模型的安全标记是根据函数以及运算的安全性进行添加的。而本文的安全标记是根据CNVD的漏洞统计结果以及在C语言自身缺陷的基础上进行添加的，具有一定的可靠性。同时本文设计了一种简单的压缩方法来对目标代码进行压缩编码，这不但减少了目标代码的长度，减少了保护开销，而且适应了低内存容量的设备。同时，由于代码是压缩存储的，因此即使是未保护的代码也不至于被反汇编程序直接破译，进一步加强了代码的安全性。

参考文献

[1] 姜青山. C语言程序设计的健壮性与安全性研究[J]. 工矿自动化，2007，5：125-126

[2] Gelbart O，Narahari B，Simha R. SPEE：A Secure Program Execution Environment Tool Using Static and Dynamic Code Verication[J]. Journal of High Speed Networks，2006，15(1)：21-32

[3] Cowan C，Pu C，Maier D，et al. Automatic adaptive detection and prevention of buffer-overflow attacks[C]// Proceedings of the Seventh USENIX Security Conference. San Antonio，TX，Jan. 1998

[4] Gelbart O，Ott P，Narahari B，et al. CODESSEAL：Compiler/FPGA Approach to Secure Applications[J]. Lecture Notes in Computer Science，2005，3495：530-535

[5] Kongubangaram V，Gelbart O，Simha R，et al. Compiler-directed Region-based Security for Low-Overhead Software Protection [C]// Third IEEE International Symposium on Dependable，Autonomic and Secure Computing(DASC). 2007：47-54

[6] Wu X，Zhou Y，Gu J C. FPGA design and celerity implementation of AES algorithm[J]. Systems Engineering and Electronics，2007，10(29)：1773-1776

[7] Vishnu M B，Tiong S K，Zaini M，et al. Enhancement of digital motion image transmission using hybrid AES-DES algorithm[C]// 2008 14th Asia-Pacific Conference on Communications(APCC). 2008

[8] Fraser T，Petroni J，Nick L，et al. Flow-sensitive CQUAL to verify MINIX authorization check placement[C]// Proceedings of the 2006 Programming Languages and Analysis for Security Workshop. 2006：3-6

[9] David W，JeffreyS F，EricA B，et al. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities[C]// NDSS. 2000

[10] http://raw.cs.berkeley.edu/pcc.html

[11] Aycock J，Manuel J，Cárdenas G，et al. Code obfuscation using pseudo-random number generators[C]// CSE 2009. 2009：418-423

[12] Preda M D，Giacobazzi R. Semantic-based code obfuscation by abstract interpretation[J]. Lecture Notes in Computer Science，2005，3580：1325-1336

[13] Cowan C，et al. FormatGuard：Automatic Protection from printf Format. String Vulnerabilities [C] // 10th USENIX Security Symposium. 2001

[14] Chen H，Wagner D. MOPS：An infrastructure for examining security properties of software[C]// Computer and Communications Security(CCS). 2002：235-244

[15] Chen H，Dean D，Wagner D. Model checking one million lines of C code[C]// Proc. of the 11th Annual Network and Distributed System Security Symp. 2004：171-185

[16] http://www.cnvd.org.cn/chart/list