基于指令虚拟化的安卓本地代码加固方法

张晓寒 张 源 池信坚 杨 珉*

(复旦大学计算机学院 上海 201203)

摘 要：安卓系统越来越广泛地被应用于各种类型的智能设备，比如智能手机、智能手表、智能电视、智能汽车。与此同时，针对这些平台应用软件的逆向攻击也日益增多，这不仅极大地侵犯了软件开发者的合法权益，也给终端用户带来了潜在的安全风险。如何保护运行在各种类型设备上的安卓应用软件不被逆向攻击成为一个重要的研究问题。然而，现有的安卓软件保护方法比如命名混淆、动态加载、代码隐藏等虽然可在一定程度上增加安卓软件的逆向难度，但是原理相对简单容易被绕过。一种更为有效的方法是基于指令虚拟化的加固方法，但已有的指令虚拟化方法只针对特定架构(x86架构)，无法兼容运行于多种架构的安卓设备。该文针对安卓应用软件中的本地代码提出了一种架构无关的指令虚拟化技术，设计并实现了基于虚拟机打包保护(VMPP)的加固系统。该系统包含一套基于寄存器架构的定长虚拟指令集、支持该虚拟指令集的解释器以及可以与现有开发环境集成的工具链。在大量C/C++代码以及真实安卓软件上的测试表明，VMPP在引入较低的运行时开销下，能够显著提升安卓本地代码的防逆向能力，并且可被用于保护不同架构上的安卓本地代码。

关键词：安卓安全；软件保护；代码加固；指令虚拟化

中图分类号：TN915.08; TP309.2 **文献标识码**：A **文章编号**：1009-5896(2020)09-2108-09

DOI: 10.11999/JEIT191036

Protecting Android Native Code Based on Instruction Virtualization

ZHANG Xiaohan ZHANG Yuan CHI Xinjian YANG Min

(*School of Computer Science*, *Fudan University*, *Shanghai* 201203, *China*)

Abstract: Android system is now increasingly used in different kinds of smart devices, such as smart phones, smart watches, smart TVs and smart cars. Unfortunately, reverse attacks against Android applications are also emerging, which not only violates the intellectual right of application developers, but also brings security risks to end users. Existing Android application protection methods such as naming obfuscation, dynamic loading, and code hiding can protect Java code and native (C/C++) code, but are relatively simple and easy to be bypassed. A more promising method is to use instruction virtualization, but previous binary-based methods target specific architecture (x86), and cannot be applied to protect Android devices with different architectures. An architecture-independent instruction virtualization method is proposed, a prototype named Virtual Machine Packing Protection (VMPP) to protect Android native code is designed and implemented. VMPP includes a register-based fix-length instruction set, an interpreter to execute virtualized instructions, and a set of tool-chains for developers to use to protect their code. VMPP is tested on a large number of C/C++ code and real-world Android applications. The results show that VMPP can effectively protect the security of Android native code for different architectures with low overhead.

Key words: Android security; Software protection; Android packer; Instruction virtualization

收稿日期：2019-12-24；改回日期：2020-06-29；网络出版：2020-07-17

*通信作者： 杨珉 m_yang@fudan.edu.cn

基金项目：国家自然科学基金(U1636204, U1836210, U1836213, U1736208, 61972099, 61602123, 61602121)，上海市自然科学基金(19ZR1404800)，国家“九七三”重点基础研究发展计划(2015CB358800)

Foundation Items: The National Natural Science Foundation of China (U1636204, U1836210, U1836213, U1736208, 61972099, 61602123, 61602121), The Natural Science Foundation of Shanghai (19ZR1404800), The National Basic Research Program of China (973 Program) (2015CB358800)

1　引言

安卓系统是当前最为流行的移动操作系统，被广泛应用于智能手机、手表、电视、汽车等新兴智能设备上。然而，针对安卓应用的逆向攻击层出不穷。根据360安全公司发表的《Android手机应用盗版情况报告》[1]，安卓软件盗版情况十分猖獗，平均每一款正版应用软件对应92.7个盗版软件。这些盗版应用不仅严重侵犯了软件开发者的合法权益，破坏了移动互联网的生态，往往还携带恶意代码，给终端用户造成了极大的安全威胁。因此，如何防止安卓软件被逆向攻击已经成为目前学术领域和工业领域重点研究的问题。

国内外学者针对这一问题进行了广泛的研究。目前较为常用的软件加固技术有代码隐藏[2–5]、代码混淆[6,7]、指令虚拟化[8,9]等方法。其中Kim等人[3]提出使用多个加密Dex的方法保护安卓应用的Java代码；Falsinat等人[4]提出一种使用动态加载代码的方式来防止静态分析。张震等人[5]提出将So文件的特定段加密来保护安卓应用中的本地代码(C/C++)。但是这类加固方法在运行时仍然会将所有代码解密出来，因此攻击者可以使用内存文件转储技术[10,11]在应用代码被解密之后将其导出，比如Yang等人[10]即根据该思路在内存中获取解密字节并将其重组为Dex文件，从而实现了自动化脱壳。赵奇[6]与张一峰等人[7]均提出了用代码混淆方法来加固安卓本地代码，其做法是通过处理本地文件生成的LLVM IR，对其进行指令替换、控制流平坦化、不透明谓词等操作，这些混淆方法提高了逆向攻击的门槛，但是仍未能有效阻止逆向攻击。

一种更为有效的方法是基于指令虚拟化的加固方法，这类方法对将被保护指令转换为虚拟指令，在运行时由一个解释器解释执行。指令虚拟化对逆向攻击者隐藏了真实的指令，大大提高了攻击门槛。胡恒伟[8]提出一种针对安卓Java代码的指令虚拟化方法，利用动态生成的虚拟指令替换Dex文件中原本的指令；李振[9]基于LLVM IR将Dex字节码抽取并转为C代码用于加固Java代码，但这些方法并不能用于本地代码的加固。张汉宁[12]、汤战勇等人[13]以及杜春来等人[14]各自提出了针对二进制文件的指令虚拟化加固方法，但是这些方法只针对x86指令进行虚拟化，无法兼容其他架构的安卓设备。

本文针对安卓软件中的本地代码，基于LLVM IR提出了一套通用的、可以支持多个架构的指令虚拟化技术，设计并实现了虚拟机打包保护 (Virtual Machine Packing Protection, VMPP)加固系统。VMPP包含一套基于寄存器架构的定长虚拟指令集、支持该虚拟指令集的解释器、以及支持开发者使用的工具链，支持将被保护代码适配到不同架构上的安卓系统。本文使用大量的C/C++代码和真实的安卓应用进行了测试，实验表明VMPP对安卓本地代码的防逆向能力有明显的提高，可以有效保障安卓软件代码的安全性，总体性能开销较低并且具有通用性。

2　VMPP加固系统

VMPP加固系统工作原理如图1所示。首先使用LLVM编译器前端将被保护的C/C++代码转为LLVM IR，然后指令转换器根据自定义虚拟指令集，将LLVM IR转换为虚拟指令。之后这些虚拟指令和一个指令解释器，由开发者工具链打包编译生成可执行文件。开发者工具链会根据目标架构生成不同的可执行文件使其可以运行在不同架构上。

2.1　虚拟指令集

虚拟指令集的设计存在两个方面的难点：(1)如何支持安卓本地代码特性和复杂语法结构，比如外

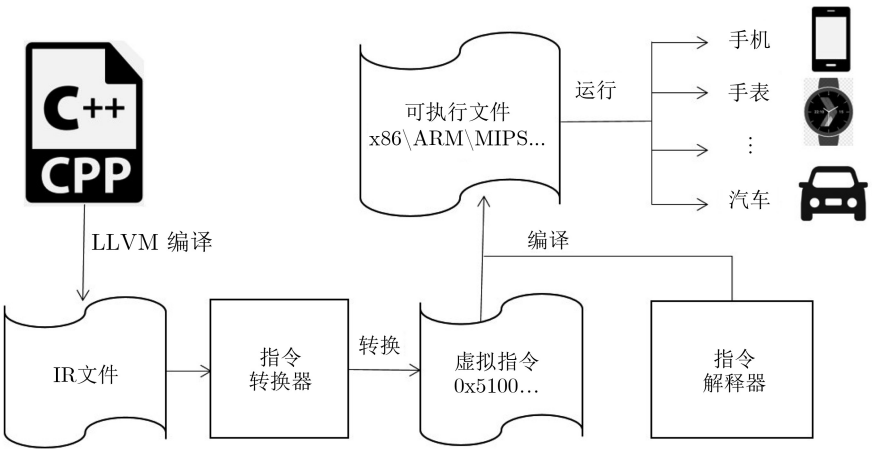

图 1 VMPP原理

部函数调用(包括对libc等库函数调用和JNI函数调用)和异常处理等机制。由于安卓系统服务以及应用代码大部分通过Java实现，安卓本地C++代码需要通过JNI和Java代码进行交互；(2)如何保护虚拟指令集不被攻击者逆向。如果攻击者破解了虚拟指令集，则最终可以逆向被保护代码。

VMPP精心设计了虚拟指令集以解决上述问题。VMPP虚拟指令集参考MIPS(Microprocessor without Interlocked Pipeline Stages)精简指令集设计，为8字节定长指令集。指令集基于寄存器架构，使用256个通用寄存器和16个参数寄存器进行运算等操作。虚拟指令集共有45条指令，分为8个类型：

(1) R类型指令为源操作数均为寄存器的指令，用于算术运算、逻辑、位移等操作；

(2) I类型指令为源操作数为寄存器和立即数的指令，用于算术运算、逻辑、位移等操作；

(3) B类型指令为跳转指令，用于实现指令跳转和内部函数调用；

(4) CMP类型指令为比较型指令，用于比较源操作数；

(5) W类型指令为访存指令和赋值寄存器指令，用于访问内存等操作；

(6) C类型指令为外部函数调用指令，用于处理外部函数调用；

(7) M类型指令包含malloc和return指令，用于分配内存和函数返回；

(8) E类型指令为异常处理的指令，用于处理异常机制。

VMPP的8种指令格式示例如表1所示。所有类型指令的第1个字节表示操作码(opcode)，即具体的指令。表1中“1*”列为“典型指令”列的操作码对应值，比如R类型中add指令的操作码是0x51。后7个字节根据指令类型有不同的格式。比如R类型的指令含义为Dst=Src1 Opcode Src2，其中第2个字节、第7个字节、第8个字节分别表示目的寄存器(Dst)、源寄存器1(Src1)、源寄存器2(Src2)，第3个字节指定了寄存器中操作数的大小，其他未使用到的字节为0。因此表1中数据第1行的指令含义是r1=r2+r3。R类型指令与I类型指令都包含算术类、逻辑类、位移类3种指令，但R类型的操作对象均为寄存器，而I类型指令的操作对象为寄存器和立即数。比如表中I类型中addi指令的操作码为0x30，第5～8个字节是立即数(immediate)，该指令意为r1=r2+immediate。其他6类指令中，CMP,W,M类型分别用于比较、访存、内存和函数返回。B类型指令用于内部函数调用，VMPP定义的调用惯例为，使用16个参数寄存器专门用于传递参数，将参数从左到右依次存入1～16号参数寄存器中，之后调用目标函数，并在其执行完后将返回值放入1号参数寄存器，完成函数调用。

VMPP指令集中C类型指令支持外部函数调用，包括libc等库函数调用和JNI函数调用。VMPP的指令转换器为外部函数生成调用表和调用代码，为JNI函数分配好JNIEnv等环境变量。为了支持C++中的异常处理机制，E类型指令通过模拟异常信号的方式处理异常。在解析指令过程中为每个异常分配一个编号，当函数中抛出异常时会抛出该编号，然后在该函数的异常处理位置比较该编号与该处的异常处理类型编号，从而得知该处是否为处理该异常的位置，若是则处理异常，否则返回上一层函数继续处理。

为了防止攻击者逆向攻击指令集，VMPP支持动态生成外部函数调用表(详见2.2.3节)和指令操作码随机生成。VMPP中一条指令的操作码可以随机改变，比如在一个程序中add指令的操作码是0x51，另一个程序时add指令可以变为0x52等任意值；同时，改变之后的指令-操作码映射会被用于生成VMPP的指令解释器，使得指令解释器在运行

表 1 VMPP虚拟指令格式

指令类型	典型指令	1*	2	3	4	5	6	7	8	指令示例	示例含义
R	add	0x51	Dst	Size	–	–	–	Src1	Src2	51 01 04 0000 00 02 03	r1=r2+r3
I	addi	0x30	Dst	Src	Size	Immediate				30 01 02 0400 00 00 01	r1 = r2+1
B	jmp	0x22	Flag	–	PC					22 00 00 0000 00 00 04	jmp 4
CMP	cmp	0xc0	Dst	Mode	Src1	Src2	–	–	–	c0 01 00 0203 00 00 00	r1=r2>r3
W	load	0xe4	Dst	Size	Src	–	–	–	–	e4 01 04 0200 00 00 00	r1=[r2]
C	call	0x90	Num	–	–	–	–	–	–	90 01 00 0000 00 00 00	call 01
M	malloc	0xa0	Dst	–	Immediate					a0 01 00 0000 00 00 08	r1=malloc(8)
E	throw	0x11	Type	–	–	ExceptionPC				11 01 00 0000 00 00 06	throw 01

时可以正确解析指令。这些方法使得不同代码使用的具体虚拟指令集不相同，防止攻击者逆向一个虚拟指令集后自动化逆向所有程序。

2.2 指令转换器

指令转换器将IR文件，转换为虚拟指令集上的虚拟指令，包括数据段、指令段和外部函数调用代码，分别对应于IR中的全局变量、指令和外部函数调用表。

图2展示了一个示例源代码(图2(a))和生成的LLRM IR文件(图2(b))。LLRM IR文件共分为两部分，第1部分为1～2行中以“@”符号开始的变量表示全局变量，对应源代码中的全局变量和静态变量，其中“@a”为未初始化的全局变量，“@.str”为已初始化的静态变量。第2部分为4～15行表示的LLVM IR的指令段，其中第4～8行为内部函数“foo”，第9～15行为开始函数“main”。函数内部的每一行表示一条指令，以“%”开始的符号表示局部变量。指令转换器解析LLVM IR文件时，使用LLVM API依次读取并处理其每条IR数据和指令，分别生成数据段、指令段和外部函数调用代码。

2.2.1 数据段

指令转换器依次读取IR中每个全局变量，解析出该变量的长度与值，将变量的值写入数据段文件中；若该变量未初始化，则将相应长度的0写入数据段文件。同时记录该变量的符号和起始地址，以及其映射关系，这样其他指令可以访问其地址来使用该变量。如图2(b)中第1个变量“@a=global i32 1, align 4”，该变量a长度为4，值为1，起始地址为0。因此，指令转换器会将其值1写入数据段，并记录下符号与地址的映射关系<a, 0>。变量a长度为4，因此下一变量“@.str”起始地址为4，其映射关系为<str, 4>。

2.2.2 指令段

在LLVM IR中一条指令通常由指令操作符、操作数、与标签等组成，在转换IR指令时需要将其一一转换。其中后两者操作数和标签的转换比较直接。IR中操作数若为立即数则直接转换；若为变量(如“%1”等)，VMPP使用256个通用寄存器存储LLVM IR中的变量，指令转换器只需要将IR中的变量转换为对应的寄存器编号，如图2中的main函数中分别为变量“%1”,“%2”分配至1, 2号通用寄存器中。LLVM IR中的标签是用来指示指令跳转的目标地址，在解析标签时只需要将其转换为当前的指令计数(Program Counter, PC)。

LLVM IR中部分指令操作符与自定义虚拟指令是一一对应的，比如store,add等，这些指令直接翻译为VMPP虚拟指令集中对应的指令。有些特殊的指令需要转换为多条指令,比如函数调用指令call。因为涉及到参数传递赋值操作，所以针对IR文件中函数调用指令call需要转换为多条指令。以图2(b)中“%6=call i32 @_Z3fooii(i32 %4, i32 %5)”为例，该条指令为调用内部函数foo，该条指令需要转化为以下4条指令，如图3所示。其中寄存器名称后括号中的值为1代表该寄存器为参数寄存器，值为0则代表该寄存器为通用寄存器，比如“%4(0)”代表“%4”为通用寄存器，“%1(1)”代表“%1”为参数寄存器。

2.2.3 外部函数调用代码

在LLVM IR转换为虚拟指令的过程中，VMPP会为每个源文件动态生成外部函数表和调用代码。比如在某源代码1中只使用了“printf”这一函数，其调用表为“1 <-> printf”；而源文件2中使用了“strlen”和“printf”两个函数，其调用表为“1 <-> strlen; 2 <-> printf”。相应地，其解释器的函数调用代码也会不同，如图4所示。这种动态生成指令解释器的外部函数调用代码的方式，可以显著提高指令解释器逆向难度，使得攻击者难以做到自动化、批量化逆向解释器。

```
(1)   #include <stido.h>
(2)
(3)   int a = 1;
(4)
(5)   int foo( int a, int b){
(6)       return a + b;
(7)   }
(8)
(9)   int main(){
(10)      int b = 2;
(11)      int c = foo(a,b);
(12)      printf("Result: %d\n", c);
(13)      return 0;
(14)  }
```

(a) 源代码文件

```
(1)   @a = global i32 1, align 4
(2)   @.str = private… [12 x i8] c"Result: %d\0A\00", align 1
(3)
(4)   define i32 @_Z3fooii(i32, i32) #0 {
(5)       %3 = alloca i32, align 4
(6)       store i32 %0, i32* %3, align 4
(7)       ...
(8)   }
(9)   define i32 @main() #1 {
(10)      ...
(11)      %6 = call i32 @_Z3fooii(i32 %4, i32 %5)
(12)      %7 = load i32, i32* %3, align 4
(13)      %8 = call i32 (i8*, …) @printf(@.str, i32 %7)
(14)      ret i32 0
(15)  }
```

(b) LLVM IR文件

图 2 示例源代码及其生成的LLVM IR文件

```
mov   %4(0)    %1(1)   //将寄存器4值传入参数寄存器1中
mov   %5(0)    %2(1)   //将寄存器5值传入参数寄存器2中
jmp    foo             //跳转至foo函数位置
mov   %1(1)    %6(0)   //将参数寄存器1值传入寄存器6中
```

图 3 转化后的指令示例

2.3 指令解释器

VMPP指令解释器依次解释执行每一条指令。在每个函数开始，解释器初始化分配256个通用寄存器和16个参数寄存器，然后根据PC依次解析每条指令。对于每一条指令(每8字节)，首先根据第1个字节识别操作码，根据操作码对应的指令格式对该条指令进行解析。比如虚拟指令“51 01 04 00 00 00 02 03”。解释器根据首字节“51”判断出该条指令为add指令，然后根据第2,7,8个字节得出目的寄存器r1,源寄存器为r2和r3，根据第3个字节判断出操作数长度4个字节，故解释器会执行“r1=r2+r3;”。执行完毕后移动PC执行下一条指令。如果当前指令是跳转指令、函数调用指令、函数返回指令，指令解释器需要跳转到目标地址执行。指令解释器最终和被保护代码一起被编译到最终的可执行文件中。

2.4 开发者工具链

安卓提供了本地代码开发工具NDK，这一工具可以编译源代码生成不同架构上的SO文件或者静态链接库文件。VMPP使用NDK导出独立工具链，通过Shell脚本控制编译过程。用户只需指定要加固的源代码文件和目标架构，该工具链便会自动运行VMPP所有流程将本地源代码加固生成数据段、代码段与解释器(包括外部函数调用代码)，并将其打包组合起来生成目标架构上的可执行文件。

图5为工具链打包生成加固后的源代码文件示例，其中第1行代码为引入解释器头文件，以调用解释器相关函数；第3行代码binst数组为虚拟指令段，该数组中每一个值即为一条虚拟指令(十进制)；第5行bglobal数组为虚拟数据段，该数组中每个值为虚拟数据段的一个字节(十进制)；第8行代码为初始化解释器，用于开辟通用寄存器、参数寄

```
1  <->  int printf(char*,int)
```

```
if (callNum = = 1 ){
    // Arg1、Arg2 为参数寄存器 1、2
    char*   arg1 = Arg1;
    int arg2 = Arg2;
    int ret = printf(arg1, arg2);
    //  返回值放入参数寄存器 1 中
    Arg1 = ret;
}
```

(a) 源代码1生成的调用表和外部函数调用代码

```
1  <->  int strlen(const char* str)
2  <->  int printf(char*,int)
```

```
if (callNum = = 1){
    // Arg1 为参数寄存器 1
    const char*   arg1 = Arg1;
    int ret = strlen(arg1);
    //  返回值放入参数寄存器 1 中
    Arg1 = ret;
}else if (callNum == 2){
    ... ...
    int ret = printf(arg1, arg2);
    Arg1 = ret
}
```

(b) 源代码2生成的调用表和外部函数调用代码

图 4 外部函数调用表(上)与解释器的外部函数调用代码(下)

```
(1)    #include “src/inte.h”
(2)
(3)    unsigned long long binst[] = { 72057594037927970, 72057594038452460,
     72057594055229932, 72057594072007404, 28823037615171272, 57646752303424672,
     57646752303424928,57646752303425184, ......};
(4)
(5)    unsigned char bgloble[] = {104, 101, 108, 108, 111, 0, 0};
(6)
(7)    JNIEXPORT jint JNICALL Java_com_dean_vmp01_JNIUtil_addString( JNIEnv* env,
     jobject obj, jstring s ){
(8)        Inte_init();
(9)        *ArgReg = (unsigned long long) env;
(10)       *(ArgReg + 1) = (unsigned long long) obj;
(11)       *(ArgReg + 2) = (unsigned long long) s;
(12)       int PC = 0;
(13)       int result = Interpreter(PC, binst, bgloble);
(14)       return result;
(15)   }
```

图 5 加固后的源文件示例

存器等操作；第9～11行代码将参数依次传入解释器参数寄存器中；第12行代码设置虚拟指令段的入口；第13行调用解释器开始执行虚拟指令。

这些源文件最后会被工具链自动编译生成各个架构的可执行文件，最终运行在不同架构的设备上。

3　实验与评估

本文使用250个C/C++源代码文件和7个真实的安卓应用程序，分别在PC平台和安卓手机上测试了VMPP加固的有效性、防逆向效果、兼容性和运行时开销。PC平台为Ubuntu 16.04, Intel Corei5核心(4核，2.71 GHz)和8 G物理内存；安卓手机有5款，覆盖3种架构。

3.1　加固有效性

为了测试VMPP对本地代码保护的有效性，本文首先从Github上下载了一个小型的编译器测试代码库[15]，里面包含了220个C代码源文件，同时我们又编写了30个C++代码源文件，共计250个C/C++文件。这250个C/C++源代码文件涵盖了C/C++代码的所有基本语法使用，每个源文件代码量从10行到200行不等。经过加固运行测试后发现250个代码文件中231个文件加固成功，19个文件加固失败。由于VMPP目前暂不支持大整数(6个文件)、浮点数(9个文件)以及类运算(4个文件)等操作，所以这些文件加固失败。

本文从开源网站Github上下载(表2中用例A-E)及自己编写(表2中用例F-G)了共7个包含本地代码的安卓程序文件，这些程序中包含AES加密、图片操作、APP签名校验以及密钥生成等常见功能，其中AES加密应用的本地代码量超过2000行，其他应用的本地代码量从50行到300行不等。使用VMPP对上述程序进行加固，发现所有程序均可以成功加固并且正常运行，表明VMPP对于真实程序的加固是有效的。

3.2　防逆向效果

图6和图7分别是VMPP加固前后，com.dean.vmp01程序(用例F)的反汇编代码。VMPP加固之前，其反汇编代码(图6)较为简单，只有一个本地代码函数，无论是直接读汇编代码还是通过读伪代码均极易被逆向分析。经过VMPP加固之后，其反汇编代码(图7)有多个函数，各函数间有大量的跳转或者调用指令，无论是汇编代码或者伪代码的可读性都很差，表明经过VMPP加固的代码安全性有了很大提高。

为了定量地评估VMPP的防逆向效果，本文邀请了3位对安卓本地代码逆向有着丰富经验的安全分析员，对上述程序加固前后的版本进行人工逆向。3位分析员均有丰富的CTF比赛经验，水平接近且独立地进行实验。对于给定二进制文件，分析员需要逆向出代码的数据结构和算法功能。

表3统计了3位分析人员针对测试用例加固前后版本逆向所花的时间。表2中的用例D由于加固后代码过于复杂，没有一个分析人员可以完成逆向任务。在本实验中，分析人员A和B先逆向6个未加固的程序，然后逆向加固后的程序。从表3中可以看出，对于50～300行代码量的简单程序，VMPP加固使得逆向所需平均时间大大增加，分析人员A和B的平均逆向时间为原来的10倍左右。考虑到先逆向未加固代码会对分析员之后逆向加固后的代码有影响，作为对照分析员C先逆向加固后的程序再逆向加固前的程序，则其逆向加固后的时间是加固前的15倍左右。实验表明，VMPP加固可以使得对简单程序的逆向时间增加10倍以上，对复杂程序VMPP加固可以使其无法在常规时间内被人工逆向。因此，VMPP加固可以极大地增加逆向的成本，有效地保护安卓本地代码。

3.3　兼容性

本文在5台不同的安卓设备(如表4所示)上测试上述7个真实的安卓程序。这5台设备涵盖了arm32, arm64以及x86 3种主流的架构，以及Android5.0到Android9.0在内的5个版本的安卓系统。7个程序均

表 2　VMPP有效性测试结果

序号	APP包名	本地代码主要功能	代码行数	运行时间(ms)		二进制体积(kB)	
				加固前	加固后	加固前	加固后
用例A	com.zizuzi.verificationdemo	通过设备ID等生成加密密钥	123	6	7	10	285
用例B	com.masonliu.testndk	计算SHA1值验证签名	230	5	6	18	297
用例C	com.chenneyu.security	反射获取APP签名并校验	221	3	4	11	241
用例D	com.panxw.aes	实现AES算法加密字符串	2301	2	29	20	308
用例E	com.ss.jni	反射Java函数修改界面UI	174	37	42	10	237
用例F	com.dean.vmp01	字符串运算操作	57	<1	<1	6	103
用例G	com.dean.vmp02	多维数组的运算	96	<1	<1	6	92

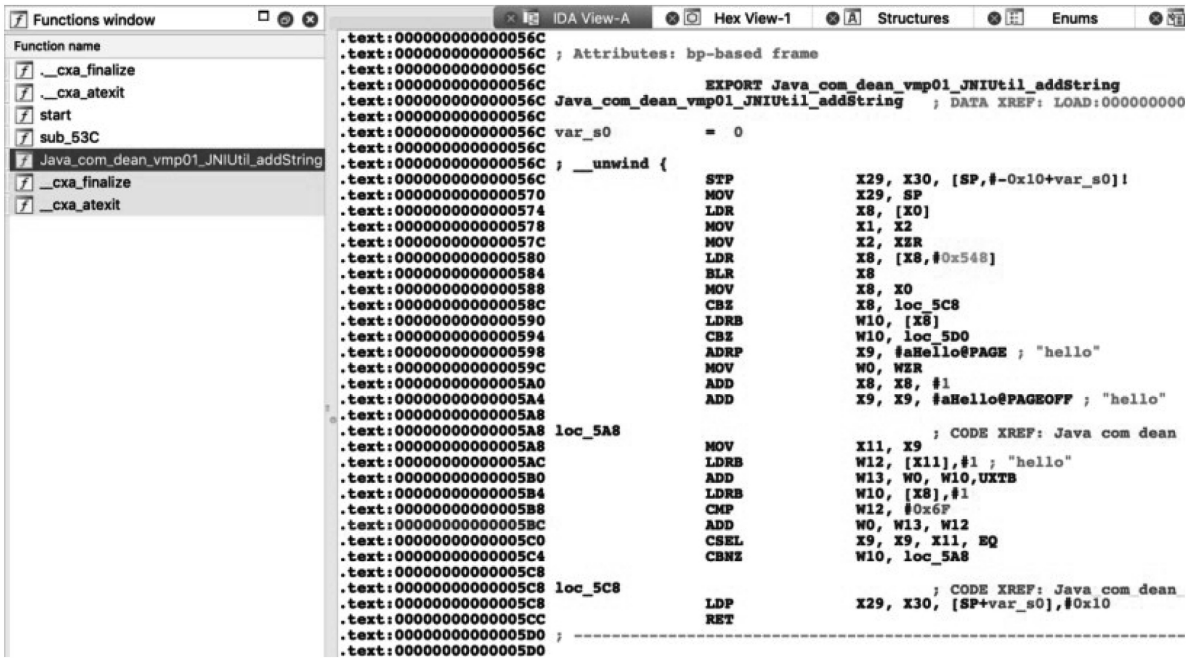

图 6 加固之前源代码生成的可执行文件反汇编结果

图 7 加固之后源代码生成的可执行文件反汇编结果

表 3 VMPP防逆向效果实验

分析人员	逆向所需时间(min)：加固前 / 加固后							加固后、加固前逆向时间比值
	用例A	用例B	用例C	用例E	用例F	用例G	合计	
A	3 / 35	6 / 65	8 / 71	9 / 92	7 / 78	7 / 82	40 / 423	10.6
B	5 / 40	9 / 59	7 / 80	10 / 112	8 / 75	9 / 96	48 / 462	9.6
C	5 / 58	7 / 82	7 / 134	8 / 165	7 / 117	9 / 122	43 / 678	15.8

在5个设备上成功运行，表明VMPP可以兼容不同架构的设备。

3.4 运行时开销

表2中统计了VMPP加固前后在Pixel 2XL上的

表 4　VMPP兼容性测试

序号	设备名称	系统版本	手机架构	是否兼容
1	Nexus 5	Android 5.0	arm32	是
2	Samsung S7	Android 6.0	arm64	是
3	Pixel 2XL	Android 8.1	arm64	是
4	Samsung S9+	Android 9.0	arm64	是
5	Genymotion Emulator	Android 8.0	x86	是

时间和空间开销。加固后大部分应用时间开销小于30%，空间开销在100～200 kB之间。用例D的时间增加了接近14倍，因其本地代码为AES算法，有大量的数学操作运算产生大量数据在寄存器之间移动(mov指令)，导致VMPP在解释执行时耗时较长。本文将VMPP和商用移动应用加固系统几维加固[16]进行了对比，在表2中用例D、用例E两个应用[1)]上进行了加固测试，二者的时间和空间开销如表5所示。可以看到和商用工具进行对比，VMPP的运行时开销在可接受范围之内。

表 5　VMPP加固和几维加固运行时开销对比

序号	APP包名	加固前运行时间(ms)	加固后运行时间(ms)		加固前体积(kB)	加固后体积(kB)	
			VMPP	几维加固		VMPP	几维加固
用例D	com.panxw.aes	2	29	4	20	308	583
用例E	com.ss.jni	37	42	38	10	237	553

3.5　加固安全性讨论

VMPP通过引入虚拟指令解释器可以加固大部分本地代码，包括相对复杂的AES算法以及异常处理等多种机制，上述实验表明了VMPP可以有效保护代码被逆向。目前已有一些对指令解释器进行解混淆的研究，如通过污点分析、符号化执行等方法解混淆虚拟机[17]，或通过自动化沙箱分析技术[18]对加固程序进行行为分析。但是通过对这些工作进行研究，发现这些方法只能在一定程度上进行解混淆，比如相关工作[17]无法处理复杂的循环和内存访问操作，很难应用到有复杂代码逻辑的实际程序中。此外，由于采用了复杂的技术手段，这些方法有较高的开销和时间复杂度，从而提高了解混淆应用的门槛。同时在未来工作中，VMPP原型系统可在当前基础上加入花指令、反调试等保护方法，进一步增强VMPP的加固安全性。

4　结束语

软件代码保护是有效防范逆向攻击的重要一环。本文结合安卓软件需要兼容运行于多种架构的特性，针对安卓本地代码提出了一种基于指令虚拟化技术的加固手段，可以有效抵御逆向攻击，而传统方案受限于加固强度不足或者只适用于单一架构。具体而言，本文基于LLVM IR设计了一套虚拟指令集，实现了虚拟指令解释器以及配套的开发者工具链。在大量C++文件和真实安卓应用程序上的实验表明，本文的方案可以有效地保护本地代码，开销较低且支持多种架构。未来工作中可以引入不定长指令集等指令扩展方法以支持浮点数、大整数、类操作等指令，并且加入花指令、反调试等保护方法进一步增强VMPP的安全性。

[1)]受该加固系统商用收费限制，本文只能在少量应用上进行测试

参 考 文 献

[1] 360安全互联网中心. 2015年Android手机应用盗版情况调研报告[EB/OL]. http://zt.360.cn/1101061855.php?dtid=1101061451&did1101657409, 2019.
360 Security Internet Center. Investigation report on piracy of Android mobile applications[EB/OL]. http://zt.360.cn/1101061855.php?dtid=1101061451&did1101657409, 2019.

[2] HUO Meimei, WU Jianzhong, CAI Jianping, *et al*. An Anti-piracy method based on encryption and dynamic loading for android applications[J]. *Applied Mechanics and Materials*, 2014, 644/650: 2740–2743. doi: 10.4028/www.scientific.net/AMM.644-650.2740.

[3] KIM N Y, SHIM J, CHO S J, *et al*. Android application protection against static reverse engineering based on multidexing[J]. *Journal of Internet Services and Information Security*, 2016, 6(4): 54–64.

[4] FALSINAT L, FRATANTONIO Y, ZANERO S, *et al*. Grab'n run: Secure and practical dynamic code loading for android applications[C]. The 31st Annual Computer Security Applications Conference, Los Angeles, USA, 2015: 201–210. doi: 10.1145/2818000.2818042.

[5] 张震, 张龙. Android平台的Native层加固技术研究与实现[J]. 计算机与现代化, 2016(10): 88–91. doi: 10.3969/j.issn.1006-2475.2016.10.018.
ZHANG Zhen and ZHANG Long. Research and implementation of native layer reinnforcement technology based on android platform[J]. *Computer and Modernization*, 2016(10): 88–91. doi: 10.3969/j.issn.1006-2475.2016.10.018.

[6] 赵奇. 基于LLVM的Android应用代码保护技术研究与实现[D]. [硕士论文], 北京邮电大学, 2018.
ZHAO Qi. Research and implementation of android application code protection based on LLVM[D]. [Master dissertation], Beijing University of Posts and Telecommunications, 2018.

[7] 张一峰, 方勇. 基于LLVM的Android Native文件保护方法[J]. 通信技术, 2017, 50(3): 533–538. doi: 10.3969/j.issn.1002-0802.2017.03.026.
ZHANG Yifeng and FANG Yong. Android native file protection based on LLVM[J]. *Communications Technology*, 2017, 50(3): 533–538. doi: 10.3969/j.issn.1002-0802.2017.03.026.

[8] 胡恒伟. 基于动态虚拟指令集的Android应用保护技术研究[D]. [硕士论文], 南京理工大学, 2018.
HU Hengwei. Research on android application protection technology based on dynamic virtual instruction set[D]. [Master dissertation], Nanjing University of Science and Technology, 2018.

[9] 李振. 基于LLVM的Android应用程序编译时虚拟化保护研究[D]. [硕士论文], 西北大学, 2019.
LI Zhen. LLVM-based android application compiletime virtualization protection method research[D]. [Master dissertation], Northwest University, 2019.

[10] YANG Wenbo, ZHANG Yuanyuan, LI Juanru, *et al*. AppSpear: Bytecode decrypting and DEX reassembling for packed android malware[C]. The 18th International Symposium on Recent Advances in Intrusion Detection, Kyoto, Japan, 2015: 359–381. doi: 10.1007/978-3-319-26362-5_17.

[11] KIM D, KWAK J, and RYOU J. Dwroiddump: Executable code extraction from android applications for malware analysis[J]. *International Journal of Distributed Sensor Networks*, 2015, 11(9): 379682. doi: 10.1155/2015/379682.

[12] 张汉宁. 基于精简指令集的软件保护虚拟机技术研究[D]. [硕士论文], 西北大学, 2010.
ZHANG Hanning. Research on software protection virtual machine technology based on reduced instruction set[D]. [Master dissertation], Northwest University, 2010.

[13] 汤战勇, 李光辉, 房鼎益, 等. 一种具有指令集随机化的代码虚拟化保护系统[J]. 华中科技大学学报: 自然科学版, 2016, 44(3): 28–33. doi: 10.13245/j.hust.160306.
TANG Zhanyong, LI Guanghui, FANG Dingyi, *et al*. A code virtualization protection system with instruction set randomization[J]. *Journal of Huazhong University of Science and Technology*: *Natural Science Edition*, 2016, 44(3): 28–33. doi: 10.13245/j.hust.160306.

[14] 杜春来, 孔丹丹, 王景中, 等. 一种基于指令虚拟化的代码保护模型[J]. 信息网络安全, 2017(2): 22–28. doi: 10.3969/j.issn.1671-1122.2017.02.004.
DU Chunlai, KONG Dandan, WANG Jingzhong, *et al*. A code protection model based on instruction virtualization[J]. *Netinfo Security*, 2017(2): 22–28. doi: 10.3969/j.issn.1671-1122.2017.02.004.

[15] C-testsuite[EB/OL]. https://github.com/c-testsuite/c-testsuite, 2019.

[16] 几维安全. 移动应用加固系统[EB/OL]. https://www.kiwisec.com/product/app-encrypt.html, 2019.

[17] SALWAN J, BARDIN S, and POTET M L. Symbolic deobfuscation: From virtualized code back to the original[C]. The 15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Saclay, France, 2018: 372–392. doi: 10.1007/978-3-319-93411-2_17.

[18] 梁光辉, 庞建民, 单征. 基于代码进化的恶意代码沙箱规避检测技术研究[J]. 电子与信息学报, 2019, 41(2): 341–347. doi: 10.11999/JEIT180257.
LIANG Guanghui, PANG Jianmin, and SHAN Zheng. Malware sandbox evasion detection based on code evolution[J]. *Journal of Electronics & Information Technology*, 2019, 41(2): 341–347. doi: 10.11999/JEIT180257.

张晓寒：男，1991年生，博士生，研究方向为系统与软件安全.
张 源：男，1987年生，副教授，研究方向为系统与软件安全.
池信坚：男，1994年生，硕士生，研究方向为系统与软件安全.
杨 珉：男，1979年生，教授，研究方向为系统与软件安全.

责任编辑：余 蓉