基于控制流挖掘的 Android 系统代码漏洞分析

孙可钦　汪孙律　刘剑

（中国科学院软件研究所 基础软件国家工程研究中心，北京 100190）

摘要：Android 操作系统被广泛应用于智能手机、平板电脑等便携移动设备，因此安全性和可靠性至关重要。本文使用控制流挖掘方法，针对 Android 内核代码的多种典型错误构建相关的分析脚本，进行了分析检测，并对 Android 系统多版本间进行分析对比。本文首次将控制流挖掘方法应用于 Android 系统，通过系统化的实验分析包含 Android 扩展的驱动在内的所有 Android 内核代码，发现了代码库中一系列脆弱点。

关键词：控制流挖掘；漏洞分析；Android

中图分类号：TP3　　**文献标识码**：A

Vulnerability Analysis of the Android System Code Based on Control Flow Mining

Sun Keqin　Wang Sunlv　Liu Jian

(National Engineering Research Center of Fundamental Software, Institute of Software Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Android operating system is widely used in smart phones, tablet PCs and other portable mobile devices. Therefore, the security and reliability of Android OS code is very important. In this paper, a systematic checking is applied to Android code based on control flow mining and manual checking scripts of typical kernel errors. An analysis and comparison among multi-version for Android OS codes is also accomplished. To our knowledge, this is the first time to apply control flow mining methods to Android system code, which include many new modules such as the addition drivers. In our experiments, many vulnerabilities are also exposed.

Key words: control flow mining; Vulnerability analysis; Android

0　引　　言

现今，Android 操作系统被广泛应用于智能手机、平板电脑等便携移动设备，因此 Android 操作系统的安全性和可靠性至关重要。近年来工业界和学术界都

作者简介：孙可钦（1988—），男，硕士研究生，主要研究领域：静态检测、漏洞挖掘、程序验证

基金项目：面向访问验证保护级的安全操作系统原型系统研发（KGCX2-YW-125）

E-mail: keqin@nfs.iscas.ac.cn

发起了一系列针对 Android 系统的分析工作。例如宾夕法尼亚大学的 William Enck,杜克大学的 Peter Gilbert,以及 Intel 实验室的 Byung-Gon Chun 等人开发的基于动态污染分析的 TaintDroid 实时监视系统,能够跟踪和精确分析用户个人隐私数据的流向,以避免数据泄露或污染[1]。但是,这些工作都仅仅关注 Android 代码的某一侧面,现有的工作尚未见到对 Android 代码进行较全面、系统的检测分析工作。

软件挖掘是近年来兴起的代码分析和检测的重要方法。Zhenmin Li, Yuanyuan Zhou 提出了 PR-Miner 方法[2],该方法利用频繁集挖掘技术高效地从大规模软件代码中抽取隐含的编程规则。和其他方法相比,本文采用的静态软件挖掘技术不需要编译、执行代码,其优势在于可以应用于局部代码的检测,处理的代码量大,分析问题的种类多,适合于对目标代码进行系统化的分析和检测,且具有更高的路径覆盖率。

基于控制流挖掘的方法就是利用静态分析对局部函数块进行控制流分析,找到潜在的违反 API 规则、非法指针引用等错误。例如一种拒绝服务漏洞是由空指针引用导致的[3]。但是,基于控制流的挖掘方法也具有自身的不足,例如:需要分析人员对目标领域有较深的理解,熟悉分析对象常见的错误、脆弱点模式。同时需要大量的人工工作,包括构建检测脚本、排除误报等。

本文使用控制流挖掘方法,针对 Android 内核代码的多种典型错误构建相关的分析脚本,并对 Android 系统多版本间进行分析对比。本文首次将控制流挖掘方法应用于 Android 系统,通过大量实验系统地分析 Android 系统代码,并对 Android 现有的各个版本进行对比分析,发现了代码库中一系列的脆弱点。

1 研究背景

1.1 Android 内核介绍

Android 内核是在 Linux 内核的基础上修改扩展而来的,因此 Linux 内核的安全问题同样影响 Android。目前通过静态分析找出的 Linux 内核 Bug 主要发生在驱动和硬件适配层[6],即 drivers 和 arch 等文件夹中。这些文件在 Android 内核中依然存在,并且随着 Android 需要兼容越来越多的硬件设备种类,这些错误还会增加。

Android 自 2008 年 9 月发布第一版 Android1.1 以来,又发布了多个版本,表 1 是 Android 主流版本与 Linux 内核版本的关系[4]。

表 1 Android 版本与 Linux 内核关系

Android 版本名	Android 版本	Linux 内核
Cupcake	1.5	2.6.27
Eclair	2.1	2.6.29
Froyo	2.2	2.6.32
Gingerbread	2.3	2.6.35
IceCreamSandwich	4.0	3.0.8

Android 与 Linux 内核的区别主要有以下几个方面：

- 没有 GUN/Linux 的 X Window 系统
- 使用 Bionic Libc 代替 glibc
- 不包括一整套标准的 Linux 使用程序
- 专有的驱动程序

另外，Android 还对 Linux 内核进行了增强，主要有：Goldfish、YAFFS2、蓝牙、调度器、新增驱动、电源管理、硬件时钟、匿名内存共享、低内存管理、日志设备、Android PMEM、switch、Timed GPIO、Android Ram Console[5]。这些改变也会带来安全问题，需要特别关注。

1.2 基于控制流挖掘的代码分析方法

软件挖掘的方法近年来成功应用于很多系统代码，特别是操作系统的分析和检测中，例如哥本哈根大学的 Nicolas Palix 等人对 Linux 系统自 2.6.0 版本至 2.6.33 版本的内核代码进行了全面的分析和比较[6]。Julia Lawall 等人对 OpenSSL 进行了错误分析并检测出 30 多个 Bug[7]。Coccinelle[8] 是一种通用的控制流挖掘方法。针对不同的系统、不同的错误目标需要编写不同的挖掘脚本。挖掘脚本由一种称为 SmPL (semantic patch language)[9] 的语言编写。SmPL 是基于补丁语法的代码转换语言，它将具体的代码细节抽象出来转换成泛化的语义补丁，可以自动化地在大量文件中进行模式匹配。

由于 Linux 操作系统源码已经几乎包含 14000 个 C 文件，代码行数已超八百万，自 2006 年起，代码增长率达百分之五十，因此巨大的文件量和代码量使得人工挖掘并分析代码已不现实，基于控制流挖掘方法可以通过自定义挖掘脚本，从大量代码中挖掘有用信息，从而高效地进行代码分析。

Coccinelle 已成功的应用于 Linux 内核分析，并且多次为 Linux 社区提交代码补丁，但对于 Android 内核源码的分析工作还没有展开。Coccinelle 面向控制流的特点使其不需要编译、执行代码，因此适合大规模代码的挖掘分析。支持对 C 语言代码的挖掘和使用 python 脚本控制结果输出，这些特点非常适合分析如 An-

droid 内核等大规模内核代码。

2 内核常见错误的分析

本节使用 Coccinelle 工具对 Android 内核常见错误进行分析。常见的错误包括 1)持有自旋锁时调用阻塞函数;2)释放后调用;3)内核中使用浮点数;4)从用户数据中得到的值被用于数组下标或循环边界,而没有对其进行检查。这些错误曾多次在其他系统代码中出现并影响系统运行的稳定性。

2.1 持有自旋锁时调用阻塞函数

执行单元如果持有自旋锁,当其调用阻塞函数时,因阻塞函数往往会调用 schedule(),从而可能会有死锁情况发生。要找到这种错误,需要获取阻塞函数的集合和持有自旋锁的函数集合,然后根据过程间分析识别调用情况来确定 Bug 可能发生的位置。持有自旋锁的阻塞函数集合可根据图 1 的语义规则来找出。

```
1 @lock_block exists@
2 position p1, p2
3 expression lock;
4 @@
5 (
6 spin_lock@p1
7)
8 (lock,...)
9... when ! =lock
10 GFP_KERNEL@p2
```

图 1 查找持有自旋锁的阻塞函数的语义规则

这条规则第 1 行声明了规则名称 lock_block,关键字 exists 是路径算子,表示“存在至少一条路径”,即至少一条路径满足规则便可匹配。第 2 行和第 3 行声明了三个变量,位置变量 p1,p2 以及表达式变量 lock,第 6 行匹配 spin_lock 函数,并将其位置与 p1 绑定。第 8 行表示函数的参数,其中第一个参数与表达式 lock 绑定,其余参数不限。第 9 行表示在遇到 GFP_KERNEL 之前没有任何对 lock 的使用,并将 GFP_KERNEL 的位置与 p2 绑定。这就能检测出在持有自旋锁之后调用阻塞函数的实例。图 2 是根据上述规则报告的一个错误,位置是 Android_2.1 的 goldfish 版本中的 drivers/isdn/mISDN/stack.c 文件的第 1 行和 6 行。第 1 行调用了 read_lock 函数,而在第 6 行程序又调用了包含 GFP_KERNEL 参数的阻塞函数。当程序走到该分支时,可能会造成死锁。

```
1 read_lock(&sl->lock);
2 sk_for_each(sk, node, &sl->head) {
3   if (sk->sk_state ! =MISDN_BOUND)
4     continue;
5   if (! cskb)
6     cskb=skb_copy(skb, GFP_KERNEL);
7   if (! cskb) {
8     printk(KERN_WARNING "%s no skb\n", __func__);
9       break;
10  }
11  if (! sock_queue_rcv_skb(sk, cskb))
12    cskb=NULL;
13  }
```

图 2 drivers/isdn/mISDN/stack.c

2.2 释放后调用

在有些函数的实现中会有释放参数的过程,当该函数被调用后,被释放的参数仍可能会被再次引用,这就造成了指针的非法引用错误。这种错误的检测方式是,首先要找到会释放参数的函数,其次检查该函数被调用之后,被释放的参数是否被再次引用。在函数中调用 kfree()是常见的释放参数的方法,因此本实验中的规则用来检测函数中存在使用 kfree()来释放参数的实例。可以通过图 3 中的规则来找出会释放参数的函数。

```
1 @free_p exists@
2 identifier fn, p;
3 type T;
4 @@
5 void fn(…, T p, …){
6  …
7  kfree(p);
8  …
9}
```

图 3 free_p 规则找出会释放参数的函数

规则的第 1 行定义规则名为 free_p 和路径算子 exists。第 2 行定义标识符变量 fn 和 p,用 fn 标识函数,用 p 标识会释放的参数。第 3 行定义类型变量 T,用来标识会释放参数的类型。第 5 行到第 9 行匹配函数实现,该函数的返回类型为

void。第 7 行匹配函数实现中调用 kfree(p),p 即为参数列表中的某一个参数。第 6 行和第 8 行的"…"表示任意代码。当规则执行完毕并且匹配成功后,第 5 行的 fn 绑定函数名称,T,p 绑定参数类型和参数名。

将 free_p 规则应用于全部系统源码,找出所有通过调用 kfree()释放参数的函数,并标记函数名和被释放参数的类型。之后利用模板,为每一个函数生成一系列语义规则。生成的规则检查该函数被调用后,程序是否还会使用被释放的参数,如果使用,就报告一个错误。在 Android_4.0 的 goldfish 版本中,free_p 规则共找到 825 个函数。以 free_msg 函数为例,其释放的参数类型为 struct msg_msg*,模板生成的语义规则见图 4。

```
1 @invoke@
2 struct msg_msg * E;
3 position p;
4 @@
5 free_msg(…, E@p, …)
6 @use exists@
7 struct msg_msg * invoke.E;
8 expression E1;
9 position p ! =invoke.p;
10 position p1;
11@@
12 free_msg@p1(..., E,...);
13 …
14 (
15   E=E1
16 |
17   E@p
18)
```

图 4 找到调用 free_msg()后使用被释放参数

将模板生成的这些规则应用于所有 Android 系统源码,即可找到所有同类错误。比如在 Android_2.1 的 goldfish 版本的 drivers/serial/icom.c 文件代码中,见图 5,icom_free_adapter 函数会释放其参数 icom_adapter,而在之后的代码参数中又引用了该参数,造成非法指针引用。

```
1 icom_free_adapter(icom_adapter);
2 pci_release_regions(icom_adapter->pci_dev);
```

图 5 drivers/serial/icom.c

另一个错误出现在 Android 4.0 的 common 版本的 hardware/ti/wlan/mac80211/ti-utils/uim_rfkill/uim.c 文件代码中，见图 6，该文件属 Android 特有文件。tist_ko_path 在第 1 行被释放，而在 2 行再次被引用。

```
1 free(tist_ko_path);
…
2 asprintf(&tist_ko_path,
      "/lib/modules/%s/kernel/drivers/staging/ti-st/bt-drv.ko",
      Name.release);
```

图 6 hardware/ti/wlan/mac80211/ti-utils/uim_rfkill/uim.c

2.3 其他错误

内核常见的错误还有以下两种：内核中使用浮点数和从用户数据中得到的值被用于数组下标或循环边界，而没有对其进行检查。

在内核代码中，大多数的浮点数经过编译器执行运算转换成为整形，或者并没有被编译到内核中。由于本文中使用的静态软件挖掘技术不实际执行编译过程和计算，因此仅能检测浮点常量的情况。

研究发现，memcpy_fromfs 和 copy_from_user 函数会访问用户数据，当通过这两个函数获得的值被用于数组下标或循环检查边界的时候，如果没有提前对其值进行检查，就可能会出现错误。

由于篇幅有限，相关的挖掘规则不再赘述。

3 实验和结果

该方法针对四种内核错误：1）持有自旋锁时调用阻塞函数，简称自旋锁阻塞；2）释放后调用；3）内核中使用浮点数；4）从用户数据中得到的值被用于数组下标或循环边界，而没有对其进行检查，简称边界未检查。分别在 Android 的 1.5 、2.1、2.2、2.3、4.0 这五个版本的内核代码中进行实验①。

① 注：本文查找出的 Bug 均为疑似 Bug，部分结果已经提交到开源社区，等待官方确认。

表 2 Android1.5 实验结果

Android1.5	Bug	误报	总数
自旋锁阻塞	2	5	7
释放后调用	3	52	55
使用浮点数	2	10	12
边界未检查	2	0	2
总计	9	67	76

表 2 是对 Android_1.5 版本内核进行的错误检测实验,该实验总共报告了 76 个错误,经手工验证后得出有 10 个疑似 Bug,以及 66 个误报。出现误报的主要原因在于该工具不具备数据流分析的能力,从而无法对不可达状态进行裁剪。表 3-表 6 是其他四个版本的实验结果。

表 3 Android2.1 实验结果

Android2.1	Bug	误报	总数
自旋锁阻塞	1	3	4
释放后调用	3	60	63
使用浮点数	2	13	15
边界未检查	2	0	2
总计	8	76	84

表 4 Android2.2 实验结果

Android2.2	Bug	误报	总数
自旋锁阻塞	1	3	4
释放后调用	4	71	75
使用浮点数	3	25	28
边界未检查	2	0	2
总计	10	99	109

表 5 Android2.3 实验结果

Android2.3	Bug	误报	总数
自旋锁阻塞	2	3	5
释放后调用	3	78	81
使用浮点数	4	35	39
边界未检查	2	0	2
总计	11	116	127

表 6 Android4.0 实验结果

Android4.0	Bug	误报	总数
自旋锁阻塞	1	3	4
释放后调用	4	70	74
使用浮点数	4	42	46
边界未检查	1	0	1
总计	10	115	125

4 实验结果分析与总结

4.1 横向结果分析

在 Android_1.5 版本中发现的自旋锁阻塞错误发生在 drivers/net/wan/lmc/lmc_main.c 中(见图 7),第 1 行处申请了一个自旋锁,在之后的一个 switch 分支中调用了 kmalloc(xc.len, GFP_KERNEL),按前文所述在最坏情况下会引起系统死锁。这个错误在随后的 2.1 版本中被修正。修正后的代码在每一个 case 分支内单独申请自旋锁,并且将 kmalloc 函数提到自旋锁之前(见图 8),从而避免错误。

```
1 spin_lock_irqsave(&sc->lmc_lock, flags);
2 switch (cmd){
…
3 data=kmalloc(sc.len, GFP_KERNEL);
…
4 }
5 spin_unlock_irqrestore(&sc->lmc_lock, flags);
```

图 7 Android_1.5 drivers/net/wan/lmc/lmc_main.c

```
1 case lmc_xilinx_load:
…
2 data=kmalloc(sc.len, GFP_KERNEL);
…
3 spin_lock_irqsave(&sc->lmc_lock, flags);
…
4 spin_unlock_irqrestore(&sc->lmc_lock, flags);
```

图 8 Android_2.1 drivers/net/wan/lmc/lmc_main.c

另一个自旋锁阻塞错误发生在 Android_1.5 版本的 drivers\infiniband\hw\nes\nes_verbs.c 文件中(见图 9),第 2 行调用了包含 GFP_KERNEL 标志位的 kzalloc 函数,这个错误也在 Android_2.1 版本中被修复(见图 10),修复方式是将 GFP_KERNEL 标志修改为 GFP_ATOMIC,后者不会因为睡眠而执行调度,也就不会造成死锁。

```
1 spin_lock_irqsave(&nesadapter->pbl_lock, flags);
…
2 kzalloc(…, GFP_KERNEL)
…
3 spin_unlock_irqrestore(&nesadapter->pbl_lock, flags);
```

图 9 Android_1.5 drivers\infiniband\hw\nes\nes_verbs.c

```
1 spin_lock_irqsave(&nesadapter->pbl_lock, flags);
…
2 kzalloc(…, GFP_ATOMIC)
…
3 spin_unlock_irqrestore(&nesadapter->pbl_lock, flags);
```

图 10 Android_2.1 drivers\infiniband\hw\nes\nes_verbs.c

一个 Android_1.5 版本的释放后调用错误出现在 drivers\serial\icom.c 文件中(见图 5),第 1 行调用了 icom_free_adapter(icom_adapter)函数,之后又调用了 pci_release_regions(icom_adapter->pci_dev)函数。前者释放了参数 icom_adapter 却又在后者函数的参数中被使用,导致释放后调用错误,该错误在 Android_2.2 版本中被修复,修复的方式是调换了两个函数调用的顺序(见图 11)。

```
1 icom_free_adapter(icom_adapter);
2 pci_release_regions(icom_adapter->pci_dev);
```

图 11 Android_2.2 drivers\serial\icom.c

实验中发现的疑似错误有些并未被修复。Android_4.0 版本中的一个疑似边界未检查错误发生在 sound/oss/pss.c 文件(见图 12)中,第 1 行函数调用了 copy_from_user(mbuf, arg, sizeof(copr_msg))方法,随后函数执行了循环 for(i=0;i<mbuf->len;i++),在该循环执行之前并没有检查 mbuf->len 的大小。编写代码的程序员熟悉 mbuf 的结构,因此并不需要对其进行检查。但在某些情况下,恶

意程序可能会使该段代码出现循环越界问题。Android_4.0 的一个疑似自旋锁阻塞错误发生在 drivers/usb/gadget/uvc_video.c 中(图 13),在第一行申请了自旋锁之后,又调用了带有标志 GFP_KERNEL 的函数,因此该段代码存在安全隐患。这些疑似错误虽没有被修复,但不意味着代码是绝对安全的,仍需进一步验证。一种称为模糊测试[11]的方法可以通过对输入进行变异而检测代码是否存在安全隐患。

```
1 copy_from_user(mbuf, arg, sizeof(copr_msg));
…
2 for(i=0;i<mbuf->len;i++){
…
```

图 12 Android_4.0 sound/oss/pss.c

```
1 spin_lock_irqsave(&video->queue.irqlock, flags);
2 buf=uvc_queue_head(&video->queue);
…
3 usb_ep_queue(video->ep, req, GFP_KERNEL);
…
4 spin_unlock_irqrestore(&video->queue.irqlock, flags);
```

图 13 Android_4.0 drivers/usb/gadget/uvc_video.c

由于篇幅原因,其他错误不再一一详述。横向实验结果分析表明,虽然基于控制流挖掘的方法不具备数据流分析能力,导致误报率较高,但对于已知模式错误,该方法的发现能力强,代码覆盖广。

4.2 纵向结果分析

从 Android 多系统版本之间的纵向分析可以看出,从 Android_1.5 开始,查出的错误总数是呈上升趋势的。从整体上看,不同错误所占的比重如图 14 所示,

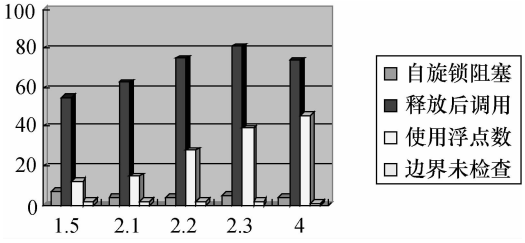

图 14 不同 Android 版本中不同错误所占的比例

从 Bug 出现的位置来看,每一个版本 Bug 出现最多的文件夹前三名如表 7 所示。图 15 表示在不同 Android 版本中的错误分布。

表 7 每个版本 Bug 出现位置前三名

版本	第一位	第二位	第三位
1.5	drivers	net	arch
2.1	drivers	net	arch
2.2	drivers	arch	net
2.3	drivers	arch	net
4.0	drivers	arch	net

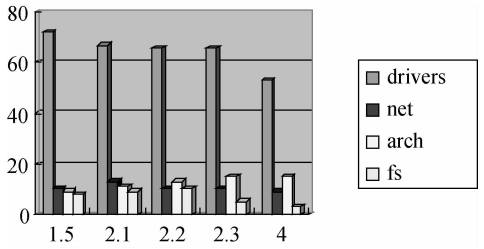

图 15 不同 Android 版本的错误分布

结论说明,驱动错误所占的比重在各版本中最大,且其比重已开始呈下降趋势,而 arch 文件夹中的错误逐渐增加。从上述结果可以看出,错误分布情况和 Android 代码演化是一致的,近年来随着各种新架构的出现,arch 目录下新代码的增加比重加大,从而导致质量和安全问题也较为严重。驱动代码的错误一致是 Linux、Android 等内核代码中错误主要类型,这主要是因为,驱动的种类繁多,很多驱动没有进行充分的测试,甚至很多驱动代码是非系统代码开发人员编写的,因此代码的错误和脆弱点较多。

6 相关工作和结束语

本文首次使用基于控制流挖掘方法对 Android 系统代码进行全面分析。通过对 Android 多系统版本的分析,展现了常见的内核错误在每个版本的出现情况,以及错误在不同文件夹的分布。Nicolas Palix 等人的工作[2]对 Linux 系统代码进行了全面的分析,版本覆盖了 2.6.0 至 2.6.33,但未对 Android 系统代码进行全面分析,尤其是对 Android 系统与 Linux 内核代码差异部分。TaintDroid 对 Android 用户数据进行污染追踪,研究者调查了 30 个流行应用程序,发现其中 15

个会向广告商服务器发送用户的地理位置信息[1]，与本文相比，该工作仅关注用户敏感信息泄漏，并不直接针对内核代码进行检测。

基于控制流挖掘方法仍有许多不足之处，其一，由于静态分析不执行代码，因此很多运行时出现的错误不能被发现。其二，误报率高，误报率与挖掘脚本的复杂度成负相关。在实验中发现，使用三条语义规则组合实现的挖掘脚本误报率较仅使用一条语义规则降低 13%。另外，为了进一步降低误报率，应与数据流分析相结合，并记录状态，以减少不可达路径产生的误报。Henrik Stuart 等人的工作[10]将挖掘脚本与 Bug 修复相结合，为分析错误并减少误报率提供了帮助。其三，新的错误模式不易被发现，需要大量的领域编程知识。Linux 内核编程需要调用内核服务程序以及直接和硬件交互，与应用程序编程有很大差别，例如对内核中浮点数的使用限制。这种差别导致在对 Bug 进行挖掘和验证时存在障碍。为了弥补该方法的不足，应将动态分析与静态分析技术相结合，并利用数据挖掘技术从海量代码中挖掘错误倾向，如 Benjamin Livshits 等人通过挖掘软件修复的历史来发现错误模式[12]。这将是本文今后研究的重点。

参 考 文 献

[1] W. Enck, P. Gilbert, B. gon Chun, et al. Taintdroid: An information-ow tracking system for realtime privacy monitoring on smartphones[C]. In Proceedings of the 9th Usenix Symposium on Operating Systems Design and Implementation, pages 393-408, August 2010.

[2] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code[C]. In 10th European Software Engineering Conference, pages 306-315, Lisbon, Portugal, 2005.

[3] Red Hat Bugzilla-Attachment 531725 Details for Bug 751297. https://bugzilla.redhat.com/attachment.cgi?id=531725&action=edit.

[4] Build number of Android source code. http://source.android.com/source/build-numbers.html.

[5] 杨丰盛.《Android 技术内幕》[M]. 北京. 机械工业出版社. 2011.5
YANG Fengsheng. Android Technology.[M]. Beijing. Machinery Industry Press. May, 2011.(In Chinese)

[6] Nicolas Palix, Ga? l Thomas, Suman Saha, et al. Faults in Linux: Ten Years Later[C]. Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, Newport Beach, California, March 2011, pages 305-318.

[7] Julia Lawall, Gilles Muller, Nicolas Palix. Finding Error Handling Bugs in OpenSSL using Coccinelle[C]. 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software, pages 7-11, Charlottesville, VA, USA, March, 2009.

[8] J. L. Lawall, J. Brunel, R. R. Hansen, et al. WYSIWIB: A declarative approach to

finding protocols and bugs in Linux code[C]. In The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 43-52, Estoril, Portugal, August 2009.

[9] Yoann Padioleau, Julia L. Lawall, Gilles Muller. Semantic Patches, Documenting and Automating Collateral Evolutions in Linux Device Drivers[C]. Ottawa Linux Symposium, June 2007.

[10] H. Stuart, R. R. Hansen, J. L. Lawall, et al. Towards easing the diagnosis of bugs in os code[C]. In PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems, pages 1-5, New York, NY, USA, 2007. ACM.

[11] MILLER B P, FREDRIKSON L, SO B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(2): 32

[12] B. Livshits and T. Zimmermann. DynaMine: A Framework for Finding Common Bugs by Mining Software Revision Histories[C]. In Proceedings of the ACM SIGSOFT 2005 Symposium on the Foundations of Software Engineering.