基于符号执行的能耗错误检测方法

徐 超[1],陈 勇[2],葛红美[1],何炎祥[3]
(1.南京审计大学,江苏南京 210029;2.中国电子科技集团第十四研究所,江苏南京 210013;3.武汉大学计算机学院,湖北武汉 430072)

摘 要: 能耗是制约便携式智能设备发展的重要瓶颈.随着嵌入式操作系统的广泛应用,因不能合理使用操作系统的API而导致的能耗错误已经成为各种嵌入式应用开发过程中不容忽视的因素.为减少应用中的能耗错误,以符号执行技术为基础,根据禁止休眠类能耗错误的特点,设计了对应的能耗错误检测方法.该方法首先利用过程内分析,获得单个函数的符号执行信息.然后借助过程间分析对单个函数的符号执行信息进行全局综合,得到更为精确的执行开销、锁变量匹配等相关信息,以更好的检测能耗错误.同时,符号执行记录了对应的分支路径信息,利用该信息能够结合约束求解器较为方便的生成出错的测试用例,进而定位错误位置.通过示例和实验,验证了该方法在能耗错误检测方面的可行性和有效性.

关键词: 能耗错误;符号执行;错误检测;过程内分析;过程间分析
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112(2016)05-1040-11
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2016.05.005

Symbolic Execution Based Energy Bug Detecting Method

XU Chao[1],CHEN Yong[2],GE Hong-mei[1],HE Yan-xiang[3]
(1. *Nanjing Audit University*,*Nanjing*,*Jiangsu* 210029,*China*;2. *The* 14*th Research Institute of CETC*,*Nanjing*,*Jiangsu* 210013,*China*;
3. *Computer School of Wuhan University*,*Wuhan*,*Hubei* 430072,*China*)

Abstract: Energy is one important bottleneck to the development of intelligent portable device. With the wide use of embedded operation system,the energy bug caused by unsuitablely using the API of the operation system has become the important factor in the designment of the embedded application. According to the characteristics of the No-Sleeping energy bug,a symbolic execution based energy bug detecting method is proposed to reduce the energy bug. It first uses intraprocedural analysis technology to analyze one function independently to get the energy information of the function. Then,the interprocedural analysis technology is applied to get the globe analysis of the program by the information of intraprocedural analysis which can get more accuate information for energy bug detection. Meanwhile,constraint solver can be combined to obtain the counter-example for locating the position of the error. Example and experiment results verify that the method is feasible and effective in energy bug detection.

Key words: energy bug;symbolic execution;bug detection;intraprocedural analysis;interprocedural analysis

1 引言

随着手持智能化设备的广泛普及,以iOS和Android等手机操作系统为基础平台的应用得到了迅猛发展.但与此同时,由于手机、平板电脑等手持设备电池容量的限制,很多应用因其使用时的高能耗而使得其难以得到用户的支持.如何降低这些手机应用运行时的能耗已经引起了不少专家学者的关注.其中,Abhinav Pathak等人指出,除了应用本身功能需要的能耗开销外,软件中存在的能耗错误是能源消耗的一个重要因素.2011年,Abhinav Pathak等人[1]以智能手机为基础,从软硬件的角度现有的移动设备中的能耗问题进行了分析,首次提出了能耗错误(ebug)的概念,并提出了一种基于eprof[2]等能耗分析工具的能耗错误检测和调试框架.随后,Abhinav Pathak等人[3]以及Panagiotis Vekris等人[4]针对No-Sleep的能耗错误,分别利用到达-定义链和过程间数据流分析的方法对该类能耗错误进行检测.Adam J. Oliner等人[5]基于统计的思想,利用其开发的Carat搜集智能设备中各个应用在各种环境下消耗的电量,并将其传递给云服务器.然后通过云服务器对数据的统计分析,给出各个应用中可能的能耗错误以

收稿日期:2015-09-27;修回日期:2016-01-04;责任编辑:马兰英
基金项目:国家自然科学基金(No.61170022);江苏省高校自然科学研究面上项目(No.15KJB520019);江苏省"六大人才"高峰项目资助;江苏省高校"青蓝工程"优秀青年骨干教师培养对象资助;江苏高校优势学科建设工程资助项目

及较好的使用方法,传递给用户以指导其较好的配置其智能设备,获得能耗的节约,提高设备的持续使用时间. 最近,Abhilash Jindal[6]等人从智能设备的更底层应用出发,分析了软件驱动程序中存在的休眠冲突问题而导致的能耗错误,并设计了一种避免该类错误的系统以提高设备的能效.

能耗错误主要是指某些设备长期处于无效工作状态而导致的能源消耗. 其错误的主要包括禁止休眠(No Sleep)错误,循环询问错误等. 禁止休眠错误是由于使用了不对称的 API 使得设备无法进入休眠状态而导致的无效能源消耗,如 Android 系统中使用了 PARTIAL_WAKE_LOCK,但当该部件工作结束后未使用对应的 UNLOCK 操作,使得该部件一直处于激活状态消耗大量能耗. 循环询问错误是指某些应用一直访问无法应答的部件而导致的无效能源消耗. 如访问远程服务器的程序,为保证连接的稳定性,程序员有可能采用无限循环频繁尝试连接服务器,直至连接到服务器. 但如果远程服务器已经崩溃,该程序将因频繁的连接操作而导致大量无效能源消耗. 根据文献[3]的研究结果,70%的能耗错误属于禁止休眠类能耗错误,检测和纠正禁止休眠类能耗错误是能耗错误检测工作的重点.

但这些能耗错误并不会影响程序的正确运行,以保障程序正确转换为基本目标的传统编译器对该类错误的检测几乎无能为力,软件开发者很难在软件开发过程中有效发现该类错误. 且鉴于禁止休眠类能耗错误的重大比例,本文将主要针对禁止休眠类能耗错误进行分析检测.

符号执行技术是上个世纪 70 年代提出的一种路径敏感静态分析方法[7,8]. 其基本思想是以符号的形式替换输入,根据程序中每条语句的具体语义,模拟程序的执行. 当遇到分支条件时,则将该条件增加到对应的后续语句的执行约束中. 其分析的最终结果将获得程序中每条执行路径及其需要满足的约束条件映射表,我们可以利用约束求解器从该结果中获得执行某条程序路径的一组输入数据组合,进而获得对应路径的分析结果及其测试用例.

由于符号执行技术能够对程序进行路径敏感分析,模拟程序各执行路径的状态信息,不但能够较好的检测加解锁匹配、加解锁操作时间间隔等与禁止休眠类能耗错误主要特征相关的问题,而且其记录的路径约束映射表也可以帮助生成对应的测试用例,帮助定位错误的位置. 因此,本文将结合符号执行技术,设计一种对禁止休眠类能耗错误的检测方法.

2　禁止休眠类能耗错误简介

禁止休眠错误是由于使用了不对称的 API 使得设备无法进入休眠状态而导致的无效能源消耗. Abhinav Pathak 等人通过进一步的研究[3],将禁止休眠类能耗错误进一步划分为 3 个类别:路径型禁止休眠(No-Sleep Code Paths)、竞争型禁止休眠(No-Sleep Race Condition)和扩张型禁止休眠(No-Sleep Dilation). 路径型禁止休眠能耗错误主要是指单线程的应用中的某条路径存在未释放的能耗相关锁. 其错误模式主要有三种:(1)当锁释放操作在条件分支中时,该操作只在部分分支中进行了释放操作,而不是所有分支都进行了释放操作;(2)当锁的释放操作在有可能出现异常的块中,由于没有在异常捕获块(即 catch 块)中添加对应的锁释放操作,而有可能在程序执行异常时出现锁未释放的情况;(3)虽然程序中存在对应的释放锁操作,但由于上层应用存在其他资源的死锁(如应用级的死锁),使得程序无法进入所释放的程序点而使得能源的大量消耗.

竞争型禁止休眠能耗错误主要针对锁操作在多线程中调用的情况:一个线程进行加锁操作,另一个线程进行释放锁操作. 由于线程执行顺序的问题,程序有可能出现释放操作的线程先于加锁操作的线程而无法完成锁的最终释放操作.

扩张型禁止休眠能耗错误是指加解锁锁操作虽然匹配,但在加解锁操作之间花费了比期望更多的时间进行相应的操作. 产生这类锁的原因主要有两种:(1)由于延迟,在获得了锁以后并非直接进行锁相关操作,而是等待其他信号到来再进行相应操作,但该操作所需时间可能无法估计(如人工交互);(2)在加解锁操作之间增加了大量与该锁操作无关的处理.

由以上三类禁止休眠类错误可以看出,其出错的因素主要同锁变量的加载与释放操作以及路径的执行情况相关. 因此如果能对锁变量进行跟踪,找到其最坏执行路径进行分析,则可以较为容易的获得对应的错误,而这正是符号执行技术主要的特点,因此本文将主要以该技术为基础进行分析.

3　符号执行技术

符号执行经历了近 40 年的研究,针对不同环境形成了一系列分析方法[9~14],出现了许多分析工具,如 JPF-SE[15],Klee[16],EXE[17],CUTE[18]等,但其分析过程也难以避免的存在一定的缺陷[19~21]:

(1)它难以处理复杂的数据结构. 如当赋值给数组下标为变量的数组元素时,则难以进行精确的确定符号.

(2)由于它需要记录所有路径的约束条件,因此当程序较大时,一方面将会遇到路径爆炸问题,严重影响符号执行的效率. 另一方面也将导致大量约束条件的产生,影响约束求解器的效率.

(3)当遇到循环时,其模拟符号执行将可能进入死

循环.

(4)当程序较大时,如果其对函数调用进行递归分析,将会由于调用函数过多或者是递归函数而使得分析开销十分巨大.

因此,为使得本文的分析技术能够有效的进行,本文根据锁变量在程序中出现的特点,对符号执行的分析过程进行一定的简化.

(1)由于本文主要针对锁变量的加载与释放进行分析,而锁变量通常不是数组元素等复杂数据结构,因此对于下标为变量的数组元素赋值,本文将采用懒赋值的方式将其赋值为“不确定”.

(2)由于本文是尽最大可能查找出程序中所有可能存在的能耗错误,因此将只需找出每个锁变量在最坏情况下的路径,因此记录的路径约束只限于锁变量最坏情况下的那天分支,以避免路径爆炸和约束求解器过大的求解复杂度.

(3)由于锁变量一般较少出现在循环体中,而且一旦其出现在其中,且是计数型的加锁操作,则将很有可能导致锁释放的不匹配,因此对于循环体,本文将不对其进行迭代分析,只是根据迭代次数将其定义的变量的符号值设定为“不确定”,以避免无限迭代.

(4)由于本文只关心最坏情况下锁变量的情况,因此对于函数调用,本文只根据该函数中的锁变量最坏情况下状态值进行迭代分析,而不重新将调用参数作为输入值对其进行递归分析,以避免函数递归分析过大的分析开销.

4 基于符号执行技术的能耗错误分析方法

由于能耗错误产生的主要原因是程序员在申请系统资源的过程中没有正确使用锁操作,使得其长期处于无效工作状态而造成的大量能源消耗. 能耗错误同加解锁操作的程序点和种类有着十分密切的关系. 为检测程序中可能存在的能耗错误,我们不但需要检测在最坏情况下是否存在锁操作不匹配的情况,以发现路径型禁止休眠类错误,而且还需要获得尽可能精确的加解锁操作之间的时间间隔,以提醒程序员核查那些出现在加解锁操作之间而等待时间长的代码段是否与其加解锁的资源相关,从而最大程度避免扩张型禁止休眠类错误的产生. 同时,为能够较为精确的定位能耗错误,还需要获得最坏情况下程序执行的路径信息,以便于反向跟踪对应的错误位置. 本节将主要结合符号执行技术的思想,对程序的语义进行扩充,以检测和定位程序中禁止休眠类能耗错误,进一步提高程序的能效. 由于竞争型禁止休眠类错误主要同线程的调度相关,而一旦明确了线程调度的序列,则可以将其归约为面向路径型和扩张型禁止休眠类能耗错误的检测问题,因此本节将主要讨论路径型和扩张型禁止休眠类能耗错误的检测.

4.1 能耗错误过程内分析

在进行能耗错误过程内分析时,本文首先对源文件进行预处理,将锁操作相关函数调用转换为对变量的整数运算,并将对应的转换后的变量保存在锁符号集合“S”中,以便于能耗错误分析时对该变量进行识别. 与文献[3]不同,本文将该文献中的两种锁操作类型扩展为四种类型的锁操作,即在原始赋值类型的锁操作基础上增加了自增和自减两种锁操作,使得该能耗检测程序既能够处理信号量型资源锁,也能够处理变量型资源锁. 其预处理的示例结果如下所示. 在该程序示例中,我们将锁操作变量 wm 和 wl 转为对应的整型变量. 并根据其不同的锁操作类型(wm 为计数锁,wl 为信号锁),分别在锁加载(acquire)以及锁释放操作(release)时采用赋值操作(转换后程序(b)中行 7 和 10)以及自增操作(转换后程序(b)中行 8、14、16 和 22). 同时,为处理源程序 (a) 中 24 行针对锁变量为空的判断而增加的锁释放操作,本文按文献[3]中所示方法增加对应的“else”操作(转换后程序(b)中行 23 和 24 所示). 能耗分析预处理示例如下:

```
1.  public class Test{
2.    PowerManager pm = ( PowerManager) getSystemService( );
3.    PowerManager. WakeLock wm = pm. newWakeLock( );
4.    String datas;
5.    int k;
6.    void OnCreate( ) {
7.      PowerManager. WakeLock wl = pm. newWakeLock( );
8.      wl. setReferenceCounted (false);
9.      wl. acquire( );
10.       wm. acquire( );
11.       net _ sync( );
12.       wl. release( );
13.   }
14.   net _ sync( ) {
15.     if( k < 3) {
16.       wm. acquire( );
17.       net _ working( );
18.       wm. release( );
19.     }
20.   }
21.   net _ working( ) {
      //C( getServerData) = MaxCost
```

```
1.  public class Test{
2.    int wm = 0;
3.    String datas;
4.    int k;
5.    void OnCreate( ) {
6.      int wl = 0;
7.      wl = 1;
8.      wm = wm + 1;
9.      net _ sync( );
10.     wl = 0;
11.   }
12.   net _ sync( ) {
13.     if( k < 3) {
14.       wm = wm + 1;
15.       net _ working( );
16.       wm = wm - 1;
17.     }
18.   }
19.   net _ working( ) {
20.     //C ( getServerData) = MaxCost
21.     datas = getServerData ( );
22.     if( wm ! = null)
23.       wm = wm - 1;
24.     else
```

```
22.         datas = getServerData();        25.              wm = wm - 1;
23.         if(wm ! = null)                 26.          }
24.           wm.release();                 27.      }
25.     }
26. }                                       S = {wl,wm}
```

(a)源程序　　　　(b)转换后程序

由于文献[22]中所针对的语言非常简单,而且又很强的表达能力,既能表达像汇编语言这种较低层次的语言,又能表达像 Java 这样的高级语言,同时它也是许多编程语言在编译器分析时实际使用的中间表达形式的语言,具有较好的通用性. 而且 Java 语言正是本文检测所针对的 Android 系统所使用的语言. 因此,在经过对源程序的预处理,将锁变量操作转为整型变量的操作后,本文主要以该语言为基础,在增加了函数调用以及块成分后对其进行分析处理,转换后的语法如下所示.

```
program p:: = function *
    function f:: = block *  returnExit
    block b:: = stmt *  blockExit
    stmt s:: = var: = exp | store(exp, exp) | goto exp | assert exp | if exp
              then block else block | exp
    exp e:: = load(exp) | exp ◇b exp | ◇u exp | var | get input(src) | call
f
    ◇b:: = typical binary operators
    ◇u:: = typical unary operators
    value var:: = 32-bit unsigned integer | exp
```

为使得符号执行能够获得能耗错误检测过程所需信息,本文对传统的符号执行进行了修改:(1)在符号执行过程中加入执行时间的估计信息,以便于提取 No-Sleep Dilation 类错误;(2)本文的符号执行将主要针对锁变量相关路径进行懒符号执行分析,以避免符号执行过程中谓词约束条件过长和路径爆炸等问题. 转换后的语法成分的符号执行过程如表 1 所示.

表 1　各语法成分符号执行扩展语义

1. function f:: = block * returnExit	2. block b:: = stmt * blockExit
$(e,\sigma,P,C,S,\Sigma,\iota,\partial)\leftarrow$SymExec(*block* *) $\forall x:(\sigma_x,P_x,C_x)\leftarrow Max_\sigma(\Re)$ return $(\sigma[S],P[S],C[S],CE,FE)$	$(e,\sigma,P,C,S,\Sigma,\iota,\partial)\leftarrow$SymExec(*stmt* *) $\partial\leftarrow\partial\cup\{\iota\}$ $\forall x:(\sigma_x,P_x,C_x\times N_b)\leftarrow Max_\sigma(\Im)$ if $N_b>1$ $\forall x\in def(b):x\leftarrow\perp$ $\iota\leftarrow\Sigma[\text{blockExit}_b]$
3. stmt s:: = store(exp′,exp″)	4. stmt s:: = assert(exp)
$\partial\leftarrow\partial\cup\{\iota\}$ $\forall x:C_x\leftarrow C_x+C(\text{exp}')+C(\text{exp}'')+2$ $C_f\leftarrow C_f+C(\text{exp}')+C(\text{exp}'')+2$ $\iota\leftarrow\iota+1$	$\partial\leftarrow\partial\cup\{\iota\}$ $\forall x:C_x\leftarrow C_x+C(\text{exp})$ $C_f\leftarrow C_f+C(\text{exp})$ $\iota\leftarrow\iota+1$
5. stmt s:: = var: = exp	6. stmt s:: = goto exp

$\partial\leftarrow\partial\cup\{\iota\}$
if($var\in S\&\&\sigma_{var}\neq 0$) then
if($C_{var}>\tau$)
CE_{var}.add(<l_{var},ι,P_{var}>)
else if(var $\in S\&\&\sigma_{var}=0$) then
$C_{var}=0,l_{var}=\iota$
$\iota\leftarrow\iota+1$
$\forall x:C_x\leftarrow C_x+C(\text{exp})+1$
$\forall x:\sigma_x\leftarrow\sigma_x[\text{val(exp)/var}]$
$C_f\leftarrow C_f+C(\text{exp})+1$
	$\partial\leftarrow\partial\cup\{\iota\}$
	$\forall x:C_x\leftarrow C_x+C(\text{exp})+1$
	$C_f\leftarrow C_f+C(\text{exp})+1$
	$\iota\leftarrow\Sigma(\text{val(exp)})$
	if(ι = blockExit)
	$\Im\leftarrow\Im\cup\{(e,\sigma,P,C,S,\Sigma,\iota)\}$
	else if(ι = returnExit)
	$\Re\leftarrow\Re\cup\{(e,\sigma,P,C,S,\Sigma,\iota)\}$

7. exp:: = $\Diamond_u$ exp′	8. exp: = load(exp′)
$\forall x\in\text{use(exp}')$: $e\leftarrow val(\Diamond_u\text{exp}'[\sigma_x/x])$ $C(\text{exp})\leftarrow C(\text{exp}')+C(\Diamond_b)$	$e\leftarrow\perp$ $C(\text{exp})\leftarrow C(\text{exp}')+2$
9. exp:: = var	10. exp:: = get _ input(src)
$e\leftarrow\text{val(var)}$ $C(\text{exp})\leftarrow C(\text{var})$	$e\leftarrow\text{val(src)}$ $C(\text{exp})\leftarrow 2$
11. exp:: = exp′ $\Diamond_b$ exp″	
$\forall x_1\in\text{use(exp}'),x_2\in\text{use(exp}'')$: $e\leftarrow val(\text{exp}'[\sigma_x/x_1]\Diamond_b\text{exp}''[\sigma_x/x_2])$ $C(\text{exp})\leftarrow C(\text{exp}')+C(\text{exp}'')+C(\Diamond_b)$	
12. exp:: = call f	
$(\sigma,P,C)\leftarrow\text{ret}(f)$	
13. stmt:: = exp	
$\partial\leftarrow\partial\cup\{\iota\}$ $\iota\leftarrow\iota+1$ $C_f\leftarrow C_f+C(\text{exp})+1$ $\forall x:C_x\leftarrow C_x+C(\text{exp})+1$ $\forall x:\sigma_x\leftarrow\sigma_x+\sigma_{\text{exp},x}$ $\forall x:P_x\leftarrow P_x+P_{\text{exp},x}$	

续表

14. stmt s : : = if exp then block$'$ else block$''$
if$(\iota \in \partial)$ then
return
$\partial \leftarrow \partial \cup \{\iota\}$
$(e',\sigma',P',C',S,\Sigma,\iota',\partial) \leftarrow \mathrm{SymExec}(\mathrm{block}')$
$(e'',\sigma'',P'',C'',S,\Sigma,\iota'',\partial) \leftarrow \mathrm{SymExec}(\mathrm{block}'')$
if$(\iota' \in \partial)$ then
$\forall x \in S$:
if$(\sigma'_x - \sigma_x > 0)$
$FE_x.\mathrm{add}(P_x)$
if$(\iota'' \in \partial)$ then
$\forall x \in S$:
if$(\sigma''_x - \sigma_x > 0)$
$FE_x.\mathrm{add}(P_x)$
$\forall x \in S$:
if$((\sigma'_x > \sigma''_x) \&\& (P_x \cap \mathrm{val}(\exp) = \varnothing))$ then
$e \leftarrow e', \sigma_x \leftarrow \sigma', P_x \leftarrow P'_x \cup \{\mathrm{val}(\exp)\}, C_x \leftarrow C_x +$
$C' + C(\mathrm{val}(\exp)), \iota \leftarrow \iota'$
else if$((\sigma'_x > \sigma''_x) \&\& (P_x \cap \neg\mathrm{val}(\exp) = \varnothing))$ then
$e \leftarrow e'', \sigma_x \leftarrow \sigma'', P_x \leftarrow P''_x \cup \{\neg\mathrm{val}(\exp)\}, C_x \leftarrow C_x$
$+ C'' + C(\exp), \iota \leftarrow \iota''$
else
$e \leftarrow e', \sigma_x \leftarrow \sigma', P_x \leftarrow P'_x \cup \{\mathrm{val}(\exp) \parallel \neg$
$\mathrm{val}(\exp)\}, C_x \leftarrow C_x + C' + C(\exp), \iota \leftarrow \iota'$
if$(C' > C'')$ then
$C_f \leftarrow C_f + C' + C(\exp), CP_f \leftarrow CP_f \cup \exp$
else if$(C' < C'')$ then
$C_f \leftarrow C_f + C'' + C(\exp), CP_f \leftarrow CP_f \cup \neg\exp$
else
$C_f \leftarrow C_f + C'' + C(\exp)$

在修改后的符号执行过程中,本文采用八元组(e,σ,P,C,S,Σ,ι,∂)记录符号执行的状态信息. 其中 e 记录当前表达式的值,σ、P 和 C 分别表示符号表、约束条件以及执行开销值的映射表. 这三个符号以符号名为索引值,构建对应符号的相关信息. S 为当前程序中使用的锁变量对应的符号名,Σ 为程序地址到对应语句的映射,ι 为下一条分析的程序地址. ∂记录已经分析过的语句地址集合. 此外,为解决多分支合并问题,本文在符号执行过程中增加了两个状态保存链表 $\Im$ 和 $\Re$,分别用于保存当前分支块结构所有可能的运行结果信息和当前函数返回状态信息. 当遇到块结束标记(或函数返回标记)时,依次遍历每个锁变量 x,通过对比对应链表中 σ_x 值的大小,以获取最坏情况下的分支状态信息(即通过 Max_σ 函数选择 σ_x 值最大的分支作为最坏分支). 同时为记录 No-Sleep Dilation 类错误和循环内错误,使用锁变量位置信息映射表 l 跟踪锁操作,使用两个全局映射表 CE 和 FE 记录对应锁变量的相应错误. 其具体对应关系如表 2 所示.

表 2　符号执行状态信息

状态符号	含义
e	表达式的符号表示值
σ	变量名与其值的映射
P	锁变量与其最坏情况下约束条件的映射
C	锁变量及函数与其执行开销累积计算值的映射
S	锁变量集合列表
Σ	程序计数器的值与其对应语句的映射
ι	下一条指令地址
∂	已分析过的语句地址
$\Im$	块内所有分支可能运行结果的状态列表
$\Re$	函数内所有分支可能运行结果的状态列表
l	锁变量位置信息映射表
CE	锁变量与其 No-Sleep Dilation 类错误的映射
FE	锁变量与其循环内不确定加速操作的映射
CP_f	函数 f 的最大执行开销对应的约束条件
val(exp)	计算表达式 exp 值的函数
SymExec(B)	对块 B 进行符号执行
ret(f)	函数 f 的符号执行结果(初始值为空)
$\sigma_x[e/\mathrm{var}]$	用 e 替换 x 值中 var 并重新进行常量合并计算以更新符号表中的 x 值
N_b	块 b 执行次数
use(exp)	表达式 exp 中使用的符号
def(b)	块 b 中定义的符号
$\perp$	不确定值

由于锁变量通常不作为函数参数传递,因此为减少分析开销以及路径爆炸等问题,本文对函数的输入参数采用懒赋值方法进行赋值(即赋值为不确定"$\perp$"). 同时,为便于分析,本文将所有的返回语言均使用 goto 语句跳转到唯一的函数出口 returnExit. 每当遇到函数 f,则采用规则 1,首先对其内所属块依次调用符号执行(即(e,σ,P,C,S,Σ,ι,∂)←SymExec(block *)),将可能包含最坏执行情况的符号执行结果的状态信息保存到 $\Re$ 列表中,然后依次分析 S 集合中的每个锁变量 x,由于根据锁变量的整形转换方法,当锁变量的值越高时,说明在该函数中其加锁操作也越多,这就越可能使得锁释放操作的不足,所以该分支也将是针对于锁变量的最坏分支. 因此我们将 σ_x 值最大的分支状态信息作为该函数对该所变量 x 的符号执行结果返回(即 $\forall x$,$(\sigma_x, P_x, C_x) \leftarrow Max_\sigma(\Re)$). 最后返回该最坏情况下与锁变量相关的符号表信息、约束条件信息以及执行

开销信息,以便于在过程间分析时计算其锁匹配、错误定位以及执行开销等信息.

对于块操作,其执行过程类似于函数(如规则 2),也是按照语句次序依次执行块内每条语句,保存该块中可能包含最坏执行情况的符号执行结果状态信息到 $\Im$ 列表中. 然后更新该语句为已分析($\partial \leftarrow \partial \cup \{\iota\}$),将每个锁变量的最坏执行结果作为该块的最终执行结果,传递给后续语句继续分析. 由于可能存在循环,一个块可能执行多次,因此块的执行开销还需要增加对应块的执行次数. 同时,当块的执行次数不为 1 时,由于其处于循环内,符号执行可能使得其块内定义过的符合出现任意值,因此我们设定该块内定义过的符号值为不可知状态. 最后,我们设定其后续执行序列为块出口语句($\iota \leftarrow \Sigma[\mathrm{blockExit}_b]$).

在对一般语句分析时,其基本操作时修改当前已分析语句(即∂值)、更新当前执行开销值(即 C 值)和设定下一条执行语句(即 ι 值)(如规则 3、4 等). 对于赋值语句,由于其将修改对应符号的当前值,因此需进一步更新对应符号的符号表($\sigma_x \leftarrow \sigma_x[\mathrm{val}(\exp)/\mathrm{var}]$). 同时对于锁变量,如果当前的值不为 0(即为加锁状态),且当前的执行开销(C_x)超过执行开销阈值 τ,则说明该锁将可能长期处于加锁状态,因而将其加入对应的 *CE* 列表中. 而如果此时锁变量的值为 0,则说明此时锁在该操作前处于关闭状态,因而重置执行开销值 $C_x = 0$. 对于 goto 语句,则需根据跳转的位置添加对应的状态信息到块状态列表 $\Im$ 和函数状态列表 $\Re$ 中.

对于 if 条件语句,由于可能存在循环,因此先判断该语句是否已经被分析,如果已经被分析,则说明是循环的节点,不作处理,直接执行后续指令. 当是第一次分析该语句时,则分别对 if 分支和 else 分支进行符号执行. 如果其中某条分支跳转到已分析指令(即 $\iota \in \partial$),则说明该块是循环中块. 由于循环执行次数通常难以计算,因此在循环的一次迭代内出现锁变量的局部增益大于 0,则该循环体很大情况下会出现锁的不匹配情况,因而此时给出对应的锁不匹配的提示信息,将其加入对应的 *FE* 列表中. 否则对每个锁相关变量 x,分别比较 if 两个分支中该变量的符号执行值,选择其最坏执行块(即 σ_x 值最大的块)作为该块的执行结果,当两个块结果一样时,则随机选择一条分支(本文选择分支 1)加入该锁变量的执行结果中. 同时按其选择的结果分支将对应的 *if* 条件表达式加入到符号执行的约束变量中. 其具体描述如表 1 规则 14 所示.

对于表达式,则主要根据其操作类型,完成对表示式 e 的符号计算以及执行开销的计算,为赋值等一般语句准备对应的数据.

在构建了符号执行策略后,过程内分析如算法 1 所示. 对于程序中的每个函数,自顶向下依次对程序进行分析,根据其匹配的符号执行规则进行相应的操作,获得对应的符号执行的状态信息.

算法 1　过程内分析算法

输入:待分析函数集合 *prog*,锁变量集合 S,执行开销初始信息 C_0

输出:锁变量相关变量符号执行信息 $L(\sigma, P, C)$

1. foreach $f \in prog$
2. $\quad L_f \leftarrow \mathrm{SymExec}(f, S, \{\{\forall x \in S \mid <x, 0>\}, \varnothing, C_0\})$
3. endfor
4. return L

4.2　能耗错误过程间分析

在过程间分析时,为避免路径爆炸和约束求解过于复杂,本文只对能耗错误相关变量进行过程间分析,对符号执行采用懒赋值方式(即函数调用结果赋值为 $\perp$). 其过程间分析如算法 2 所示.

在过程间分析中,本文以迭代分析的方法依次计算程序中的每个函数,直到所有函数的计算结果均不发生变化(即达到不动点)时,结束循环(行 2 – 11). 每次迭代过程中,将函数f 的符号执行结果 L_f' 同上次迭代的执行结果 L_f 进行比较,以判断是否发生变化(行 6). 当所有函数均达到不动点后,将符号执行过程中检测的 No-Sleep Dilation 类错误列表 *CE* 和循环内加锁错误列表 *FE* 加入最终错误列表(行 12),然后对入口函数的符号执行结果进行分析,找出加解锁不匹配的操作变量(即锁变量值不为 0 的元素)及其对应的路径信息(即 P 值)加入错误列表中(行 13 – 18),最终将统计的错误信息 EBugInfo 返回.

算法 2　能耗错误过程间分析算法

输入:待分析函数集合 prog,锁变量 S,过程内分析结果 L,入口函数 entry

输出:能耗错误信息 EBugInfo

1. flag = true
2. while(flag)
3. $\quad$ flag = false
4. $\quad$ foreach $f \in$ prog
5. $\quad\quad L_f' \leftarrow \mathrm{SymExec}(f, S, L)$
6. $\quad\quad$ if $L_f \neq L_f'$ then
7. $\quad\quad\quad$ flag = true
8. $\quad\quad$ endif
9. $\quad\quad L_f \leftarrow L_f'$
10. $\quad$ endfor
11. endwhile
12. EBugInfo. add(*CE*), EBugInfo. add(*FE*)
13. $(\sigma, P, C) \leftarrow L_{entry}$
14. foreach $v \in S$
15. $\quad$ if $\sigma_v \neq 0$ then
16. $\quad\quad$ EBugInfo. add($<x, P_x>$)
17. $\quad$ endif

18. endfor
19. return EBugInfo

4.3 基于懒替换的符号执行策略

基于懒替换的符号执行是文献[23]中提出的缓解传统符号执行符号计算过于复杂,时间和空间开销过大而引入的有效方法. 其基本想法是在符号执行过程中,放宽必须进行符号替换的要求,减少不必要的符号替换和符号计算.

如对于 for 循环语句,如下所示:

```
for(i=0;i<high;i++)
    {
        c += calc(i);
    }
```

当 high 为函数参数时,若使用传统的符号执行跟踪 c 值,则要么通过限定 high 值进行有限长度的展开,其不但需要多次计算,而且每次计算结果均要保存. 即当设 high 值限定为 K 值时,对于 c 变量,通过该步骤,需要计算 K 次,为 c 保存 K 个空间. 当 c 继续进行后续操作时,将需要对每个 c 值进行计算,从而导致其计算次数和保存空间成指数增长.

但对于懒替换策略,则直接将 c 认为是不确定值,从而将原来的指数增长变为了常数空间. 由于本文分析的主要是锁变量,其要求前后必须完全匹配,因此当出现不确定值时,则可说明其存在风险的可能性极大. 当然也存在不准确的情况,如下所示:

```
for(i=0;i<high;i++)
        c += calc(i);
    for(i=0;i<high;i++)
        d += calc(i);
    if(c==d)
        goto right;
    else
        goto error;
```

但根据后续实验的检测结果可以看出,所有测试用例的检测结果均为出现误检测率,说明该情况出现的概率极低,本文的检测方法具有较强的适应性.

5 应用举例

为更好的说明以上能耗错误分析方法,本文以能耗分析预处理源程序为例,对 OnCreate 函数进行分析,其第一次迭代过程分析过程中各状态及使用的规则如表 4 所示,其中初始输入数据如表 3 所示. 假设函数 getServerData 主要完成从服务器端获得数据,因而该函数有可能因网络问题而处于长时间等待状态,因此初始化该函数的执行开销为最大值 τ.

表 3 初始输入数据

$S=\{wm,wl\}$
$prog=\{$OnCreate, net_sync, net_working$\}$
$C_0=\{$ <OnCreate,0>, <net_sync,0>, <net_working,0>, <getServerData,ι> $\}$
entry: OnCreate

表 4 OnCreate 函数第一次迭代分析过程

OnCreate 状态信息(1)

pc	e	σ	P	$CP:C$	ι	l	rule
6	0	$<wm,0>,<wl,0>$	$\varnothing$	$<C(\exp),0>$	6	$\{\}$	9
	0	$<wm,0>,<wl,0>$	$\varnothing$	$<wl,1>,<C_f,1>$	7	$<wl,6>$	5
7	1	$<wm,0>,<wl,0>$	$\varnothing$	$<wl,1>,<C(exp),0>$	7	$<wl,6>$	9
	0	$<wm,0>,<wl,1>$	$\varnothing$	$\{<wl,2>,<C_f,2>$	8	$<wl,6>$	5
8	1	$<wm,0>,<wl,1>$	$\varnothing$	$<wl,1>,<C(exp),1>$	8	$<wl,6>$	11
	1	$<wm,1>,<wl,1>$	$\varnothing$	$<wl,4>,<wm,2>,<C_f,4>$	9	$<wl,6>,<wm,8>$	5
9	$\perp$	$<wm,1>,<wl,1>$	$\varnothing$	$<C(exp),0>$	9	$<wl,6>,<wm,8>$	12
	$\perp$	$<wm,1>,<wl,1>$	$\varnothing$	$<wl,4>,<wm,2>,<C_f,4>$	10	$<wl,6>,<wm,8>$	13
10	0	$<wm,1>,<wl,1>$	$\varnothing$	$<C(exp),0>$	10	$<wl,6>,<wm,8>$	9
	0	$<wm,1>,<wl,0>$	$\varnothing$	$<wl,5>,<wm,3>,<C_f,5>$	*blockExit*	$<wl,6>,<wm,8>$	5,6
	0	$<wm,1>,<wl,0>$	$\varnothing$	$<wl,5>,<wm,3>,<C_f,5>$	*returnExit*	$<wl,6>,<wm,8>$	2,6
	0	$<wm,1>,<wl,0>$	$\varnothing$	$<wl,5>,<wm,3>,<C_f,5>$		$<wl,6>,<wm,8>$	1

OnCreate 状态信息(2)

pc	$\Im$	$\Re$	∂	CE	FE	rule

续表

6		∅	∅	∅	∅	∅	9
		∅	∅	6	∅	∅	5
7		∅	∅	6	∅	∅	9
		∅	∅	6,7	∅	∅	5
8		∅	∅	6,7	∅	∅	11
		∅	∅	6,7,8	∅	∅	5
9		∅	∅	6,7,8,9	∅	∅	12
		∅	∅	6,7,8,9	∅	∅	13
10	∅		∅	6,7,8,9	∅	∅	9
	$\{\{<wm,1>,<wl,0>\},\varnothing,\{<wl,5>,<wm,3>,<C_f,5>\},\varnothing,\varnothing\}$		∅	6,7,8,9,10	∅	∅	5,6
	$\{\{<wm,1>,<wl,0>\},\varnothing,\{<wl,5>,<wm,3>,<C_f,5>\},\varnothing,\varnothing\}$		$\{\{<wm,1>,<wl,0>\},\varnothing,\{<wl,5>,<wm,3>,<C_f,5>\},\varnothing,\varnothing\}$	6,7,8,9,10	∅	∅	2,6
	$\{\{<wm,1>,<wl,0>\},\varnothing,\{<wl,5>,<wm,3>,<C_f,5>\},\varnothing,\varnothing\}$		$\{\{<wm,1>,<wl,0>\},\varnothing,\{<wl,5>,<wm,3>,<C_f,5>\},\varnothing,\varnothing\}$	6,7,8,9,10	∅	∅	1

返回值:
$=\{<wm,1>,<wl,0>\};P=\varnothing;C=\{<wl,5>,<wm,3>,<C_f,5>\};CE=\varnothing;FE=\varnothing$

表 5　各函数多次迭代结果分析

第二次迭代	返回结果	变化标记 flag
OnCreate 函数	$\sigma=\{<wm,1>,<wl,0>\};P=\{<wm,(k<3 \| k\geqslant 3)>\};C=\{<wl,7>,<wm,7>\};CE=\varnothing;FE=\varnothing$	true
net_sync 函数	$\sigma=\{<wm,0>,<wl,0>\};P=\{<wm,k\geqslant 3>\};C=\{<wm,\tau>,(k<3):<C_f,\tau>\};CE=\{[14,16,k<3]\};FE=\varnothing$	true
net_working 函数	$\sigma=\{<wm,-1>\};P=\{<wm,(wm\neq null \| wm=null)\};C=\{<wm,3>,<C(f),r>\};CE=\varnothing;FE=\varnothing$	false
第三次迭代	**返回结果**	**变化标记 flag**
OnCreate 函数	$\sigma=\{<wm,1>,<wl,0>\};P=\{<wm,k\geqslant 3>\};C=\{<wl,\tau>,<wm,k\geqslant 3>\};CE=\{[6,10,k<3],[14,16,k<3]\};FE=\varnothing$	true
net_sync 函数	$\sigma=\{<wm,0>,<wl,0>\};P=\{<wm,k\geqslant 3>\};C=\{<wm,\tau>,(k<3):<C_f,\tau>\};CE=\{[14,16,k<3]\};FE=\varnothing$	false
net_working 函数	$\sigma=\{<wm,-1>\};P=\{<wm,(wm\neq null \| wm=null)\};C=\{<wm,3>,<C(f),\tau>\};CE=\varnothing;FE=\varnothing$	false

续表

第四次迭代	返回结果	变化标记 flag
OnCreate 函数	$\sigma=\{<wm,1>,<wl,0>\};P=\{<wm,k\geqslant 3>\};C=\{<wl,\tau>,<wm,\tau>\};CE=\{[6,10,k<3],[14,16,k<3]\};FE=\varnothing$	false
net_sync 函数	$\sigma=\{<wm,0>,<wl,0>\};P=\{<wm,k\geqslant 3>\};C=\{<wm,\tau>,(k<3):<C_f,\tau>\};CE=\{[14,16,k<3]\};FE=\varnothing$	false
net_working 函数	$\sigma=\{<wm,-1>\};P=\{<wm,(wm\neq null \| wm=null)\};C=\{<wm,3>,<C(f),\tau>\};CE=\varnothing;FE=\varnothing$	false

表 5 给出各迭代遍中各函数的返回结果及变化标记信息. 有该表可以看出,经过 4 次迭代后,3 个函数的结果集均达到不动点,即变化标记值均为“false”. 此时从入口函数 OnCreate 的返回结果可以看出,其 CE 集合中有两个元素,即“$[6,10,k<3]$”和“$[14,16,k<3]$”,从而表明在程序的行 6 到行 10 以及行 14 到行 16 之间,一旦满足 $k<3$ 的条件时,可能存在的 No-Sleep Dilation 类错误(主要由于 getServerData 长时间等待操作). 从最终锁变量映射表 $\sigma=\{<wm,1>,<wl,0>\}$ 可知,由于 wm 锁变量的值不为 0,存在锁的添加与释放不匹配的情况. 根据对应的锁约束条件 P 可知,其锁不匹配

现象将在 $k \geqslant 3$ 的情况下出现.

6 实验结果与分析

6.1 实验构建

为评估该方法的有效性,本文以 Java 为目标语言,先对 Android 源码社区的 10 个开源 App 随机插入能耗错误,并对 18 个恶意 App 进行反编译获得对应的源程序,然后利用 eclipse 中的 ASTParser 作为 Java 源代码的前端解析器,生成对应的 AST 分析树. 接着以该分析树为基础,构建上节提出的错误检测方案. 最后将分析结果同文献[3]中提出的基于数据流分析的方法进行对比,以获得该方法的最终评估效果. 具体实验框架如图 1 所示.

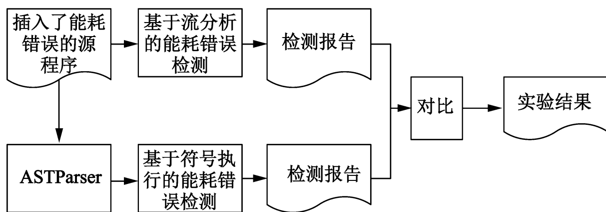

图1 实验方案

6.2 实验结果与分析

在具体实验过程中,本文采用开源社区 AppCodes 的 9 个 Android 源程序为基础,通过在其中随机插入能耗错误构建本文实验的测试用例. 通过与文献[3]中基于数据流错误检测技术的对比,获得如表 6 所示实验结果. 根据前面所述的能耗模型,编译器在能耗错误检测过程的绿色指标主要与错误检测的精度(包括检测的总错误数和误检测的错误数)以及给出的错误路径的精度(包括给出出错时对应的正确执行路径数以及错误的执行路径数)两大部分组成,因此在该中主要给出了 4 个指标:其中 *PE* 和 *DE* 分别表示路径型禁止休眠能耗错误和扩展型禁止休眠能耗错误,*MD* 和 *MP* 分别表示误报的错误数和误报的路径数. 由于竞争型能耗错误通常通过人为指定事件触发顺序,然后进行检测. 而在知晓了事件触发顺序后,则可以通过该触发顺序很直接的按照前两类能耗错误检测方法进行检测,因而本文未对其进行分析验证.

表 6 能耗错误检测实验结果

用例名	植入错误数		数据流分析				符号执行技术			
	PE	*DE*	*PE*	*DE*	*MD*	*MP*	*PE*	*DE*	*MD*	*MP*
Amazed	1	0	1	0	0	0	1	0	0	0
AndroidGlobalTime	3	0	2	0	0	0	3	0	0	0
HeightMapProfiler	3	0	1	0	0	0	3	0	0	0
Photostream	2	2	1	0	0	0	2	2	0	0
Radar	1	0	1	0	0	0	1	0	0	0
RingsExtended	3	3	0	3	0	0	3	3	0	0
Flashlight	1	1	1	0	0	0	1	1	0	0
Calculator	2	2	1	0	0	0	2	2	0	0
WordPress	3	2	1	0	0	0	3	2	0	0

接着,本文对现有的 18 个恶意 App 进行反编译,获得其对应的源程序,然后通过与文献[3]中基于数据流错误检测技术的对比,获得如表 7 所示实验结果.

表 7 能耗错误检测试验结果 2

用例名	数据流分析检测错误数		符号执行技术检测错误数		用例名	数据流分析检测错误数		符号执行技术检测错误数	
	检测数	误报数	检测数	误报数		检测数	误报数	检测数	误报数
车友信	6	0	12	0	美白相机	13	0	14	0
陈赫奔跑吧兄弟	8	0	14	0	疯狂猜灯谜	11	0	17	0
文件管理器	9	0	13	0	吉日黄历	3	0	11	0
驾考科目一	3	0	8	0	圣经故事	4	0	9	0
韩语学习大全	5	0	7	0	懒人听英语	6	0	16	0
捕鱼达人 3	11	0	15	0	开心猜猜看	10	0	19	0
反正	14	0	19	0	儿童宝宝抓玩具	13	0	21	0
一键 ROOT	15	0	21	0	爱查	9	0	10	0
彩铃声声	18	0	25	0	消灭星星 2015	2	0	8	0

由以上实验结果可以看出,基于数据流分析的能耗错误检测方法与符号执行技术能耗错误检测方法均有较高的检测准确度,对于植入的能耗错误,其误报率均为 0. 但基于数据分析的能耗错误主要针对于不计数性的锁变量进行检测,且未考虑检测扩展型禁止休眠能耗错误,因此对于该类错误无能为力,因而其总体错误检测率明显低于本文方法.

通过对表 7 中 18 个测试用例检测的能耗错误进行修正,然后采用红米 note 手机实测半小时,获得的能耗

对比结果如表 8 所示.

表 8　能耗对比结果

用例名	数据流分析		符号执行技术检测错误数		用例名	数据流分析检测错误数		符号执行技术检测错误数	
	纠错前耗电量	纠错后耗电量	纠错前耗电量	纠错后耗电量		纠错前耗电量	纠错后耗电量	纠错前耗电量	纠错后耗电量
车友信	1.0%	0.6%	1.0%	0.3%	美白相机	1.3%	0.9%	1.3%	0.6%
陈赫奔跑吧兄弟	1.5%	1.0%	1.5%	0.4%	疯狂猜灯谜	1.1%	1.0%	1.1%	0.6%
文件管理器	1.9%	1.2%	1.9%	0.3%	吉日黄历	0.9%	0.7%	0.9%	0.5%
驾考科目一	1.2%	0.7%	1.2%	0.2%	圣经故事	0.5%	0.3%	0.5%	0.2%
韩语学习大全	0.9%	0.3%	0.9%	0.2%	懒人听英语	0.8%	0.6%	0.8%	0.7%
捕鱼达人3	1.1%	0.8%	1.1%	0.3%	开心猜猜看	1.7%	1.2%	1.7%	0.9%
反正	1.7%	1.2%	1.7%	0.5%	儿童宝宝抓玩具	1.2%	0.9%	1.2%	0.5%
一键 ROOT	1.9%	1.5%	1.9%	0.6%	爱查	0.9%	0.7%	0.9%	0.4%
彩铃声声	2.0%	1.8%	2.0%	0.9%	消灭星星2015	2.2%	1.5%	2.2%	0.8%

由以上能耗结果可以看出,修正能耗错误后,采用数据流分析的方法平均能够节约 29% 的能耗,而本文的分析方法能够进一步提升能耗的效率,平均能够节约 62.6% 的能耗,这对于依靠容量有限的电池作为电源的智能移动终端设备将是十分重要的.

7　总结

能耗是绿色需求中一个重要指标,能耗错误检测对提高系统的绿色程度具有不容忽视的作用. 本文首先对能耗错误的分类以及符号执行技术进行简要分析,然后针对禁止休眠类能耗错误,以符号执行技术为基础,设计了能耗错误检测和定位方法. 该方法首先利用过程内分析,获得单个函数的符号执行信息,然后利用过程间分析对单个函数的符号执行信息进行全局分析,以获得较为准确的执行开销、锁变量匹配等相关信息,检测出对应的能耗相关错误. 同时,符号执行记录了对应的分支路径信息,利用该信息不但较好的生成对应的测试用例,而且可以结合约束求解器快速定位错误位置,为开发出高绿色指标的软件提供保障.

参考文献

[1] Pathak A,Hu Y C,Zhang M. Bootstrapping energy debugging on smartphones:a first look at energy bugs in mobile devices[A]. Proceedings of the 10th ACM Workshop on Hot Topics in Networks[C]. Cambridge, USA: ACM; 2011. 5 – 10.

[2] Pathak A,Hu Y C,Zhang M. Where is the energy spent inside my app?:fine grained energy accounting on smartphones with eprof[A]. Proceedings of the 7th ACM european conference on Computer Systems[C]. Bern, Switzerland: ACM;2012. 29 – 42.

[3] Pathak A, Jindal A, Hu Y C, et al. What is keeping my phone awake:characterizing and detecting no-sleep energy bugs in smartphone apps[A]. Proceedings of the 10th international conference on Mobile systems, applications, and services[C]. Ambleside, United Kingdom: ACM; 2012. 267 – 280.

[4] Vekris P, Jhala R, Lerner S, et al. Towards verifying android apps for the absence of no-sleep energy bugs[A]. Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems[C]. New York, USA: USENIX Association;2012. 3 – 13.

[5] Oliner A J, Iyer A, Lagerspetz E, et al. Collaborative energy debugging for mobile devices[A]. Proceedings of the Eighth USENIX conference on Hot Topics in System Dependability[C]. Berkeley, CA, USA: USENIX Association;2012. 6 – 6.

[6] Jindal A, Pathak A, Hu Y C, et al. Hypnos: understanding and treating sleep conflicts in smartphones[A]. Proceedings of the 8th ACM European Conference on Computer Systems[C]. Prague, Czech Republic: ACM; 2013. 253 – 266.

[7] King J C. Symbolic execution and program testing[J]. Communications of the ACM, 1976, 19(7):385 – 394.

[8] Boyer R S, Elspas B, Levitt K N. SELECT-a formal system for testing and debugging programs by symbolic execution[J]. Acm Sigplan Notices, 1975, 10(6):234 – 245.

[9] Khurshid S, Pϖsϖreanu C S, Visser W. Generalized Symbolic Execution for Model Checking and Testing[M]. Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2003. 553 – 568.

[10] P Godefroid, N Klarlund, K Sen. DART: Directed Automated Random Testing[A]. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation[C]. Chicago, USA;2005. 213 – 223.

[11] Majumdar R, Sen K. Hybrid concolic testing[A]. 29th International Conference on Software Engineering[C]. Minneapolis, USA: IEEE;2007. 416 – 426.

[12] Burnim J, Sen K. Heuristics for Scalable Dynamic Test Generation[J]. IEEE Computer Society, 2008(8): 443 – 446.

[13] Anand S, Godefroid P, Tillmann N. Demand-Driven Compositional Symbolic Execution[M]. Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2008. 367 – 381.

[14] Staats M, Păsăreanu C. Parallel symbolic execution for structural test generation[A]. Proceedings of the 19th international symposium on Software testing and analysis [C]. Trento, Italy: ACM; 2010. 183 – 194.

[15] Anand S, Pϖsϖreanu C S, Visser W. JPF-SE: A Symbolic Execution Extension to Java Pathfinder[M]. Springer Berlin Heidelberg, Tools and Algorithms for the Construction and Analysis of Systems, 2007: 134 – 138.

[16] Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs[A]. OSDI[C]. Berkeley, USA; 2008. 209 – 224.

[17] Cadar C, Ganesh V, Pawlowski P M, et al. EXE: automatically generating inputs of death[J]. ACM Transactions on Information and System Security (TISSEC), 2008, 12(2): 10 – 20.

[18] Sen K, Marinov D, Agha G. CUTE: A Concolic Unit Testing Engine for C[M]. ACM, 2005. 50 – 60.

[19] 梅宏,王千祥,张路,等. 软件分析技术进展[J]. 计算机学报,2009,32(9):1697 – 1710.
MEI Hong, WANG Qian-Xiang, ZHANG Lu, et al. Software Analysis: A road map[J]. Chinese Journal of Computers, 2009, 32(9): 1697 – 1710. (in Chinese)

[20] 范文庆. 分段符号执行模型及其环境交互问题研究[D]. 北京:北京邮电大学,2010. 40 – 50.
Fan Wenqing. Research on the implementation model of the sub symbolic execution model and its environment interaction[D]. Beijing: Beijing University of Posts and Telecommunications, 2010. 40 – 50. (in Chinese)

[21] Boonstoppel P, Cadar C, Engler D. RWset: Attacking Path Explosion in Constraint-Based Test Generation[M]. Springer Berlin Heidelberg, Tools and Algorithms for the Construction and Analysis of Systems, 2008. 351 – 366.

[22] Schwartz E J, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask) [A]. 2010 IEEE Symposium on Security and Privacy [C]. Oakland, California; 2010. 317 – 331.

[23] 林梦香,陈胤立,陈睿,等. 基于懒替换的 C 符号执行[J]. 北京航空航天大学学报,2009,06(6):687 – 691.
Lin Mengxiang, Chen Yinli, Chen Rui, et al. Execution of C symbols based on lazy replacement[J]. Journal of BeiJing University of Aeronautics and Astronautics, 2009, 06(6): 687 – 691. (in Chinese)

作者简介

徐 超 男,1980 年出生,湖北红安人,博士,副教授,研究方向为可信软件、嵌入式系统和计算机审计.
E-mail: xuchao@nau.edu.cn

陈 勇(通信作者) 男,1986 年出生,湖南娄底人,博士,工程师,研究方向为主要研究领域为编译优化,可信软件,嵌入式系统优化.
E-mail: cyong1000@163.com