计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202110990
60(2): 326−340, 2023

基于控制流和数据流分析的内存拷贝类函数识别技术

尹小康　芦　斌　蔡瑞杰　朱肖雅　杨启超　刘胜利
（数学工程与先进计算国家重点实验室（信息工程大学）　郑州　450001）
（yxksjtu@sjtu.edu.cn）

Memory Copy Function Identification Technique with Control Flow and Data Flow Analysis

Yin Xiaokang, Lu Bin, Cai Ruijie, Zhu Xiaoya, Yang Qichao, and Liu Shengli
（*State Key Laboratory of Mathematical Engineering and Advanced Computing* (*Information Engineering University*), *Zhengzhou*, 450001）

Abstract　Memory error vulnerability is still one of the most widely used and harmful vulnerabilities in current cyber-attacks, whose timely discovery and repair in binary programs bear great value in preventing cyber-attacks. Memory error vulnerabilities are often associated with the misuse of memory copy functions. However, the current identification techniques of memory copy functions mainly rely on the matching of symbol tables and code feature pattern, which have high false positive and false negative rates and poor applicability, and there are still many problems to be solved. To address the above problems, we propose a memory copy function identification technology CPYFinder, based on the control flow of memory copy functions. CPYFinder lifts the binary code into the VEX IR (Intermediate Representation) code to construct and analyze the data flow, and identifies binary code according to the pattern of the memory copy function on the data flow. This method can identify the memory copy functions in stripped binary executables of various instruction set architectures (i.e. x86, ARM, MIPS and PowerPC) in a short runtime. Experimental results show that CPYFinder has better performance in identifying memory copy functions in C libraries and user-defined implementations. Compared with the state-of-the-art works BootStomp and SaTC, CPYFinder gets a better balance between precision and recall, and has equal time consumption compared with SaTC and its runtime only amounts to 19% of BootStomp. In addition, CPYFinder also has better performance in vulnerability function identification.

Key words　static analysis; data flow analysis; intermediate representation; memory copy function; function identification

摘　要　内存错误漏洞仍是当前网络攻击中造成危害最严重的漏洞之一.内存错误漏洞的产生往往与对内存拷贝类函数的误用有关.目前针对内存拷贝类函数的识别主要借助于符号表和代码特征模式匹配,具有较高的误报率和漏报率,并且适用性较差.提出了一种内存拷贝类函数识别技术CPYFinder（copy function finder）.该技术在内存拷贝类函数控制流特征的基础上,将二进制代码转换为中间语言表示VEX IR（intermediate representation）进行数据流的构建和分析,根据内存拷贝类函数在数据流上的特征进行识别.该技术能够在较低的运行时间下对多种指令集架构（x86, ARM, MIPS, PowerPC）的二进制程序中的内存拷贝类函数进行识别.实验结果表明,相比于最新的工作BootStomp和SaTC, CPYFinder在对内存拷

收稿日期: 2021−10−11; **修回日期**: 2022−02−10
基金项目: 科技委基础加强重点项目（2019-JCJQ-ZD-113）
This work was supported by the Foundation Strengthening Key Project of Science & Technology Commission (2019-JCJQ-ZD-113).
通信作者: 刘胜利（mr_shengliliu@163.com）

贝类函数识别上具有更好的表现，在精准率和召回率上得到更好的平衡，并且运行时间与 SaTC 几乎相等，仅相当于 BootStomp 耗时的 19%.此外，CPYFinder 在漏洞函数识别上也具有更好的表现.

关键词 静态分析；数据流分析；中间表示；内存拷贝函数；函数识别

中图法分类号 TP309

利用漏洞发起网络攻击仍是当前网络攻击中的主要方式.漏洞攻击能够使目标设备瘫痪、实现对目标设备的突破控制，或者对重要文件的窃取.在利用漏洞发起的攻击中，危害性最高的漏洞之一为内存错误漏洞.内存错误漏洞包括内存溢出漏洞、内存泄漏漏洞和内存释放后重用漏洞等，这些漏洞能够实现任意代码执行，造成密钥、口令等信息的泄露.在通用漏洞披露组织(Common Vulnerabilities and Exposures, CVE)发布的 2021 年最危险的 25 种软件脆弱性分析[1]中，CWE-787[2]即越界写(out-of-bounds write, OOBW)排在了第 1 位，CWE-125[3]即越界读(out-of-bounds read, OOBR)排在第 3 位以及 CWE-119[4]即不当的内存缓冲区范围内的操作限制排在第 17 位.内存溢出漏洞和内存泄漏漏洞的发生常常跟内存的拷贝相关，在进行内存拷贝的时候缺少对长度的检查，或者对特殊的字节进行转换时发生长度的变化，都会导致对内存的越界写或者越界读[5].因此内存拷贝类函数(下文简称拷贝类函数)的识别对内存错误漏洞的发现和修补具有重大意义和价值.

本文提出了一种拷贝类函数识别技术 CPYFinder (copy function finder).该技术在不依赖函数名、符号表等信息的情况下，对函数的控制流进行分析，并将二进制代码转换成中间语言代码进行数据流的分析，识别二进制程序中的拷贝类函数.本文的主要贡献包括 4 个方面:

1)分析了不同架构下拷贝类函数的特点，构建了基于静态的分析方法的拷贝类函数识别模型.

2)提出并实现了二进制程序中拷贝类函数识别技术 CPYFinder，该方法不依赖函数名、符号表等信息，能够识别无论是 C 语言库中还是用户自定义实现的拷贝类函数.

3)提出的 CPYFinder 具有良好的适用性和扩展性，通过将二进制代码转换成中间语言代码进行数据流的分析，使得支持 x86, ARM, MIPS, PowerPC 指令集架构的二进制程序.

4)从 GitHub 上收集了拷贝类函数，构建了测试数据集进行测试，并选取了真实的 CVE 漏洞函数进行了测试.实验结果表明 CPYFinder 具有更好的表现，在精准率和召回率上得到更好的平衡，并且具有较低的运行时耗. CPYFinder 对提高下游分析任务具有重大价值.

1 相关工作

拷贝类函数的识别对二进制程序中内存错误漏洞的发现和修补具有重大意义和价值.由于版权或者安全的需要，软件供应商在程序发布的时候往往是以二进制的形式进行发行，甚至会剥除程序中的函数名、符号表等信息.因此安全研究员只能在缺少源码的情况下对二进制程序进行分析，进而发现程序中的脆弱点并提供给供应商进行漏洞的修补.相比于对源代码的分析[6]，由于二进制代码中丢失了高级语言具有的信息，例如 C/C++中的函数原型、变量名、数据结构等信息，因此对二进制程序进行分析具有更大的挑战.针对二进制程序的分析技术主要有二进制代码审计[7]、污点分析[8-9]、符号执行[10-11]等，这些技术在漏洞发现和分析中具有较好的效果.然而这些技术在一定程度上依赖函数名、符号表等信息，例如 Mouzarani 等人[12]在提出的基于混合符号执行的模糊测试技术中仍需要借助符号表对 *memcpy*, *strcpy* 等函数的识别.DTaint[13]和 SaTC[14]等在进行静态污点分析中需要借助函数名，例如 *memcpy*, *strncpy* 等函数，来定位关键函数，进而进行后续的污点记录和传播等操作. 当二进制程序剥除了函数名等信息时，上述技术的效果将会受到严重影响.

此外，当前的研究中还存在一个被忽视的问题：开发者在开发程序时会自定义实现类似于内存拷贝功能的函数，进而会引入内存错误漏洞.例如，漏洞 CVE-2020-8 423[15]发生在一个 TP-Link TL-WR841N V10 路由器设备中，由函数 int *stringModify* (char **dst*, size_t *size*, char **src*)引发的栈溢出漏洞.尽管 BootStomp[8], Karonte[9], SaTC[14]考虑到了对拷贝类函数的识别，但是它们的识别方法依据简单的代码特征：1)从内存中加载数据；2)存储数据到内存中；3)增加 1 个单元(字节 byte、字 word 等)的偏移值.然而满足上述 3 个特征的函数并不一定具有拷贝数据的

功能,而且会遗漏用户自定义实现的拷贝类函数,因此具有较高的误报率和漏报率.

当前针对剥除函数名等信息的二进制程序中的函数识别技术,主要是以静态签名的方法为主,例如IDA Pro中使用库函数快速识别技术(fast library identification and recognition technology, FLIRT)[16],工具Radare2[17]使用Zsignature技术对程序中的函数名进行识别,此类方法能够识别出签名库中包含已经签名的函数,例如*strcpy*, *memcpy*等函数.然而基于签名的函数识别技术容易受编译器类型(GCC, ICC, Clang等)、编译器版本(v5.4.0, v9.2.0等)、优化等级(O0~O3)以及目标程序的架构(x86, ARM, MIPS等)的影响.而且,每种优化等级又可以分为数十种优化选项[18](在GCC v7.5.0中O0开启了58种优化选项,O1开启了92种优化选项,O2开启了130种优化选项,O3开启了142种优化选项),这些优化选项可以通过配置进行手动地开启和关闭.因此,这些选项组合起来将产生成千上万种编译方案,即同一份源码经过不同的编译配置编译后会产生成千上万个函数,但这些函数的静态签名存在一定的差异,给基于签名的函数识别造成了阻碍.因此,为准确地识别函数需要构建各种各样的函数签名,这些工作显得尤为繁重.此外,基于签名的函数识别只能识别已知的函数,对于未知的拷贝类函数(签名库中不包含该函数的签名)则无法识别,这仍是一个亟待解决的问题.

因此,为解决当前研究中存在的依赖函数名等信息、无法识别未知的拷贝类函数以及识别的误报率较高等问题,本文提出了一种新颖的拷贝类函数识别技术CPYFinder,用于对剥除函数名等信息的二进制程序中拷贝类函数的识别.该方法基于拷贝类函数的代码结构特征和数据流特征,通过对函数的控制流[19]和数据流[20]进行分析,识别二进制程序中具有内存拷贝功能的函数.CPYFinder一定程度上避免了编译器和优化等级的影响,不依赖于函数名的信息,并且能够识别开发者自定义实现的内存拷贝类函数,通过将二进制代码转换成中间语言表示(intermediate representation, IR)代码进行数据流的分析,使得CPYFinder适用于x86,ARM,MIPS,PowerPC(PowerPC与PPC等同)等指令集的二进制程序,具有良好的适用性和扩展性,以及较高的准确率.

经过实验评估表明,CPYFinder相比于最新的工作BootStomp和SaTC,在对无论C语言库中的拷贝类函数还是对用户自定义实现的拷贝类函数的识别上都具有更好的表现;在精准率和召回率上得到更好的平衡.在实际的路由器固件的5个CVE漏洞函数测试中,BootStomp和SaTC均未发现导致漏洞的拷贝类函数,而CPYFinder发现4个漏洞函数.在识别效率测试中发现,CPYFinder具有更高的识别效率;在增加数据流分析的情况下与SaTC耗时几乎相同,耗时仅相当于BootStomp的19%. CPYFinder能够为下游的分析任务,例如污点分析[21]、符号执行[12]、模糊测试[22]等提供支持,在对内存错误漏洞的发现和检测上具有较高的价值.

2 问题描述及相关技术

本节主要介绍相关的定义、结合具体的实例对拷贝类函数识别的重要性进行介绍、对现有方法存在的问题进行分析以及对VEX IR中间语言进行简要介绍.

2.1 相关概念

1)内存拷贝类函数(memory copy function)[8]. 将数据从内存中的一段区域(源地址)直接或者经过处理(例如对字节进行转换)转移到另一段内存区域(目的地址)中的函数,或者部分代码片段实现了内存拷贝功能的函数.

2)函数控制流图(control flow graph, CFG). 函数方法内的程序执行流的图,是对函数的执行流程进行简化而得到,是为了突出函数的控制结构.本文以G_c表示程序的控制流图,以V表示节点的集合,以E表示边的集合.其中$G_c=(V,E)$, $V=\{v_1, v_2, \cdots, v_n\}$, $E=\{e_1, e_2, \cdots, e_m\}$.需要注意的是CFG图是一个有向图.通过对CFG的遍历来判断其是否包含循环结构.

3)循环路径(loop path, LP). 从图中的一个节点出发,沿着其相连接的边进行遍历,若还能回到这个节点,则该图包含循环结构,其中从该节点出发又回到该节点的所有节点及其边构成了循环的路径.对于CFG来说,循环路径更多关注的是其路径上的节点,即基本块和基本块内的指令,该指令序列组成了函数的执行流.

4)数据流图(data flow graph, DFG). 是记录程序中的数据在内存或者寄存器间的传播和转移变化情况的图.通过对变量或者内存的数据流进行跟踪分析能够判断函数的行为.

对二进制程序中拷贝类函数识别的第1步是二进制函数边界的识别,二进制函数边界识别的技术[23-24]已经相对成熟,以及现有的IDA Pro工具已经满足需要,这里不再赘述.

2.2 拷贝类函数识别的重要性

据本文研究发现，C 语言库中封装的拷贝类函数并不能满足所有开发的需求，例如需要在内存拷贝时对字节进行处理或者转换时无法再使用 C 语言库中的拷贝类函数，开发者只能自己开发相应功能的函数来满足需求.此类拷贝类函数仍是安全研究需要关注的重点，对此类函数的不正确使用仍会造成内存错误漏洞的产生.

如图 1 所示，函数 *alps_lib_toupper* 在拷贝的时候将所有的小写字母转换为大写字母，开发者调用此函数时缺乏对长度的检查，导致了漏洞 CVE-2017-6736[25] 的产生；另一个例子是 MIPS 架构下 TP-Link WR940N 无线路由器中的一个栈溢出漏洞 CVE-2017-13772[26]，如图 2 所示，导致该漏洞的是函数 *ipAddrDispose* 中的一段负责内存拷贝（地址转换）的代码，导致该漏洞产生的基本块路径是 loc_478568, loc_478550, loc_478560, 该循环在一定条件下从寄存器$a1 指向的内存中取出 1B 的数据存放到寄存器 $v0 中，当$v0 的值与$t0 的值不相等时，将寄存器 $v0 中的数据存放到 0x1C($a2) 指向的内存中，由于缺少对长度严格的检查导致了漏洞的产生.

```
void alps_lib_toupper(char *dst,
    const char *src, unsigned short len)
{
    unsigned short ii;
    for (ii = 0; ii < len; ++ii) {
        dst[ii] = toupper(src[ii]);
    }
    dst[len] = 0x00;
}
```

Fig. 1 Memory copy function implemented by developer

图 1 开发者实现的内存拷贝类函数

由此可见，不仅对 C 语言库中拷贝类函数的错误调用会导致内存错误漏洞，对开发者自定义实现的拷贝类函数调用也存在产生内存错误漏洞的风险.因此本文提出通过识别二进制程序中的拷贝类函数对具备内存拷贝功能的代码片段进行检测，以便为下游的分析任务，例如污点分析、模糊测试等提供更多的支持，提高分析的准确率和发现内存错误漏洞的可能性.

2.3 拷贝类函数的特点

本节将对拷贝类函数的特点、不同指令集下的变化以及检测的难点进行分析.

在 C 程序开发时，一般情况下开发者会直接调用库中封装好的内存拷贝类函数，如 *strcpy*, *memcpy* 等函数.图 3 展示了 C 语言下函数 *strcpy* 的源码经过

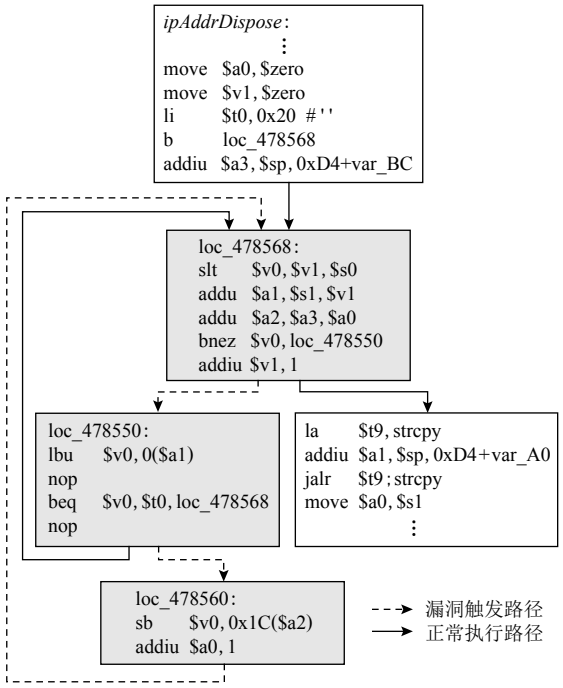


Fig. 2 Function for IP address conversion in the httpd service

图 2 httpd 服务中 IP 地址转换的函数

编译后的二进制代码.图 4 展示了 MIPS 指令集下的函数 *strncpy*.从源码中可以看出，对于内存中数据的转移或者修改往往借助于 while 或者 for 循环进行实现，经过编译器编译后在二进制函数 CFG 中的表现即为存在循环路径.

```
char *strcpy(char *dst, const char *src)
{
    while ((*dst++ = *src++) != '\0')
        return dst;
}
```

Fig. 3 Implementation of *strcpy* function in C library

图 3 C 语言库中 *strcpy* 函数的实现

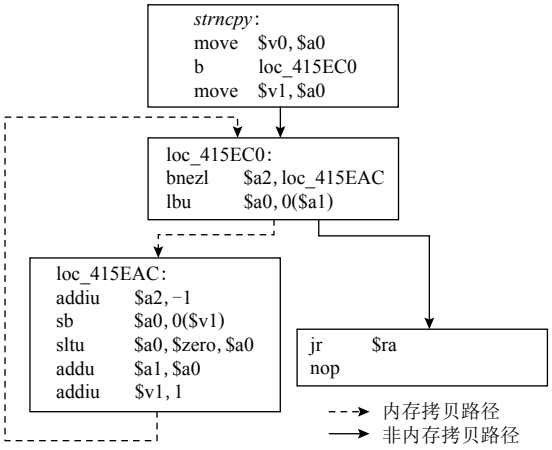


Fig. 4 The *strncpy* function under the MIPS instruction set

图 4 MIPS 指令集下的 *strncpy* 函数

然而并不是所有的内存拷贝函数都存在循环路径，如图 5 所示为 x86 指令集下的函数 *memcpy*，从图中可以看出该函数不存在循环路径，原因是 x86 指令集下存在特殊的指令可以直接实现字节的连续转移，如 rep movsb，rep movsd 指令等，即由于 x86 指令集下存在 rep 指令可以完成重复的功能，借助此指令来完成内存的拷贝.但由于其他指令集下，如 ARM，MIPS，PPC 不存在这种特殊的指令，则需要借助循环来实现内存的拷贝.此外，由于 rep 只能实现无变化的数据拷贝，因此 x86 指令下二进制同样存在基于循环实现的内存拷贝类函数.因此 CFG 中存在循环路径仍是拷贝类函数最大的特点.内存拷贝类函数一定存在对内存的访问，即对内存进行读写，因此在循环路径中一定要存在对内存的读取和写入指令，在反汇编代码中即表现为存在内存的加载和存储的操作码，例如在 MIPS 指令下为 lbu 和 sb(如图 4 所示，在基本块 loc_415EC0 和基本块 loc_415EAC 中)；在数据流的层面表现为，字节从内存的一个区域流向了另一段内存区域.以 MIPS 下的函数 *strncpy* 为例，如图 4 所示，即字节从寄存器$a1 指向的内存流向了寄存器$v1 指向的内存中，此过程未对数据进行修改.

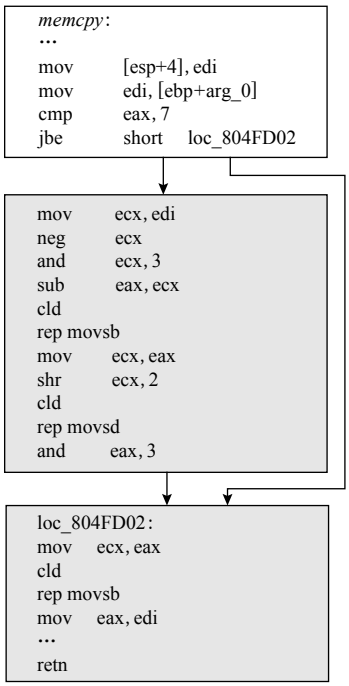


Fig. 5 The *memcpy* function under the x86 instruction set

图 5 x86 指令集下的 *memcpy* 函数

拷贝类函数的另一个特点就是存在偏移的更新，即需要对内存地址进行更新，以便将数据储存到连续的内存单元.以简单的拷贝类函数为例，如图 4 所示，即存在对寄存器$v1(addiu $v1, 1)的更新.然而，自定义实现的拷贝类函数可能包含更多的复杂操作以及分支判断，导致循环路径上可能包含多个基本块，并不是如图 4 中所示的那样只有两个基本块，并且编译器的优化也对函数造成一定的影响，数据流会经过多次的转移变化，因此拷贝类函数并不是像 BootStomp 以及 SaTC 中所提的特征那么简单，对其的检测需要更准确的数据流特征.

2.4 现有相关方法及其不足

基于循环路径的溢出漏洞检测在二进制程序分析中和源码分析中都存在诸多的应用，例如 2012 年，Rawat 等人[27]提出检测二进制程序中由循环路径导致的溢出漏洞(buffer overflow inducing loops, BOILs)，实现了一个轻量级的静态分析工具来检测 BOILs.然而此方法未对数据流进行分析，仅依靠 BOILs 的特征进行检测，并且只支持 x86 的二进制程序.2020 年，Luo 等人[28]提出在源码层面检测由循环路径导致的溢出漏洞，由于该方法针对的是源代码，而二进制程序丢失了源码中较多的语义、变量类型、结构体等信息，难度更大，该方法不适用于对二进制程序的分析.

随着静态污点分析技术和物联网技术的发展，研究者开始将静态污点分析应用到对固件的脆弱性检测中，如 Redini 等人先后在 2017 年和 2020 年分别提出 BootStomp[8] 和 Karonte[9]，使用静态污点分析来检测安卓手机中 bootloader 的脆弱性和嵌入式设备系统中，如路由器、网络摄像头等由于多二进制交互而产生的漏洞.如图 6 所示，Karonte 仍需要借助函数名的信息.2021 年，Chen 等人[14]提出使用静态污点分析检测嵌入式设备中与前端关键字关联的漏洞技术 SaTC，如图 7 所示，借助函数名信息对嵌入式设备的固件进行静态分析依赖于对拷贝类函数的识别.然而当前的方法依赖于函数名等信息，并且仅仅通过简单的特征匹配模式进行检测，存在较高的漏报率和误报率.

```
Karonte(S&P20)
1 def get_memcpy_like(p):
2   addrs = get_dyn_sym_addrs(p, [ 'strcpy' ])
3   summarized_f = {}
4   for f in addrs
5     summarized_f[f] = summary_functions.memcpy_unsized
6   addrs = get_dyn_sym_addrs(p,[ 'strncpy' , 'memcpy' ])
7   for f in addrs
8     summarized_f[f] = summary_functions.memcpy_sized
9 return summarized_f
```

Fig. 6 Static taint analysis in Karonte relies on function name information

图 6 Karonte 中静态污点分析依赖于函数名信息

```
SaTC(Usenix21)
1 def get_memcpy_like(p):
2   summarized_f = {}
3   addrs = get_dyn_sym_addrs (p, [ 'sprintf' ])
4   for f in addrs
5     summarized_f[f] = summary_functions.snprintf
6   addrs = get_dyn_sym_addrs (p, [ 'snprintf' ])
7   for f in addrs
8     summarized_f[f] = summary_functions.snprintf
9   addrs = get_dyn_sym_addrs (p, [ 'strcpy' , 'stristr' ])
10  for f in addrs
11    summarized_f[f] = summary_functions.memcpy_unsized
12  addrs = get_dyn_sym_addrs (p, [ 'strncpy' , 'memcpy' ])
13  for f in addrs
14    summarized_f[f] = summary_functions.memcpy_sized
15  return summarized_f
```

Fig. 7 Static taint analysis in SaTC relies on function name information

图 7 SaTC 中静态污点分析依赖于函数名信息

本文研究发现，由于拷贝类函数多种多样，仅仅依靠模式匹配很难识别出拷贝类函数.因此本文在CFG 分析和模式匹配的基础上，增加对函数的 DFG 的分析，通过对 DFG 的分析来提高拷贝类函数识别的准确率，降低误报率和漏报率，并且为了支持多指令集架构，将不同指令的二进制代码转换为 VEX IR 代码进行分析，使得该方法具有较高的适用性.

2.5 VEX IR 中间语言

由于 IR 能够保留指令的语义信息，具有出色的可拓展性，可以将不同指令集的代码进行归一化，被广泛应用于跨指令集的程序分析[29-30].其中较为著名的是 VEX IR，由于其支持的指令集架构相对齐全，并且提供了 Python 的 API[31]，被较多的二进制分析工具 angr[32]，MockingBird[33]，Binmatch[34] 使用.为了能够支持对多指令集架构的二进制程序进行分析，本文选用 VEX IR 作为中间语言对函数的数据流进行分析.

本节对 VEX IR 的指令类型进行了梳理，并给出了对应的操作码和示例，结果如表 1 所示.将 VEX IR 指令分为 8 种类型，分别为寄存器访问、内存访问、算术运算、逻辑运算、移位运算、转换、函数调用以及其他指令.其中，寄存器访问指令是对寄存器的读取和写入，内存访问指令是从内存中加载数据和将数据存储到内存中；函数调用指令会修改 *pc* 寄存器的值，并给出一个关键字 Ijk_Call；其他指令包括比较指令以及其他不常见的指令，如 ITE(if-then-else). 通过对 VEX IR 指令的分析获取变量间的数据流动.

Table 1 Instruction Types and Examples of VEX IR

表 1 VEX IR 的指令类型及示例

指令类型	VEX IR 操作码	示例
寄存器访问	GET, PUT	PUT($r7$) = $t2$
内存访问	LDb(l)e, STb(l)e	STbe($t22$) = $t23$
算术运算	Sub, Add, Mul, Div	$t6$ = Add16($t4$,0x001)
逻辑运算	Xor, Not, Or, And	$t2$ = Not32($t3$)
移位运算	Shl, Shr	$t57$ = Shl32($t54$,0x02)
转换	*to*, *Uto*, *Sto*	$t16$ = 1Uto32($t17$)
函数调用	Ijk_Call	PUT(pc) = 0x11008
其他指令	Cmp, ITE, Ctz, Clz, if	$t10$ = CmpNE8($t6$,$t9$)

3 拷贝类函数识别

拷贝类函数识别是给定一个二进制函数(汇编代码或二进制代码)，在不借助符号表的情况下，通过静态分析技术或者动态分析技术来判断该函数是否具有内存拷贝的功能. 二进制程序中拷贝函数识别是给定一个含有n个函数的二进制程序 B(剥离或者保留函数名等信息)，即$B=\{f_1, f_2, \cdots, f_i, \cdots, f_n\}$，其中$f_i$为包含$m$个基本块的二进制函数(通常以函数的起始地址命名)，$f_i=\{b_1, b_2, \cdots, b_j, \cdots, b_m\}$，$b_i$为函数 f_i 的基本块.用$L=\{b_k|b_k \in f_i \wedge k \in LP\}$表示参与循环的基本块，其中$k$为基本块编号，$LP$为循环的路径，由 CFG 遍历算法获得保存着参与循环的基本块的编号. 因此当$LP=\varnothing$时，函数不为拷贝类函数(x86 指令集除外). 对二进制程序中每个函数进行识别，判断是否为拷贝类函数，输出该二进制程序中对所有函数的识别结果.

本文对具备拷贝类功能的函数研究发现，具备拷贝功能的函数可能不单单完成一项任务，即将数据从一段内存区域转移到另一段内存区域，它可能具备更多其他的功能，例如数据转移后的处理.因此可以将拷贝类函数分为粗粒度的拷贝类函数和细粒度的拷贝类函数. 细粒度的拷贝类函数只完成内存拷贝的功能；粗粒度的拷贝类函数除了做内存拷贝的功能外，还存在更多的功能. 因此为减少待分析函数的数量，可以通过对函数的复杂程度进行过滤，例如以基本块的数量进行过滤，通常情况下细粒度的拷贝类函数的基本块数量小于 50(这里阈值的设置根据对当前的拷贝类函数分析所得，实际应用中可以根据具体的后续任务需求设置阈值进行过滤).

4 拷贝类函数识别技术

本节对二进制程序中拷贝类函数识别技术进行详细介绍. 二进制程序中拷贝类函数识别技术 CPYFinder 的流程如图 8 所示：

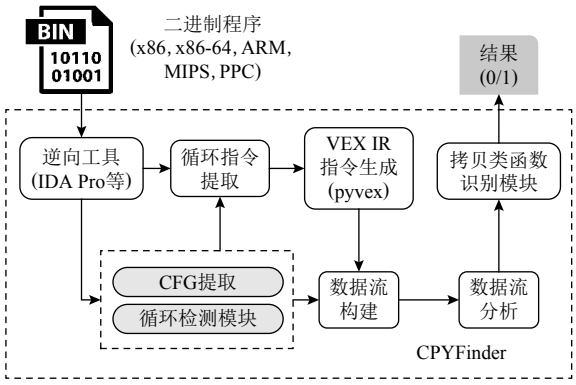


Fig. 8 Workflow of memory copy function identification

图 8 内存拷贝类函数识别的流程

CPYFinder 总共分为 6 个模块: CFG 提取和循环检测模块、循环指令提取模块、VEX IR 指令生成模块、数据流构建模块、数据流分析模块和拷贝类函数识别模块. 其中: CFG 提取和循环检测模块主要分析二进制程序中是否包含循环路径; 循环指令提取模块借助于 IDA Pro 等逆向分析工具从二进制程序中提取循环路径上的指令; VEX IR 指令生成模块借助于 pyvex 工具将指令转换为中间语言指令, 方便系统支持对多种指令集架构的程序进行分析; 数据流构建和数据流分析模块从 VEX IR 指令提取变量间的数据关系, 并对其进行分析; 拷贝类函数识别模块依据拷贝类函数数据流的特点对函数循环路径上的数据流进行判断. 具体的拷贝类函数识别算法如算法 1 所示.

算法 1. 拷贝类函数识别算法.

输入: 函数起始地址;

输出: 函数是否为拷贝类函数(值为 0 或 1).

① *func* = *get_func*(*ea*);

② *blocks* = [*v* for *v* in *FlowChart*(*func*)];

③ *cfg* = *get_cfg* (*blocks*); /*生成控制流图*/

④ *loops* = *simple_cycles*(*cfg*); /*遍历 CFG 获取循环路径*/

⑤ if *loops* ≠ []

⑥ *vex_irs* = *generate_vex_ir*(*blocks, loops*); /*将循环路径上的指令转换为 VEX IR 指令*/

⑦ *data_flows* = *get_data_flow*(*vex_irs*); /*在 IR 指令的基础上生成数据流*/

⑧ if *is_copy_func*(*data_flows*) /*对数据流进行分析, 判断是否为拷贝类函数*/

⑨ return 1;

⑩ end if

⑪ end if

⑫ return 0.

4.1 函数控制流的生成及循环识别

在第 2 节中, 本文对拷贝类函数的特点进行了分析, 除了 x86 指令集下存在使用 rep movsb 等实现内存拷贝的特殊指令(x86 下需要添加对 rep movsb 指令进行检测), 在 ARM, MIPS, PowerPC 指令集下拷贝类函数均需借助循环进行内存数据的转移. 因此, 对循环的检测是拷贝类函数识别的基础.CPYFinder 首先对二进制函数的 CFG 进行提取, 以判断函数执行流中是否存在循环, 在非 x86 指令集下, CFG 不包含循环则直接视为非拷贝类函数.

CFG 中保存着函数内基本块间的关系, 通过对汇编指令的解析可以获取基本块之间的关系, IDA Pro 提供了 API 即函数 *FlowChart*() 来获取二进制函数的基本块以及基本块间的关系, 因此本文直接借助 IDA Pro 获取基本块间的关系.为了快速地生成图以及对后续的处理, 选择使用 networkx 库[35](用于创建、操作和处理复杂网络的 Python 库).通过将基本块间的关系输出到 networkx 的创建图的 API 函数 *DiGraph*() 中, 生成函数的 CFG(算法 1 行②和③).

在获得了函数的 CFG 后, 为进一步判断是否为拷贝类函数, 需要判断 CFG 是否包含循环.由于判断图结构中是否包含环, 以及对算法的效率的分析是图论中研究的内容, 这里不再做探讨.此外, 判断图结构中是否包含环的算法已经十分成熟, 这里将利用 networkx 提供的函数 *simple_cycles*() 判断 CFG 是否包含循环路径以及生成循环路径, 用于后续循环内数据流的生成(算法 1 行④).当 CFG 包含循环路径时, 进行后续的拷贝类函数的识别; 当 CFG 中无循环路径, 并且该二进制程序非 x86 指令集, 认为此函数为非拷贝类函数(对于 x86 指令集的函数, 如果指令中存在 rep movsb 等指令, 认为是拷贝类函数).

4.2 基于 VEX IR 的数据流生成与分析

在获取了循环路径$L=\{b_k|b_k\in f_i\wedge k\in LP\}$后, 为了支持多指令集架构并方便后续的处理, 将基本块的指令全部转换为 VEX IR 指令, 然后对 VEX IR 指令进行过滤分析, 构建变量之间的数据关系(算法 1 行⑥和⑦).如图 9 所示, 为 ARM 指令集下二进制字节码 E5 F1 E0 01 生成的汇编代码(图 9 行 1)和 VEX IR 代码(图 9 行 3~10). CPYFinder 依据表 1 中指令的类型, 使用正则表达式对指令的变量进行提取, 将指令中的变量分为目的和源, 即 (*dst*, *src*), 其中由于存在指令对多个变量进行操作, 例如 Add 操作, 因此, 源操作数 *src* 又记录为 (src_1, src_2), 然后根据指令的类型

和特点对源变量和目的变量构建数据流关系.在整个变量关系记录过程中，只记录变量之间的关系，不记录常量.以图 9 的行 4 为例说明，数据流动方向为从 *r*1 到 *t*18（VEX IR 的临时变量以 *t* 开头），行 5 数据流动方向为从 *t*18 流向了 *t*17，不记录常量 0x00000001 流向了 *t*17，依次构建所有的变量之间的关系. 由于 *pc*, *eip* 等特殊的寄存器与拷贝类函数的判断无关，因此在进行变量关系生成时会将此类的寄存器过滤掉，不做处理，图 9 中变量的关系记录为 {(*t*18, *r*1), (*t*17, *t*18), (*t*20, *t*17), (*t*38, *t*20), (l*r*, *t*38), (*r*1, *t*17)}.

```
1  .text:0001059C LDRB    LR, [R1,#1]!
2                  VS
3   | ------ IMark(0x1059c, 4, 0) ------
4   | t18 = GET:I32(r1)
5   | t17 = Add32(t18, 0x00000001)
6   | t20 = LDbe:I8(t17)
7   | t38 = 8Uto32(t20)
8   | PUT(1r) = t38
9   | PUT(r1) = t17
10  | PUT(pc) = 0x000105a0
```

Fig. 9　Assembly code and VEX IR code

图 9　汇编代码和 VEX IR 代码

在对所有的 VEX IR 代码遍历的同时记录直接参与内存访问（加载指令点记录为 *LD* 点，存储指令点记录为 *ST* 点）和参与算术运算的变量，例如图 9 中的行 5 和行 6，其中行 5 为加运算操作，行 6 为从 *t*17 指向的内存单元中加载数据.在对所有的 VEX IR 指令遍历后，生成了所有变量之间的关系、内存加载变量集合、内存存储变量集合、算术运算变量集合.图 9 中的所有数据为变量关系 {(*t*18, *r*1), (*t*17, *t*18), (*t*20, *t*17), (*t*38, *t*20), (l*r*, *t*38), (*r*1, *t*17)}、内存加载集合为 [*t*20]、内存存储集合为∅、算术运算变量集合为 [*t*17]. 由于图 9 中 VEX IR 指令中无 *ST* 指令，因此内存存储集合为空.

在获取上述数据后，将所有变量之间的关系输入到函数 *DiGraph*() 中构建 DFG.经过大量的分析发现，拷贝类函数的 DFG 具有 5 个特征：

特征 1. 数据流图上存在 *LD* 点；

特征 2. 数据流图中存在 *ST* 点；

特征 3. 存在从 *LD* 点到 *ST* 点的路径，并且 *LD* 点先于 *ST* 点出现；

特征 4. 数据流图上存在环结构，并且环上存在算术运算；

特征 5. 环上的点能够到达特征 3 上的 *ST* 点.

基于 5 个特征，对函数的循环内的数据流进行分析，数据流满足 5 个特征的函数被认为是拷贝类函数. 这里以存在 *LD* 点和 *ST* 点（特征 1 和特征 2）的数据流图为例进行具体识别过程的描述.使用 networkx 的路径通过函数 *all_simple_paths*() 来判断 DFG 中是否存在 *LD* 点到 *ST* 点的路径（特征 3），如果不存在该路径，则认为不满足拷贝类函数数据流的特征. 当从 *LD* 点到 *ST* 点存在路径时，则进一步判断 DFG 中是否存在环结构，并且环上节点的关系是否存在算术运算（特征 4），如果满足特征 4，就继续判定环上的点能否到达满足特征 5 的 *ST* 点，如果满足上述所有特征，认为此函数为拷贝类函数. 由于函数内可能存在多条循环路径，因此只要存在 1 条循环路径的 DFG 满足 5 个特征，则认为此函数为拷贝类函数.

4.3　对特殊拷贝类函数的处理

正如第 2 节中的函数 *alps_lib_toupper* 存在函数在循环块内调用其他函数的情况，因此，本文认为在内存拷贝中调用了其他函数的函数为特殊拷贝类函数.此类函数由于函数的调用会导致数据流的中断，因此需要对变量间的数据流进行连接，即在函数的参数与函数的返回寄存器之间建立数据流关系.当发现循环中存在函数调用时，对被调用函数的参数传递情况及返回值寄存器使用情况进行分析；当函数调用后存在返回值寄存器被使用的情况，则将被调用函数参数与返回值寄存器之间构建数据流，当前方法中是将函数的第 2 个参数与函数返回值寄存器建立数据流.如果返回值寄存器未被使用，则直接在第 2 个参数和第 1 个参数间建立数据流关系（一般情况下，目的寄存器是第 1 个参数，源地址寄存器是第 2 个参数，返回值寄存器的值会指向目的寄存器）.对于不同的指令集，其参数的传递方式和结果返回的方式（返回值寄存器）均不同，因此需要对各个指令集根据其参数传递方式的约定进行数据流的连接，对于 x86 指令集则根据其利用栈进行参数传递的方式，追溯压入栈的指令和变量，与返回值寄存器进行连接；对于 ARM, MIPS, PPC, 由于都是首先使用寄存器进行参数的传递，参数寄存器不足时采用栈进行参数传递.研究发现，通常情况下参数寄存器即可满足此类拷贝类函数中的参数传递，因此，直接在参数寄存器之间和返回值寄存器之间或者参数寄存器之间建立数据流关系，以避免数据流的中断，导致对特殊拷贝类函数的遗漏.

5　实验评估

本节从不同的角度利用 CPYFinder 对拷贝类函数识别的效果进行评估，并与现有方法 BootStomp,

Karonte, SaTC 进行对比, 包括对库函数识别效果、自定义函数识别效果、多架构的支持、受编译器的影响和编译优化等级的影响、在实际的漏洞函数检测中的效果以及识别的运行效率等进行评估.

5.1 实验设置和数据集

1)实验环境.实验所使用计算机的配置为 Intel® Core™ 6-core、3.7 GHz i7-8700K CPU 和 32 GB RAM. 软件为 Python(版本为 3.7.9)、pyvex(版本为 9.0.6136)、networkx(版本为 2.5)、IDA Pro(版本 7.5)以及基于 buildroot 构建的交叉编译环境用于生成不同架构的二进制程序.

2)对比方法. BootStomp(Usenix17), Karonte(S&P20), SaTC(Usenix21)(由于 Karonte 与 SaTC 对拷贝类函数实现完全相同, 本文只与 SaTC 进行对比), 经过分析发现由于 SaTC 借助于工具 angr 来识别函数的参数个数, 在识别过程中限制了参数的个数 n 即 $2 \leqslant n \leqslant 3$, 由于 angr 对参数个数错误的识别, 导致很多拷贝类函数被过滤掉, 例如函数 *strcpy* 实际上存在 2 个参数, 而 angr 识别出来的参数个数为 4, 因此, 本文对 SaTC 进行修改后, 取消其对参数个数限制的方法为 SaTC+.

3)工具实现.CPYFinder 基于 IDAPython 和 pyvex 进行实现, 并借助于 network 对 CFG 和 DFG 进行处理.表 2 展示各个方法实现主要依赖的工具和支持的指令集.

Table 2 Comparison of Existing Methods and CPYFinder

表 2 现有方法与 CPYFinder 对比

方法	依赖工具	支持的指令集
BootStomp	IDA Pro	ARM
Karonte	angr, pyvex	ARM, MIPS
SaTC	angr, pyvex	ARM, MIPS
CPYFinder	IDA Pro, pyvex	x86, ARM, MIPS, PPC

4)数据集.本文首先对 C 语言库中的拷贝类函数进行分析, 根据其应用程序编程接口(application programming interface, API)编写一个主程序来调用所有 string.h 和 wchar.h 中的函数拷贝类函数, 如表 3 所示, 列出了 C 语言库中的拷贝类函数, 其中 string.h 为普通拷贝类函数, wchar.h 为宽字符拷贝类函数, 用于测试不同工具的效果.此外为验证对用户自定义实现(非 C 语言库中)的拷贝类函数的识别效果, 本文从 Github(Netgear R7000 路由器代码[36]以及 Mirai 代码[37])上搜集了 22 个此类的函数, 整合到一个 C 程序中进行编译测试.

Table 3 Memory Copy Functions Used in the Experiment

表 3 实验中使用的内存拷贝类函数

来源文件	函数
string.h	*memcpy*, *memmove*, *strcpy*, *strncpy*, *strcat*, *strncat*, *strxfrm*
wchar.h	*wcscat*, *wcscpy*, *wcsxfrm*, *mbsnrtowcs*, *wcsncpy*, *wmemmove*, *wcsnrtombs*, *wcsncat*, *wmemcpy*
Netgear R7000 及 Mirai	*my_copy*, *yystpcpy*, *yy_flex_strncpy*, *zmemcpy*, *stpcpy*, *unistrcpy*, *BUF_strlcpy*, *util_memcpy*, *tr_strlcpy*, *g_strlcpy*, *wxStrncpy*, *wxStrcpy*, *wxStrcat*, *w_copy*, *strlcpy*, *MD5_memcpy*, *resolv_domain_to_hostname*, *alps_lib_toupper*, *strncpy_w*, *sstrncpy*, *alpha_strcpy_fn*, *StrnCpy_fn*

5)评估标准.本文使用精准率(precision, P), 以及召回率(recall, R)来评估本文所提方法的效果, 精准率和召回率越高效果越好.

$$P = \frac{TP}{TP + FP}, \tag{1}$$

$$R = \frac{TP}{TP + FN}. \tag{2}$$

TP 为将拷贝类函数识别为拷贝类函数的数量; FP 为将非拷贝类函数识别为拷贝类函数的数量; FN 为将拷贝类函数识别为非拷贝类函数的数量.

6)编译器以及优化等级.如表 4 所示为实验评估中使用的编译器、编译器类型及优化选项, 使用 GCC 和 Clang 这 2 个编译器进行测试, 并使用 buildroot 构建了不同指令集的交叉工具链, 用于生成不同目标架构的程序(在当前的 Clang 编译器中, O3 优化等级与 O4 优化等级仍相同).

Table 4 Compilers Used in the Evaluation Experiments

表 4 评估实验中所使用的编译器

编译器	编译器版本	优化等级
GCC	4.5.4, 4.6.4, 4.7.4, 4.8.5, 5.4.0, 5.5.0, 6.5.0, 7.5.0	O0~O3
Clang	3.9.1, 4.0.1, 5.0.1, 6.0.1	O0~O4

5.2 实验结果

5.2.1 对 C 语言库中拷贝类函数的识别效果

为了验证对 C 语言库中拷贝类函数的识别效果, 本文编写 C 代码, 调用 string.h 和 wchar.h 库中的函数来编译成不同的二进制程序(由于 BootStomp 只支持 ARM 指令), 这里使用静态方法将源代码编译成 ARM 架构的二进制程序, 使用的编译器版本为 4.5.4, 4.6.4, 4.7.4, 4.8.5, 优化等级为 O0, 此外由于 5.0 以上版本的 GCC 使用的 libc 库在拷贝类函数的实现, 不同的拷贝类函数会调用同一个函数, 导致拷贝类函数的种类下降, 例如 *strcpy*, *strncpy* 的拷贝功能的实现是直接调用函数 *memcpy* 进行的, 这些函数本身不存在拷贝类函数的特点, 只是函数 *memcpy* 的封装,

因此这里未使用高版本的编译器.各个方法的识别效果如表 5 所示，从表 5 中可以看出，CPYFinder 的效果优于其他 3 种方法，即 BootStomp, SaTC 和 SaTC+.尽管 BootStomp 具有较高的精准率 *P*，但是召回率 *R* 却不高于 0.5，尽管 SaTC+识别效果好于 SaTC(后续实验只与 SaTC+进行对比)，但是仍低于 BootStomp 和 CPYFinder，而在实际分析中，为避免遗漏关键函数，应该在保证召回率的情况下提高精准率.分析发现，误报的函数来源于编译器的静态编译会引入其他的函数，例如 *__do_global_dtors_aux*，*__getdents* 函数，由于静态分析数据流的准确性有限，导致误报的产生.此外，CPYFinder 在 gcc-4.7.4-0 上的识别效果最好，精准率到达了 0.81，召回率为 1.0.

从表 5 中可以看出编译器版本对拷贝类函数识别存在一定的影响.总的来说，对 C 语言库中拷贝类函数的识别，CPYFinder 优于 BootStomp，SaTC 和 SaTC+.

Table 5　Comparison of Methods for Identifying Memory Copy Functions in C Libraries

表 5　各方法对 C 语言库中内存拷贝函数识别对比

二进制程序 (ARM)	BootStomp		SaTC		SaTC+		CPYFinder	
	R	*P*	*R*	*P*	*R*	*P*	*R*	*P*
gcc-4.5.4-0	0.50	1.0	0.07	0.09	0.36	0.28	1.0	0.56
gcc-4.6.4-0	0.50	1.0	0.14	0.20	0.36	0.28	1.0	0.56
gcc-4.7.4-0	0.43	1.0	0.00	0.00	0.29	0.44	1.0	0.81
gcc-4.8.5-0	0.43	1.0	0.00	0.00	0.36	0.50	0.85	0.79

注：*R* 为召回率；*P* 为精准率.

5.2.2　用户自定义的拷贝类函数的识别效果

为了测试用户自定义实现的拷贝类函数的识别效果，正如 5.1 节中介绍，本文从开源库中收集了相关的用户自定义实现的拷贝类函数，使用不同的编译器编译成 ARM 程序进行测试，测试结果如表 6 所示.从表 6 中可以看出，BootStomp 虽然对库函数中拷贝类函数识别效果好于 SaTC+，但是对自定义实现的拷贝类函数识别效果较差，虽然其准确率几乎为 1，但是召回率几乎为 0.从表 6 中准确率和召回率得出结论：CPYFinder 对自定义拷贝类函数识别的效果好于 BootStomp 和 SaTC+，并且随着编译器版本的升级，识别效果越好.

Table 6　Comparison of Methods on the Identification of Custom Memory Copy Functions

表 6　各方法对自定义内存拷贝类函数识别对比

二进制程序（ARM）	BootStomp		SaTC+		CPYFinder	
	R	*P*	*R*	*P*	*R*	*P*
gcc-4.5.4-O0	0.00	0.00	0.29	0.64	0.45	0.62
gcc-4.6.4-O0	0.00	0.00	0.29	0.64	0.45	0.62
gcc-4.7.4-O0	0.00	0.00	0.29	0.64	0.45	0.62
gcc-4.8.5-O0	0.02	1.0	0.29	0.64	0.68	0.81
gcc-5.4.0-O0	0.02	1.0	0.29	0.61	0.68	0.81
gcc-5.5.0-O0	0.02	1.0	0.29	0.61	0.68	0.81
gcc-6.5.0-O0	0.02	1.0	0.29	0.61	0.68	0.81
gcc-7.5.0-O0	0.02	1.0	0.29	0.61	0.70	0.87

注：*R* 为召回率；*P* 为精准率.

5.2.3　不同指令集及优化等级下的识别效果

为了展示 CPYFinder 对不同架构的拷贝类函数识别效果以及受优化等级的影响，本文将源码使用 GCC 编译器版本 5.4.0 编译成 4 种架构的程序(x86, ARM, MIPS, PPC)以及不同的优化等级(O0~O3).由于 BootStomp 和 SaTC 不完全支持上述指令集架构，因此不再测试 BootStomp 和 SaTC 受指令集架构的影响，首先测试 CPYFinder 受指令集架构和优化等级的影响，然后在 ARM 架构的程序上测试 BootStomp, SaTC 和 CPYFinder 受优化等级的影响.从表 7 中可以看出，CPYFinder 支持 x86, ARM, MIPS, PPC 指令集架构的二进制程序，并且识别的精准率和召回率几乎相同.此外，CPYFinder 会受编译优化等级的影响，在无优化(O0 等级)下识别的效果最好，随着编译优化等级的提升，召回率和精准率均会略微下降.从表 8 中可以看出，除了 BootStomp 在 O1 优化下表现较好，SaTC 和 CPYFinder 均是在 O0 下表现较好.

5.2.4　不同编译器下拷贝类函数的识别效果

为了展示不同方法受编译器种类以及优化等级的影响，本文将使用不同源码版本的 GCC 和 Clang 编译器以 O1 优化等级将源码编译成 ARM 架构的二进制程序，各个方法识别的效果如表 9 所示.

从识别的结果来看，BootStomp 识别的精准率为 1，但是召回率却极低；而 SaTC+识别的精准率和召回率均低于 CPYFinder.此外，3 种方法识别的效果均显示对 Clang 编译的程序识别效果更好.因此，从表 9 数

Table 7 Effect of Different Instruction Sets and Optimization Levels on the Identification

表 7 不同指令集及优化等级对识别的影响

二进制程序	x86		ARM		MIPS		PPC	
	R	*P*	*R*	*P*	*R*	*P*	*R*	*P*
gcc-O0	0.62	0.76	0.81	0.68	0.81	0.65	0.87	0.63
gcc-O1	0.78	0.64	0.78	0.64	0.78	0.64	0.78	0.61
gcc-O2	0.71	0.62	0.64	0.60	0.71	0.62	0.71	0.58
gcc-O3	0.71	0.66	0.64	0.64	0.71	0.66	0.71	0.62

注：*R* 为召回率；*P* 为精准率.

Table 8 Effect of Optimization Level on the Identification of Each Method

表 8 优化等级对各个方法识别的影响

二进制程序（ARM）	BootStomp		SaTC+		CPYFinder	
	R	*P*	*R*	*P*	*R*	*P*
gcc-O0	0.03	1.0	0.30	0.61	0.81	0.68
gcc-O1	0.22	1.0	0.19	0.44	0.78	0.64
gcc-O2	0.22	0.89	0.08	0.60	0.64	0.60
gcc-O3	0.19	1.0	0.08	0.75	0.64	0.64

注：*R* 为召回率；*P* 为精准率.

Table 9 Comparison of the Identification Effect of Each Method Under Different Compilers

表 9 不同编译器下各方法的识别效果对比

编译器	二进制程序（ARM）	BootStomp		SaTC+		CPYFinder	
		R	*P*	*R*	*P*	*R*	*P*
GCC	gcc-4.5.4-O1	0.13	1.0	0.24	0.56	0.78	0.64
	gcc-4.6.4-O1	0.16	1.0	0.18	0.43	0.78	0.64
	gcc-4.7.4-O1	0.18	1.0	0.16	0.42	0.78	0.64
	gcc-4.8.5-O1	0.18	1.0	0.16	0.42	0.76	0.62
Clang	clang-3.9.1-O1	0.29	1.0	0.27	0.58	0.81	0.72
	clang-4.0.1-O1	0.29	1.0	0.27	0.58	0.81	0.72
	clang-5.0.1-O1	0.29	1.0	0.27	0.58	0.87	0.73
	clang-6.0.1-O1	0.29	1.0	0.27	0.58	0.87	0.73

注：*R* 为召回率；*P* 为精准率.

据可得出结论：CPYFinder 识别效果好于 BootStomp 和 SaTC+，并且对 Clang 编译的程序的识别效果优于由 GCC 编译生成的程序.

此外，本文还测试了 Clang 编译下，不同版本(3.9.1, 6.0.1)以及不同优化等级(O0~O4)对拷贝类函数识别效果的影响，对比结果如表 10 所示.从表 10 中可以看出，BootStomp 和 CPYFinder 受 Clang 优化等级的影响均较少，并且在版本较高(6.0.1 版本)的编译器下，识别效果更好；而 SaTC+受编译优化等级的影响较大，随着编译优化等级的升高，其识别效果

Table 10 Effect of Clang Compiler on the Identification of Memory Copy Function

表 10 Clang 编译器对内存拷贝类函数识别的影响

二进制程序（ARM）	BootStomp		SaTC+		CPYFinder	
	R	*P*	*R*	*P*	*R*	*P*
clang-3.9.1-O0	0.0	0.0	0.16	0.66	0.75	0.63
clang-3.9.1-O1	0.29	1.0	0.27	0.58	0.81	0.72
clang-3.9.1-O2	0.24	1.0	0.0	0.0	0.78	0.68
clang-3.9.1-O3	0.24	1.0	0.02	0.50	0.78	0.68
clang-3.9.1-O4	0.24	1.0	0.02	0.50	0.78	0.68
clang-6.0.1-O0	0.0	0.0	0.16	0.66	0.81	0.65
clang-6.0.1-O1	0.29	1.0	0.27	0.58	0.87	0.73
clang-6.0.1-O2	0.24	1.0	0.02	0.50	0.78	0.68
clang-6.0.1-O3	0.24	1.0	0.0	0.0	0.78	0.68
clang-6.0.1-O4	0.24	1.0	0.0	0.0	0.78	0.68

注：*R* 为召回率；*P* 为精准率.

逐渐下降.此外，BootStomp 和 SaTC+存在精准率 *P* 和召回率 *R* 均为 0 的情况.在 Clang 编译的 ARM 程序下，CPYFinder识别的效果好于 BootStomp 和 SaTC+的识别效果.

5.2.5 固件中已知漏洞函数的检测

为了测试 CPYFinder 在实际固件中拷贝类函数识别的效果，本文收集了近 5 年来公开分析的路由器设备固件中的漏洞，其中多数与函数 *strcpy* 相关，由于函数 *strcpy* 作为导入函数调用，无法对其进行识别，不符合实验的要求，因此，最终筛选出由循环拷贝导致的溢出漏洞，共获得了 5 个 CVE，如表 11 所示.这 5 个漏洞分别发现于 TP-Link 和 D-Link 的路由器设备固件中，总共 4 个固件(其中 CVE-2018-3950 和 CVE-2018-3951 由同一个程序中的不同函数导致)，这 4 个固件均是 MIPS 指令集架构的二进制程序，并且均是在由循环实现的内存拷贝中由于对长度未进行校验导致的栈溢出漏洞，并且循环路径中包含多个基本块.对这 5 个导致栈溢出漏洞的函数进行识别

Table 11 Identification Results for Overflow Vulnerabilities Caused by Loop Copy

表 11 对由循环拷贝导致的溢出漏洞的识别结果

溢出漏洞	厂商	检测结果		
		BootStomp	SaTC+	CPYFinder
CVE-2017-13772	TP-Link	*	×	√
CVE-2018-3950	TP-Link	*	*	√
CVE-2018-3951	TP-Link	*	*	√
CVE-2018-11013	D-Link	*	×	×
CVE-2020-8423	TP-Link	*	×	√

注：* 表示不支持或者运行崩溃；×表示未识别出；√表示识别出.

测试，判断 BootStomp, SaTC+, CPYFinder 是否能够识别出这 5 个函数为拷贝类函数，即是否存在内存拷贝的代码片段.

测试结果如表 11 所示，BootStomp 由于不支持 MIPS 架构的二进制程序的识别，给出的结果全为*，SaTC+未识别出这 5 个函数，其中在包含 CVE-2018-3950 和 CVE-2018-3951 的程序中运行时直接崩溃，未给出结果. 而 CPYFinder 识别出 4 个溢出漏洞，其中导致 CVE-2018-11013 的漏洞函数未检测出，导致 CVE-2018-11013 的函数未检测出的原因如图 10 所示.经过分析发现，由于该循环内首先对数据进行存储（基本块 loc_41EB04 中第 1 条指令 sb v1,0x40(v1, 0x40(v0)），然后进行数据的加载（基本块 loc_41EB04 中第 3 条指令 lbu v1,0(v1,0(a0)），这与 4.2 节设定的数据流的特征 3 冲突，导致对函数 *websRedirect* 识别为非拷贝类函数，在关闭特征 3 的限制后，CPYFinder 能够识别出该 CVE，但是随之 CPYFinder 的识别精准率会下降，误报率会增加，在实际的固件程序分析中，可以根据需要来决定是否开启特征 3 的限制.

综上所述，基于控制流和数据流的 CPYFinder 在实际的漏洞函数发现上，效果远远好于基于特征匹配的 BootStomp 和 SaTC+.

5.2.6 效率分析

本节对 BootStomp, SaTC, SaTC+, CPYFinder 在拷贝类函数识别中的效率进行分析，选取的程序为 5.2.1 节中测试的程序，各个方法时耗的对比结果如表 12 所示，CPYFinder 的效率远远高于 BootStomp 和 SaTC. 由于不再对函数参数个数进行分析，SaTC+的时耗低于 SaTC.因此，CPYFinder 在较高的精准率 P 和较高的召回率 R 情况下仍具有较低的时耗.其中 CPYFinder 只用了相当于 BootStomp 19% 的时间，与 SaTC 和 SaTC+的耗时相当.尽管 CPYFinder 对数据流进行了分析，然而由于 SaTC 是基于 angr 开发的，angr

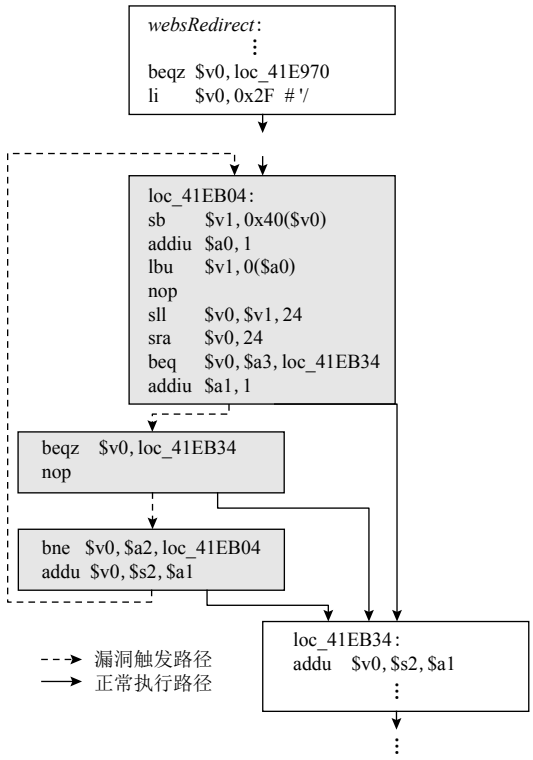


Fig. 10 Vulnerable functions not identified by CPYFinder

图 10 CPYFinder 未识别出的漏洞函数

Table 12 Comparison of Time Consumption for Memory Copy Functions Identification

表 12 对内存拷贝类函数识别时耗对比

二进制程序	时耗/s			
	BootStomp	SaTC	SaTC+	CPYFinder
gcc-4.5.4-O0	12.54	4.16	3.95	3.8
gcc-4.6.4-O0	14.47	4.26	4.57	3.60
gcc-4.7.4-O0	9.75	1.59	0.84	2.56
gcc-4.8.5-O0	9.68	0.99	0.86	2.56
gcc-5.5.0-O0	71.84	15.12	13.89	10.47
总时耗/s	118.28	26.12	24.11	22.99
时耗比值/%	19	88	95	100

注：时耗比值为 CPYFinder 的时耗除以其他方法的时耗.

在分析二进制程序时的效率较低，因此 CPYFinder 借助于 IDA Pro 尽管增加了数据流分析但整个耗时却几乎未增加，而 BootStomp 借助了 IDA Pro 的反编译引擎，由于反编译分析耗时较久，因此 BootStomp 的耗时较高.

6 CPYFinder 与 IDAPro 集成

为方便对工具的使用，本文将 CPYFinder 以 IDA 插件的形式与 IDA Pro 进行了集成，并实现可视化的

结果输出;支持对单个函数的识别和整个二进制程序中所有函数的识别,将识别分为 single 和 all 这 2 种识别模式,其中 single 模式识别当前光标指向的地址所在的函数是否为拷贝类函数,如图 11 所示.single 模式的输出如图 12 所示.对整个二进制程序的识别结果可视化界面如图 13 所示,该界面共分为 5 个部分,其中 Line 为检测时的序号、Local Address 展示函数的地址、Local Name 展示函数名、Loop Address 展示发现的拷贝函数的循环地址入口以及 Is Copy Function 展示该函数是否为拷贝类函数(是为 1,不是为 0).借助于 IDA Pro 的跳转功能,能够方便后续手工分析对结果的确认.

Fig. 11 The interface that CPYFinder provides to the user for mode selection.

图 11 CPYFinder 提供给用户模式选择的界面

```
Python>
[+] is 'strcpy' a copy function?
[+] True (1).
[+] is 'random' a copy function?
[-] False (0).
Python
```

Fig. 12 Output of CPYFinder in single function mode

图 12 CPYFinder 的单函数模式 single 下的输出

Line	Local Address	Local Name	Loop Address	Is Copy Function
019	0x0000a10c	strcpy	0x0000a110	1
022	0x0000a178	strxfrm	0x0000a1a0	1
025	0x0000a264	strcat	0x0000a280	1
027	0x0000a2fc	strncpy	0x0000a31c	1
029	0x0000a368	wcsncat	0x0000a398	1
030	0x0000a3bc	wcsxfrm	0x0000a3e4	1
032	0x0000a46c	wcscat	0x0000a488	1
033	0x0000a49c	wmemcpy	0x0000a4b0	1
038	0x0000a578	wcscpy	0x0000a57c	1
039	0x0000a590	wmemmove	0x0000a5ac	1
040	0x0000a5d0	wcsncpy	0x0000a5f0	1

Fig. 13 Visualization results of CPYFinder's all-mode in IDA Pro

图 13 CPYFinder 的 all 模式在 IDA Pro 中的可视化结果

7 讨 论

本文提出的拷贝类函数识别技术 CPYFinder 通过将二进制函数转换成中间语言 VEX IR 进行数据流的分析,支持对多指令集架构(x86, ARM, MIPS, PPC)的二进制程序中拷贝类函数的识别,不依赖于符号表等信息,并且受编译器版本、优化等级的影响较小.此外,CPYFinder 不仅能够识别库函数中的内存拷贝类函数,还能识别用户自定义的内存拷贝类函数,具有较高的适用性.虽然基于静态分析的识别方法具有快速、高效的特点,但是由于静态数据流分析很难做到十分精确,并且由于用户自定义实现的拷贝类函数多样以及受编译器和优化等级的影响,无可避免地会存在一定的误报和漏报,所以在识别的准确率上具有一定局限性.在后续的研究中,尝试将动态执行以及动静态分析结合的方式应用到拷贝类函数的识别中,提高对拷贝类函数识别的准确率.

8 结束语

拷贝类函数的识别对内存错误漏洞的检测具有重大价值,能够提升下游分析任务的能力,如污点分析、符号执行等.本文提出了一种基于控制流和数据流分析的拷贝类函数识别技术 CPYFinder,通过将二进制程序转换为中间语言 VEX IR,进行后续的数据流分析,提高了对拷贝类函数识别的精准率和召回率,使得 CPYFinder 具有较高的适用性,并且具有较低的运行耗时,支持多种指令集架构(x86, ARM, MIPS, PPC).实验结果表明,CPYFinder 不仅能够有效地识别出 C 语言库中的拷贝类函数,还能够识别用户自定义实现的拷贝类函数,并且受编译器版本、编译优化等级的影响较小,能够发现路由器固件中由此类函数导致的溢出漏洞,这对内存错误类漏洞的发现和分析具有重要作用.

作者贡献声明:尹小康负责算法和对比实验的设计、算法的实现、初稿撰写和修改;芦斌完成算法和对比实验可行性分析、论文的审阅;蔡瑞杰负责实验结果分析、论文的修改;朱肖雅协助实验数据收集、论文的修改;杨启超负责论文的审阅、修改和完善;刘胜利提出研究问题、负责算法和实验的可行性分析、论文的审阅和修改.

参 考 文 献

[1] The MITRE Corporation. 2021 CWE top 25 most dangerous software weaknesses[EB/OL].[2021-08-20]http://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[2] The MITRE Corporation. CWE-787: Out-of-bounds write[EB/OL].[2021-07-20]. https://cwe.mitre.org/data/definitions/787.html

[3] The MITRE Corporation. CWE-125: Out-of-bounds read[EB/OL].[2021-07-20]. https://cwe.mitre.org/data/definitions/125.html

[4] The MITRE Corporation. CWE-119: Improper restriction of operations within the bounds of a memory buffer[EB/OL].[2021-07-20]. https://cwe.mitre.org/data/definitions/119.html

[5] Wang Yawen, Yao Xinhong, Gong Yunzhan, et al. A method of buffer overflow detection based on static code analysis[J]. Journal of Computer Research and Development, 2012, 49(4): 839−845 (in Chinese)
(王雅文, 姚欣洪, 宫云战, 等. 一种基于代码静态分析的缓冲区溢出检测算法[J]. 计算机研究与发展., 2012, 49(4): 839−845)

[6] Li Zhen, Zou Deqing, Wang Zeli, et al. Survey on static software vulnerability detection for source code[J]. Chinese Journal of Network and Information Security, 2019, 5(1): 1−14 (in Chinese)
(李珍, 邹德清, 王泽丽, 等. 面向源代码的软件漏洞静态检测综述[J]. 网络与信息安全学报, 2019, 5(1): 1−14)

[7] Heelan S, Gianni A. Augmenting vulnerability analysis of binary code[C] //Proc of the 28th Annual Computer Security Applications Conf. New York: ACM, 2012: 199−208

[8] Redini N, Machiry A, Das D, et al. BootStomp: On the security of bootloaders in mobile devices[C] //Proc of the 26th USENIX Security Symp (USENIX Security 17). Berkeley, CA: USENIX Association, 2017: 781−798

[9] Redini N, Machiry A, Wang Ruoyu, et al. Karonte: Detecting insecure multi-binary interactions in embedded firmware[C] //Proc of the 41st IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2020: 1544−1561

[10] Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems[J]. ACM SIGPLAN Notices, 2011, 46(3): 265−278

[11] Cha S K, Avgerinos T, Rebert A, et al. Unleashing mayhem on binary code[C] //Proc of the 33rd IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2012: 380−394

[12] Mouzarani M, Sadeghiyan B, Zolfaghari M. A smart fuzzing method for detecting heap-based buffer overflow in executable codes[C] //Proc of the 21st IEEE Pacific Rim Int Symp on Dependable Computing (PRDC). Piscataway, NJ: IEEE, 2015: 42−49

[13] Cheng Kai, Li Qiang, Wang Lei, et al. DTaint: Detecting the taint-style vulnerability in embedded device firmware[C] //Proc of the 48th Annual IEEE/IFIP Int Conf on Dependable Systems and Networks (DSN). Piscataway, NJ: IEEE, 2018: 430−441

[14] Chen Libo, Wang Yanhao, Cai Quanpu, et al. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems[C] //Proc of the 30th USENIX Security Symp (USENIX Security 21). Berkeley, CA: USENIX Association, 2021: 303−319

[15] The MITRE Corporation. A buffer overflow in the httpd daemon on TP-Link TL-WR841N V10[EB/OL]. [2021-07-20]. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-8423

[16] Hex-ray Corporation. Fast library identification and recognition technology[EB/OL]. [2021-07-30]. https://hex-rays.com/products/ida/tech/flirt/

[17] Radare2 Organization. The Official Radare2 Book[M/OL]. 1st ed. 2018 [2021-07-30]. https://github.com/radareorg/radare2-book

[18] Free Software Foundation. Options that control optimization[EB/OL]. [2021-07-15]. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[19] Liu Jian, Su Purui, Yang Min, et al. Software and cyber security—A survey[J]. Journal of Software, 2018, 29(1): 42−68 (in Chinese)
(刘剑, 苏璞睿, 杨珉, 等. 软件与网络安全研究综述[J]. 软件学报, 2018, 29(1): 42−68)

[20] Chen Qian, Cheng Kai, Zheng Yaowen, et al. Function-level data dependence graph and its application in static vulnerability analysis[J]. Journal of Software, 2020, 31(11): 3421−3435 (in Chinese)
(陈千, 程凯, 郑尧文, 等. 函数级数据依赖图及其在静态脆弱性分析中的应用[J]. 软件学报, 2020, 31(11): 3421−3435)

[21] Wang Lei, He Dongjie, Li Lian, et al. Sparse framework based static taint analysis optimization[J]. Journal of Computer Research and Development, 2019, 56(3): 480−495 (in Chinese)
(王蕾, 何冬杰, 李炼, 等. 基于稀疏框架的静态污点分析优化技术[J]. 计算机研究与发展, 2019, 56(3): 480−495)

[22] Tonder R, Kotheimer J, Goues C. Semantic crash bucketing[C] //Proc of the 33rd ACM/IEEE Int Conf on Automated Software Engineering. New York: ACM, 2018: 612–622

[23] Bao T, Burket J, Woo M, et al. BYTEWEIGHT: Learning to recognize functions in binary code[C] //Proc of the 23rd USENIX Security Symp (USENIX Security 14). Berkeley, CA: USENIX Association, 2014: 845−860

[24] Shin E C R, Song D, Moazzezi R. Recognizing functions in binaries with neural networks[C] //Proc of the 24th USENIX Security Symp (USENIX Security 15). Berkeley, CA: USENIX Association, 2015: 611−626

[25] Cisco. SNMP remote code execution, CVE-2017-6736[EB/OL]. [2021-08-10]. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20170629-snmp

[26] The MITRE Corporation. Multiple stack-based buffer overflows in TP-Link WR940N[EB/OL]. [2021-08-10]. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13772

[27] Rawat S, Mounier L. Finding buffer overflow inducing loops in binary executables[C] //Proc of the 6th IEEE Int Conf on Software Security and Reliability. Piscataway, NJ: IEEE, 2012: 177−186

[28] Luo Peng, Zou Deping, Du Yajuan, et al. Static detection of real-world buffer overflow induced by loop[J]. Computers & Security, 2020, 89: 101616

[29] Song D, Brumley D, Yin Heng, et al. BitBlaze: A new approach to computer security via binary analysis[C] //Proc of the 4th Int Conf on Information Systems Security. Berlin: Springer, 2008: 1−25

[30] Pewny J, Garmany B, Gawlik R, et al. Cross-architecture bug search in binary executables[C] //Proc of the 36th IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2015: 709−724

[31] Shoshitaishvili Y, Wang Ruoyu, Salls C, et al. Python bindings for Valgrind's VEX IR[EB/OL]. [2021-05-23]. https://github.com/angr/pyvex

[32] Shoshitaishvili Y, Wang Ruoyu, Salls C, et al. Sok (state of) the art of war: Offensive techniques in binary analysis[C] //Proc of the 37th IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2016: 138−157

[33] Hu Yikun, Zhang Yuanyuan, Li Juanru, et al. Cross-architecture binary semantics understanding via similar code comparison[C]

//Proc of the 23rd IEEE Int Conf on Software Analysis, Evolution, and Reengineering (SANER). Piscataway, NJ: IEEE, 2016: 57−67

[34] Hu Yikun, Zhang Yuanyuan, Li Juanru, et al. Binmatch: A semantics-based hybrid approach on binary code clone analysis[C] //Proc of the 34th IEEE Int Conf on Software Maintenance and Evolution (ICSME). Piscataway, NJ: IEEE, 2018: 104−114

[35] Hagberg A, Schult A, Swart J. Exploring network structure, dynamics, and function using networkx[C/OL] //Proc of the 7th Python in Science Conf (SciPy2008). 2008 [2021-05-23]. http://conference.scipy.org/proceedings/SciPy2008/paper_2/

[36] Netgear. Netgear GPL code[EB/OL]. [2021-06-15]. https://github.com/jameshilliard/R7000

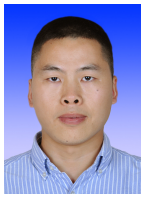
[37] Senpai A. Leaked Mirai source code for research purposes[EB/OL]. [2021-06-15]. https://github.com/jgamblin/Mirai-Source-Code

Yin Xiaokang, born in 1993. PhD. His main research interests include network security, binary code analysis, and machine learning.

尹小康, 1993 年生.博士.主要研究方向为网络安全、二进制代码分析和机器学习.

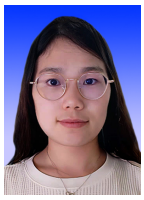
Lu Bin, born in 1982. PhD, associate professor. His main research interests include information security and machine learning.

芦　斌, 1982 年生.博士，副教授.主要研究方向为信息安全和机器学习.



Cai Ruijie, born in 1990. Master, lecturer. His main research interests include network security, binary code analysis, and vulnerability mining.

蔡瑞杰, 1990 年生.硕士，讲师.主要研究方向为网络安全、二进制代码分析和漏洞挖掘.

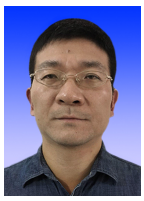


Zhu Xiaoya, born in 1996. Master candidate. Her main research interests include binary code analysis and machine learning.

朱肖雅, 1996 年生.硕士研究生.主要研究方向为二进制代码分析和机器学习.

Yang Qichao, born in 1992. Master, lecturer. His main research interests include network confrontation and network security.

杨启超, 1992 年生.硕士，讲师.主要研究方向为网络对抗和网络安全.



Liu Shengli, born in 1973. PhD, professor, PhD supervisor. His main research interests include network attack detection and network infrastructure security.

刘胜利, 1973 年生.博士，教授，博士生导师.主要研究方向为网络攻击检测和网络基础设施安全.