基于动态执行的基带固件函数安全性分析*

曲海鹏，于 芮，孙 磊，吕文杰
（中国海洋大学信息科学与工程学院，山东 青岛 266100）

摘 要： 移动通信设备中的基带固件运行于独立的计算环境，其漏洞既可危及设备安全，又无法通过设备主操作系统的安全机制进行防护，因此其安全性备受攻防研究者的关注。实验中发现，得到的基带固件由于函数重写、调试信息剥离等原因，现有对比工具难以准确识别其危险函数，进而无法进行后续的漏洞发掘工作。本文提出一种基于动态执行的函数安全性分析方法 DEx。根据预处理过程得到的基带固件的函数信息和段内容，在交叉编译和虚拟机环境下对基带固件函数实现动态执行。基于运行过程中产生的语义特征，设计优先级排序以对函数的安全性进行分析，识别得到内存拷贝函数。基于 DEx 方法构造了 dyndiff 工具，与基于二进制代码相似性方法的主流工具 BinDiff 对比，dyndiff 的危险函数识别率是 BinDiff 的 5.5 倍。最后，阐述了本文工作在后续漏洞挖掘工作中的应用。

关键词： 基带固件；动态执行；语义特征；安全性分析；危险函数
中图法分类号： TP31 **文献标志码：** A **文章编号：** 1672-5174(2022)01-065-06
DOI： 10.16441/j.cnki.hdxb.20210074
引用格式： 曲海鹏，于芮，孙磊，等. 基于动态执行的基带固件函数安全性分析[J]. 中国海洋大学学报(自然科学版)，2022，52(1)：65-70.
Qu Haipeng，Yu Rui，Sun Lei，et al. Security analysis of baseband firmware function based on dynamic execution[J]. Periodical of Ocean University of China，2022，52(1)：65-70.

基带处理器是移动通信设备的重要组成部分，负责处理与蜂窝网络的通信，拥有独立的 RAM 和固件。基带固件使用 C 编写，不使用 libc、glibc 等通用的函数库，使用静态编译且剥离了调试信息，运行于实时操作系统(RTOS)和 ARM 处理器上，包含有与蜂窝协议相关的众多功能[1]。虽然基带固件的这些特点给逆向分析和漏洞发掘带来很大困难，但基带固件漏洞近年来还是不断被曝出[2-4]。由于基带固件的闭源性和大量使用内存拷贝操作，我们认为更有针对性的安全分析将可能发现其更多安全漏洞。目前，研发基带处理器的厂商主要包括，高通，三星，华为等。其中，本文研究的是三星 Shannon 基带芯片，其处理器架构是 Armv7-R，提取得到的基带固件大小有 36.5 MB，经 IDA 识别得到的函数达 61 535 个。

可执行目标文件中的调试信息中包含有函数名称[5]，因此针对调试信息剥离的基带固件的安全分析分为函数安全性分析和漏洞发掘两个阶段。函数安全性分析的目标是重新识别基带固件中执行可变长度内存拷贝功能的危险函数，漏洞发掘的目标是发现由于危险函数的不正确使用，导致的缓冲区溢出漏洞。第一阶段的分析方法多使用基于二进制代码相似性[6]比对的函数识别方法，以目前应用最广泛的 Google BinDiff[7-8]为例，通过将目标固件与使用标准函数库编译得到的固件进行比对，可匹配出与标准函数库中危险函数类似的函数。BinDiff 通过对比多种语义特征，匹配输入文件之间相似的函数，但因基带固件函数重写、调试信息剥离、静态链接等方式导致只能发现少量危险函数。

2012 年 RP Weinmann[3]首次系统性的介绍了在基带固件中发现的内存损坏漏洞，并搭建伪基站对漏洞实现了利用。在 RP Weinmann 的工作中，首先使用二进制文件对比工具 BinDiff，对固件中使用的内存拷贝危险函数进行了重新识别，之后，通过分析危险函数处的代码实现和相关协议规范，发现了多个内存破坏缺陷。在之后的国际黑客会议上，针对基带发现的漏洞多是由于固件中的危险函数导致的缓冲区溢出漏洞，例如函数 memcpy()[9]和 memcpy_s()[2]。

现在已有多种对危险函数进行重写识别的二进制文件对比工具。BinDiff 是目前最流行，应用最广泛的对比工具。BinDiff 依赖于 IDA，并将 IDA 数据库文件作为输入文件，它忽略汇编层指令的结构，工作在函数的控制流程图和调用关系图层面上，基于图形结构中

* 基金项目：国家自然科学基金项目(61827810)资助
Supported by the National Natural Science Foundation of China(61827810)
收稿日期：2021-02-22；修订日期：2021-03-29
作者简介：曲海鹏(1972—)，男，博士，副教授。E-mail：quhaipeng@ouc.edu.cn

所包含的语义特征进行相似性检测，具有很好的抗混淆[10-11]特性。BinDiff 根据两种图形结构中基本块的数量，基本块间边的数量和子函数调用关系，为每个函数计算得到一个“指纹”，并利用该“指纹”在两个二进制文件中匹配相似的函数。此外，对于一些仅使用以上两种图形结构，无法识别相似性的函数，BinDiff 通过比较函数内容的哈希值、函数内循环的次数和函数内引用的字符串等进行更深一步的探索。另外和 BinDiff 一样，工作在抽象图形模型层面上的二进制对比工具还有 Diaphora[12-13]，Turbodiff[14]，BinKit[15]。近几年，随着机器学习方法的流行，跨学科知识的交叉开始在基于语义特征的相似性检测中得到应用，包括 Asm2vec 模型[16]，SAFE 网络[17]等。

另一种识别危险函数的二进制文件对比工具 IDA FLIRT[18]，则基于函数的二进制字节序列进行比对。FLIRT 创建了一个包含所有想要识别的库函数的数据库，每个库函数在 FLIRT 数据库中被表示为一个签名，这个签名是由函数的前 32 个字节组成的，其中包括所有的变体字节。通过将待识别的函数的字节签名与数据库中的签名进行对比，FLIRT 判断其是否为已知的库函数。但是，与使用抽象图形模型的 BinDiff 对比，FLIRT 灵活性较差。

其他的开源的二进制文件对比工具还包括主要应用于微软的补丁分析的 eEye Binary Diffing Suite[19]；适用于较小的文件比较的 IDACompare[20]等。

本文提出了一种基于动态执行的基带固件函数安全性分析方法 DEx。DEx 方法通过把固件正确映射到内存中，对固件函数实现调用执行。同时，针对危险函数的特征，DEx 捕获函数动态执行过程中产生的三种语义信息，包括函数对内存的读写，存储在寄存器 R0 中的返回值以及函数执行过程中产生的输出内容，并设置优先级排序，以用于后续函数的安全性分析。

动态执行方法 DEx 与传统的二进制代码相似性方法存在几个重要的不同之处。首先，DEx 方法只对基带固件进行分析。与二进制代码相似性方法还需使用到对比文件相比，只需要用到目标固件即可。其次，DEx 更加关注危险函数所包含的语义特征。与二进制代码相似性方法所使用的多种的、广泛的特征信息相比，更具针对性。同时，本文基于 DEx 方法，实现了 dyndiff 工具，评估结果显示，dyndiff 提高了基带固件中危险函数的识别准确率。最后，阐述了本文所做的函数安全性分析工作在后续漏洞发掘工作中的应用。

1 方法和实现

1.1 问题分析

为了降低文件的大小和更好的抵抗逆向分析，基带固件中包括符号表在内的调试信息都是被剥离的。针对调试信息剥离的基带固件的逆向分析首先需要使用二进制补丁对比工具重新识别基带固件中执行内存拷贝的危险函数。因此，通过使用目前应用最广泛的 BinDiff 将固件与使用标准函数库编译得到的固件进行比对，重新识别了目标固件中使用的 strcpy()，strncpy()，strcat()，strncat()等危险函数。但是，在人工分析基带伪代码的过程中，我们发现了多个 BinDiff 工具未能正确识别的危险函数，如图 1(a)所示。与图 1(b)中所示的标准库中危险函数相比，其未进行源指针和目的指针所指向的地址是否为空(图 1(b)第二行代码)或是否重叠(图 1(b)第七行代码)等情况的判断，因此，我们推测该函数是在程序开发过程中由编程人员重新实现的。基带固件的上述特征，特别是函数重写导致的危险函数与标准库中危险函数相比实现太过简单，导致了基于多特征信息进行函数匹配的 BinDiff 只能发现少量危险函数。由于基带固件中函数个数达 6 万多个，人工分析以发现全部危险函数显然是不现实的。针对上述问题，本文提出了基于动态执行的 DEx 方法。

```
int __fastcall memcpy_0(int result, int a2, unsigned int a3)
{
  unsigned int i; // r3@1

  for ( i = 0; i < a3; i = (unsigned __int16)(i + 1) )
    *(_BYTE *)(result + i) = *(_BYTE *)(a2 + i);
  return result;
}
```

(a) 固件中危险函数
A dangerous function in firmware

```
1  void* memcpy(void* dst, void* src, size_t size) {
2      if (dst == NULL || src == NULL)
3          return NULL;
4      char* new_dst = (char*)dst;
5      char* new_src = (char*)src;
6
7      if (dst > src && new_dst - new_src <= size) {
8          for (int i = size - 1; i >= 0; i--) {
9              new_dst[i] = new_src[i];
10         }
11     } else {
12         for (int i = 0; i < size; i++) {
13             new_dst[i] = new_src[i];
14         }
15     }
16     return dst;
17 }
18
```

(b) 标准库中危险函数
A dangerous function in standard library

图 1 固件中与标准库中危险函数代码对比

Fig.1 Comparison of dangerous function codes in firmware and standard library

1.2 基于动态执行的 DEx 方法

DEx 方法整体流程框架如图 2 所示。首先，经预处理模块得到基带固件的函数信息和段内容。之后，特征信息收集模块加载基带固件段内容到内存中，根据函数信息动态执行内存中符合条件的函数，并得到函数的语义特征信息。最后，根据函数的语义特征信息，经函数安全性分析模块识别得到危险函数。

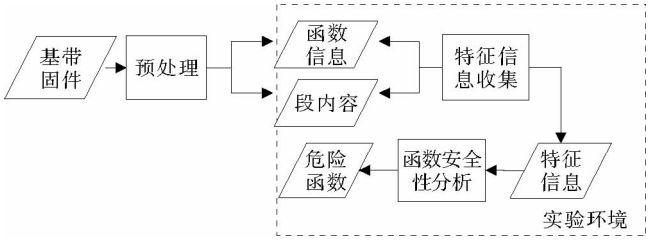

图 2　DEx 方法整体流程

Fig.2　The process of DEx

1.2.1 预处理　　以三星 Shannon 基带固件为例，代码中包含多达 6 万余个函数，为了提高后续动态执行的整体效率，首先经预处理模块过滤掉一部分无关函数。为了确定应该监控并记录基带固件中哪些函数的动态执行时信息，通过对标准函数库中危险函数的分析，DEx 方法将执行内存拷贝的危险函数 mem_copy 分为以下 3 种类型：

re_val mem_copy(void* dest, [dest_length], void* src, [src_length])；

(1)该函数接收两个指针型参数 src 和 dest，并将源指针 src 指向内存中的数据复制或追加到目的指针 dest 指向的内存中。

(2)该函数接收两个指针型参数 src、dest 和一个数值型参数 src_length，并将源指针 src 指向内存中的 src_length 个字节的数据复制或追加到目的指针 dest 指向的内存中。

(3)该函数接收两个指针型参数 src、dest 和两个数值型参数 src_length、dest_length，其中 dest_length 表示目的地址存储空间的大小。

基于以上对危险函数的分类，预处理模块根据 python 模块 idaapi，idc 和 idautils 中提供的函数得到基带固件的函数信息和段内容。其中函数信息包括函数的参数个数，函数距离基带固件文件头的偏移量，以及对函数指令集的说明，用常数 0 和 1 来区分。0 表示 ARM 指令集，1 表示 Thumb 指令集。基带固件的段内容包括每段所包含的字节数量和具体的字节序列内容。

1.2.2 特征信息收集　　根据预处理模块得到的基带固件函数信息和段内容，特征信息收集模块主要完成以下两个操作：

(1)加载基带固件段内存到内存中。

(2)根据基带固件的函数信息对内存中的函数实现动态执行，监视并记录函数运行期间产生的语义特征信息。

一旦段内容被正确映射到内存，DEx 将控制转移到内存中的函数以对其实现动态执行。在此之前，根据函数信息，对待执行的基带固件函数做如下调用约定：仅执行基带固件中参数个数为两个、三个或四个的函数。

(1)如果该函数的参数个数为两个，约定全部为指针类型。

(2)如果该函数的参数个数为三个，约定两个为指针类型，一个为数字类型。

(3)如果该函数的参数个数为四个，约定两个为指针类型，两个为数字类型。

其中，段内容在内存中的基地址与函数信息中函数的偏移量之和即为函数在内存中的地址，通常，通过把该值传值到调用程序处便可实现对函数的执行。但是，ARMv7-R 支持 ARM 和 Thumb 两种指令集。ARM 指令集的指令长度为 32 位，Thumb 指令集的指令长度为 16 位或者 32 位。处于 Thumb 状态(即正在执行 Thumb 指令)的处理器可以通过执行 BX、BLX 等指令进入到 ARM 状态，更改为执行 ARM 指令，反之亦然。当处理器在 Thumb 状态和 ARM 状态之间切换时，传值到调用程序处的值较正常情况略有差异，除了对待执行的固件函数内存地址的说明，还需说明函数使用的是何种指令集。例如分支指令 BLX(register)，如果处理器是由 ARM 状态切换至 Thumb 状态，传值到调用程序处的值等于段内容在内存中的基地址加上函数的偏移量再加上 1，其中“1”表示当前待执行的基带固件函数使用的是 Thumb 指令集。

在实际运行过程中，为保证对基带固件的调用执行总在合理的时间内终止，对于一个函数的一次执行，我们定义了以下标准用于判断函数是否已经运行结束：

(1)执行到达函数的结尾，函数正常返回。

(2)函数执行出错。

(3)函数的运行时间超过约定的时间。

为了确定某个函数是否是执行可变长度内存拷贝功能的危险函数，特征信息收集模块记录基带固件函数的各种动态运行时信息。在分析了标准库中相关危险函数的实现后，本文选取的特征信息即可捕获各种系统级别信息(例如内存访问)，也可获取函数本身可能的输出。同时，对选取的特征信息设置了如下优先级排序以用于后续的函数安全性分析：

(1)第一类优先级是函数对内存的读写。

(2)第二类优先级是存储在寄存器 R0 中函数的返回值。

(3)第三类优先级是函数运行期间产生的输出内容。

执行内存拷贝功能的危险函数,在运行过程中会对参数指针指向的内存进行读写操作。例如,标准函数库中危险函数

char* strcpy(char* dest,const char* src)。

将指针 src 指向内存中的字符串复制到指针 dest 指向的内存中。由于 strcpy()函数没有长度参数,无法确定目的缓冲区的大小,不正确的使用可能会导致缓冲区溢出并破坏相邻其他内存。同时,strcpy()函数将指向目标地址的指针作为返回值传递给上层调用者。因此,特征集收集模块捕获基带固件函数执行前后参数指针指向内存中的内容,以及上层调用者得到的返回值。危险函数的返回值既可以是指向目的地址指针的副本,也可以是常数。例如函数 memcpy_s(),执行成功返回零,错误返回非零值。

1.2.3 *函数安全性分析* 根据特征信息收集模块得到的函数特征信息以及设置的优先级排序,如果基带固件函数访问了参数指针指向的内存,并且函数执行完成后目标地址字符串等于或者包含源地址字符串,则认为该函数是执行内存拷贝功能的危险函数。同时,实验过程中,我们发现基带固件中存在某类函数 A,A 函数只是对危险函数进行了简单调用。安全性分析模块依据第二类优先级,存储在寄存器 R0 中函数的返回值,对 A 类函数实现过滤。最后,第三类优先级,函数执行过程中产生的所有输出内容被发送到指定文件并提供给研究人员,以发现其他有趣的部分。

2 实验与结果分析

2.1 实验结果

基于 DEx 方法,我们实现了 dyndiff 函数安全性分析工具(https://github.com/xibeiandchangan/dyndiff),并以三星 Shannon 基带固件为对象进行了对比实验。实验环境为 ubuntu linux 18.04.2 LTS 系统,安装交叉编译工具链 arm-linux-gnueabihf[21] 和 QEMU 虚拟机管理器[22],其中 arm-linux-gnueabihf 支持 x86 处理器架构上进行 ARM 处理器架构可执行代码的交叉编译生成,QEMU 用户模式支持在 x86 处理器架构上进行 ARM 可执行文件的模拟执行。实验结果如表 1 所示。

2.2 与 BinDiff 性能对比

dyndiff 重新识别得到固件中 22 个危险函数,而 BinDiff 识别结果为 4 个。其中,图 3 为本文实现工具 dyndiff 和 BinDiff 识别结果对比图。dyndiff 比 BinDiff 表现更好:dyndiff 识别得到的危险函数的个数是 BinDiff 对比工具的 5.5 倍,其中包括对 BinDiff 识别结果的全覆盖。

表 1 dyndiff 识别结果

Table 1 Recognition result of dyndiff

识别得到的危险函数的地址 The address of the identified dangerous function	参数个数 Number of parameters
0xf94fb0, 0xa9fcb4, 0x9d3248	2
0x2435b9e, 0x2435b64, 0x166edc0, 0x166edbc, 0x1642118, 0x15bdb58, 0x13d2268, 0xf21b72, 0xe152d6, 0xde2dda, 0xcde34e, 0xa8ae84, 0x94da8a, 0x89e80c, 0x75c5c6, 0x71e4f8, 0x6c63d2,	3
0xf2a3b0, 0x9d42c6	4

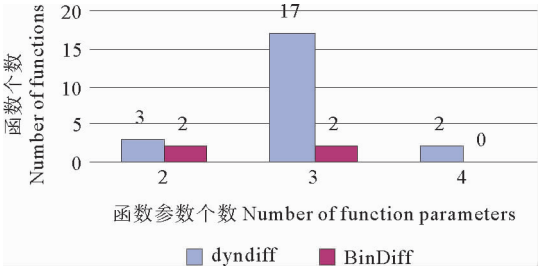

图 3 dyndiff 和 BinDiff 识别结果对比图

Fig.3 Comparison of recognition results between dyndiff and BinDiff

2.3 结果分析

在对 dyndiff 工具的识别结果分析过程中,本文发现一个有趣的现象,即在基带固件中存在多个代码实现完全相同的危险函数,如图 4 所示。我们推测这可能是由于基带固件源程序是由多个程序开发人员分工合作编写完成,因此出现了对同一功能函数的多次实现。

在 3GPP 规范中,运行在手机端的 GSM 协议栈的第 3 层消息由 IEs(Information Elements)组成,而某些 IEs 被定义为可变长度的。当危险函数将可变长度的 IEs 复制至本地栈时,不充分的长度检查将导致缓冲区溢出漏洞。因此,本论文进行的函数安全性分析工作,是后续漏洞发掘的基础。基于 dyndiff 识别结果,通过分析危险函数处的代码实现和与之相对应的协议规范,我们发现了存在的内存破坏缺陷,更深入的利用研究仍在进一步进行。

```
015BDB58 sub_15BDB58                                    ; CO
015BDB58                                                ; su
015BDB58                PUSH            {R4,LR}
015BDB5A                MOVS            R3, #0
015BDB5C                B               loc_15BDB66
015BDB5E ; ---------------------------------------------
015BDB5E
015BDB5E loc_15BDB5E                                    ; CO
015BDB5E                LDRB            R4, [R1,R3]
015BDB60                STRB            R4, [R0,R3]
015BDB62                ADDS            R3, R3, #1
015BDB64                UXTH            R3, R3
015BDB66
015BDB66 loc_15BDB66                                    ; CO
015BDB66                CMP             R3, R2
015BDB68                BCC             loc_15BDB5E
015BDB6A                POP             {R4,PC}
015BDB6A ; End of function sub_15BDB58
```

(a) 函数sub_15BDB58l
Function sub_15BDB58

```
00F21B72 sub_F21B72                                     ; COI
00F21B72                                                ; sul
00F21B72                PUSH            {R4,LR}
00F21B74                MOVS            R3, #0
00F21B76                B               loc_F21B80
00F21B78 ; ---------------------------------------------
00F21B78
00F21B78 loc_F21B78                                     ; COI
00F21B78                LDRB            R4, [R1,R3]
00F21B7A                STRB            R4, [R0,R3]
00F21B7C                ADDS            R3, R3, #1
00F21B7E                UXTH            R3, R3
00F21B80
00F21B80 loc_F21B80                                     ; COI
00F21B80                CMP             R3, R2
00F21B82                BCC             loc_F21B78
00F21B84                POP             {R4,PC}
00F21B84 ; End of function sub_F21B72
```

(b) 函数sub_F21B72
Function sub_F21B72

图 4　两个实现完全相同的危险函数

Fig.4　Two dangerous functions that implement exactly the same

3　结语

本文针对基带固件函数的安全性分析问题，提出了动态执行方法 DEx。DEx 通过对基带固件函数实现调用执行，并根据其运行过程中产生的语义信息和设置的优先级排序，解决了基带固件中危险函数的识别问题。实验结果表明，dyndiff 工具表现出比 BinDiff 更好的识别效果。由于基带固件中函数数量达 6 万多个，后续的工作将研究 DEx 方法的预处理模块，对待执行函数做进一步的筛选，以进一步提高 dyndiff 的工作效率。本文实现工具 dyndiff 的下载地址 https://github.com/xibeiandchangan/dyndiff。

参考文献：

[1] Kim E, Kim D, Park C J, et al. BASESPEC: Comparative Analysis of Baseband Software and Cellular Specifications for L3 Protocols[C]. [s.l.]: The Network and Distributed System Security Symposium 2021, 2021.

[2] 腾讯科恩实验室. 腾讯科恩实验室 Black Hat 2018 议题解读 | 现代智能手机基带破解[EB/OL]. https://paper.seebug.org/667/: seebug, 2018.
KeenSecurityLab. KeenSecurityLab Black Hat 2018 Issue interpretation | Exploitation of a Modern Smartphone Baseband[EB/OL]. https://paper.seebug.org/667/: seebug, 2018.

[3] WEINMANN R P. Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks[C]. California: USENIX Association, 2012: 12-21.

[4] Maier D, Seidel L, Park S. BaseSAFE: Baseband sanitized fuzzing through emulation[C]. New York: Association for Computing Machinery, 2020: 122-132.

[5] Patrick-Evans J, Cavallaro L, Kinder J. Probabilistic naming of functions in stripped binaries[C]. New York: Association for Computing Machinery, 2020: 373-385.

[6] Haq I U, Caballero J. A survey of binary code similarity[J]. arXiv preprint arXiv: 1909, 2019: 11424.

[7] Flake H. Structural comparison of executable objects[J]. Dimva, 2004: 161-173.

[8] Zynamics. BinDiff[EB/OL]. https://www.zynamics.com/bindiff.html: Zynamics, 2020.

[9] Daniel Komaromy, Nico Golde. Breaking Band[EB/OL]. https://recon.cx/2016/talks/Breaking-Band.html: RECON, 2016.

[10] Collberg C, Martin S, Myers J, et al. Distributed Application Tamper Detection Via Continuous Software Updates[C]. New York: Association for Computing Machinery, 2012: 319-328.

[11] Ugarte-Pedrero X, Balzarotti D, Santos I, et al. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers[C]. San Jose: 2015 IEEE Symposium on Security and Privacy, 2015: 659-673.

[12] joxeankoret. diaphora. https://github.com/joxeankoret/diaphora: github, 2021.

[13] Karamitas C, Kehagias A. Efficient Features For Function Matching Between Binary Executables[C]. Campobasso: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering, 2018: 335-345.

[14] CORE SECURITY. Turbodiff[EB/OL]. https://www.coresecurity.com/core-labs/open-source-tools/turbodiff-cs: CORE SECURITY, 2021.

[15] ohjeongwook. Binkit[EB/OL]. https://github.com/ohjeongwook/binkit: github, 2020.

[16] Ding S H H, Fung B C M, Charland P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization[C]. San Francisco: 2019 IEEE Symposium on Security and Privacy, 2019: 472-489.

[17] Massarelli L, Di Luna G A, Petroni F, et al. Safe: Self-Attentive Function Embeddings for Binary Similarity[C]. [s.l.]: Detection of Intrusions and Malware, and Vulnerability Assessment, 2019: 309-329.

[18] Hex-Rays. IDA F. L. I. R. T. Technology: In-Depth[EB/OL]. https://www.hex-rays.com/products/ida/tech/flirt/: Hex-Rays, 2020.

[19] Darknet. eEye Binary Diffing Suite (EBDS)[EB/OL]. https://www.darknet.org.uk/2006/08/eeye-binary-diffing-suite-ebds/: Darknet, 2010.
[20] Dzzie. IDACompare[EB/OL]. https: github.com/dzzie/IDACompare: github, 2019.
[21] Linaro. Latest Linux Targeted Binary Toolchain Releases [EB/OL]. https://www.linaro.org/downloads: Linaro, 2019.
[22] QEMU. QEMU the FAST processor emulator[EB/OL]. https://www.qemu.org/: QEMU, 2020.

Security Analysis of Baseband Firmware Function Based on Dynamic Execution

Qu Haipeng, Yu Rui, Sun Lei, Lv Wenjie
(College of Information Science and Engineering, Ocean University of China, Qingdao 266100, China)

Abstract: The baseband firmware in mobile communication devices runs in an independent computing environment, and its vulnerabilities can endanger the security of the device and cannot be protected by the security mechanism of the device's main operating system. Therefore, its security has attracted the attention of attack and defense researchers. In the experiment, it was found that due to function rewriting and debugging information stripping of the obtained baseband firmware, it is difficult for the existing comparison tools to accurately identify its dangerous functions, and subsequent vulnerability discovery work cannot be performed. This paper proposes a function safety analysis method DEx based on dynamic execution. According to the function information and segment content of the baseband firmware obtained in the preprocessing process, the baseband firmware functions are dynamically executed in the cross-compilation and virtual machine environment. Based on the semantic features generated in the running process, the priority ranking is designed to analyze the safety of the function and identify the memory copy function. The tool called dyndiff is constructed based on the DEx method. Compared with the existing mainstream tool BinDiff based on the binary code similarity method, the risk function recognition rate of dyndiff is 5.5 times that of BinDiff. Finally, the application of this work in the follow-up vulnerability mining work is explained.

Key words: baseband firmware; dynamic execution; semantic feature; security analysis; dangerous functions

责任编辑 徐 环

(上接第 55 页)

附录:其他软体动物寄生桡足类的调查情况

作者的研究中,除调查了黄渤海 32 种双壳类外,还对其它 32 种贝类(1 451 个体)进行了调查,但均未检查到寄生桡足类,采样情况如下:多板纲(Polyplacophora):朝鲜鳞带石鳖(*Lepidozona coreanica*(Reeve)),大连、青岛,2 批 2 个;红条毛肤石鳖(*Acanthochitona rubrolineata* (Lischke)),青岛,5 批 106 个;双刻鳞带石鳖(*Lepidozona bisculpta* (Carpenter in Pilsbry)),青岛,1 批 1 个。腹足纲(Gastropoda):半褶织纹螺(*Nassarius sinarum*(Philippi)),连云港,1 批 1 个;背小节贝(*Lottia dorsuosa* (Gould)),青岛、日照,2 批 24 个;扁玉螺(*Neverita didyma* (Röding)),莱州、青岛,2 批 2 个;朝鲜花冠小月螺(*Lunella correensis* (Récluz)),青岛、日照,6 批 98 个;单齿螺(*Monodonta labio*(Linnaeus)),大连、葫芦岛、莱州、青岛、日照,8 批 191 个;单一丽口螺(*Tristichotrochus unicus* (Dunker)),大连,1 批 4 个;短滨螺(*Littorina brevicula* (Philippi)),大连、葫芦岛、莱州、青岛、日照、连云港,9 批 417 个;方斑东风螺(*Babylonia areolata* (Link)),南通,1 批 1 个;古氏滩栖螺(*Batillaria cumingii* (Crosse)),莱州、青岛,2 批 15 个;红带织纹螺(*Nassarius succinctus* (Adams)),日照,1 批 1 个;嫁虫戚(*Cellana toreuma* (Reeve)),青岛,3 批 29 个;杰氏裁判螺(*Funa jeffreysii*(Smith)),青岛,1 批 8 个;口马丽口螺(*Calliostoma koma* (Shikama & Habe)),青岛,1 批 1 个;蓝无壳侧鳃海牛(*Pleurobranchaea maculata* (Quoy & Gaimard)),青岛,1 批 1 个;丽小笔螺(*Mitrella albuginosa* (Reeve)),葫芦岛、青岛、日照,3 批 54 个;脉红螺(*Rapana venosa* (Valenciennes)),莱州、青岛、日照,6 批 30 个;皮氏蛾螺(*Volutharpa perryi* (Jay)),青岛,2 批 25 个;日本菊花螺(*Siphonaria japonica* (Donovan)),青岛,1 批 1 个;润泽角口螺(*Ceratostoma rorifluum* (Adams & Reeve)),大连,1 批 20 个;史氏背尖贝(*Nipponacmea schrenckii*(Lischke)),大连、青岛,3 批 56 个;双带小笔螺(*Mitrella bicincta*(Gould)),青岛,1 批 10 个;托氏琯螺(*Umbonium vestiarium*(Linnaeus)),莱州、青岛,2 批 15 个;习见织纹螺(*Nassarius pyrrhus*(Menke)),莱州,1 批 2 个;秀丽织纹螺(*Reticunassa festiva* (Powys)),青岛、连云港,2 批 9 个;锈凹螺(*Tegula rustica* (Gmelin)),大连、葫芦岛、龙口、威海、青岛、日照,8 批 104 个;疣荔枝螺(*Reishia clavigera* (Küster)),大连、葫芦岛、龙口、青岛、日照,10 批 185 个;纵肋织纹螺(*Nassarius variciferus*(Adams)),莱州,1 批 2 个。头足纲(Cephalopoda):火枪鱿(*Loliolus*(*Nipponololigo*) *beka* (Sasaki)),青岛,2 批 15 个;双喙耳乌贼(*Lusepiola birostrata* (Sasaki)),青岛,2 批 21 个。