为了能够快速探索待测程序状态空间,现有工作通常统计执行不同测试用例所能覆盖的基本块,然后与当前已被覆盖的基本块比较,计算新增基本块数量来反映测试用例对基本块覆盖的贡献,借此评定测试用例被调度的优先级.以图 4 所示未被覆盖的代码片段为例,假设翻转第 1 行、第 3 行等条件语句后求解生成的新用例分别为 $t_0,t_1,\ldots,t_n$.记 $score(t)$为采用测试用例 t 进行测试时所覆盖的新增基本块数量,有 $score(t_i)=a_i+m$,其中,$0\leqslant i\leqslant n$.假设 $a_0>a_1>\ldots>a_n$,则 t_0 将被优先调度进行路径约束的收集和新测试用例的生成.假定新一轮中,以 t_0 为输入,所生成的新测试用例集合为 *newset*.随着 t_0 的执行,测试用例 $t_1,\ldots,t_n$ 所能覆盖的新增基本块数量更新为 $a_1,\ldots,a_n$,它们原有的分数 $score(t_i)=a_i+m$,其中,$1\leqslant i\leqslant n$,将不再反映其对基本块覆盖的贡献.在计算用例对基本块覆盖的贡献时,若评估算法不仅运行 *newset* 内的新测试用例,而且重复运行当前工作列表内所有的备选用例(如 $t_1,\ldots,t_n$ 等),更新每个用例所能产生的新增基本块[9],则称其为完全覆盖评估方法(total_eval).完全覆盖评估能够精确地计算每个用例所能产生的新增基本块,但是评估过程会消耗大量资源.相对而言,若算法在评估用例时只针对集合 *newset* 内的新用例,统计执行新用例时所覆盖的新增基本块数量[7],则称其为部分覆盖评估方法(part_eval).部分覆盖评估方法能够有效减少评估用例过程所需的资源,但是由于部分覆盖评估方法在评估过程中只关注新生成的用例,所以该方法的精确度不高,且容易受到路径发散问题(divergence)[3,5]的影响.路径发散是指动态符号执行系统预计测试用例 t_i 可以指导程序抵达条件分支指令 c_i,并向预订方向迁移.然而,由于实际运行过程的一些不确定因素(如多线程竞争、浮点数对路径约束收集的影响等),使得执行路径无法抵达 c_i 或无法向预订方向迁移.路径发散现象普遍存在于动态符号执行中,例如,SAGE 生成的测试用例发生路径发散的几率高达 60%[3].我们在实验时也发现,很多由不同路径约束求解所得到的测试用例实际上却执行同一条路径.在本例中,假设测试用例 t_0 与 t_k ($0<k<n$)由于路径发散执行同一路径,随着 t_0 被调度,$score(t_k)$虽然很高,但不能覆盖任何新基本块.当 t_k 被调度时,测试资源浪费在已被执行的路径上,从而降低了部分覆盖评估方法的效率.

```
1. if (in==c0)
2.    a0 new blocks
3. else if (in==c1)
4.    a1 new blocks
5.    ...
6. else if (in==cn)
7.    an new blocks
8. m new blocks
```

Fig.4 Example code of test case evaluation

图 4 测试用例评估示例代码

我们在计算不同测试用例对基本块覆盖贡献时,引入了对执行效率和路径发散问题的考量:首先,每轮测试用例评估时,只计算集合 *newset* 内新测试用例的权重,以减少评估过程的资源消耗;其次,维护一个基本块首地址的集合 H_b,在对新用例进行评估时,该集合累计更新运行不同用例所覆盖的基本块.具体来说,对于任意新测试用例 t,令 $cover(t)$为采用 t 进行测试时可以覆盖的基本块集合,首先计算用例 t 能产生的新增基本块数量为 $score(t)=|cover(t)-H_b|$,然后更新集合 $H_b=H_b\cup cover(t)$.以上述代码片段为例,对于首先被评估的 t_0,有 $score(t_0)=a_0+m$,$H_b=H_b\cup cover(t_0)$;后续被评估的用例分数变化为 $score(t_i)=a_i$,$H_b=H_b\cup score(t_i)$,其中,$1\leqslant i\leqslant n$.采用这种计算方法,由于路径发散而执行重复路径的 t_k 可以被正确地鉴别,因为有 $score(t_k)=0$.由第 3.2.3 节的实验可知,虽然此评估方法提高了前期被评估的用例的优先级,相对于完全覆盖评估方法损失了部分精确度,导致测试任务中用例调度次数的增加,但该方法可有效地减少用例评估过程所需时间;同时,相对于部分覆盖评估,该评估方法可消除由路径发散引入的冗余用例,从而达到加速探索待测程序的目的.

危险路径定位的流程如算法 4 所示.定位算法以危险路径遍历过程中被覆盖的基本块为起点,更新当前已被覆盖的基本块列表 *lBlocksCovered*(第 1 行).然后选择备选测试用例列表 *lWorklist* 前端的用例作为输入,按照动态符号执行流程生成新测试用例(第 8 行~第 11 行).对于每个新测试用例,算法按照上文描述的评估方法计算它们对基本块覆盖的贡献(第 12 行).新用例将按照贡献值依次进行调度.对于贡献为 0 的测试用例,为了能够快速地定位新危险操作和相关路径,定位算法在调度过程中选择忽略这部分用例(第 15 行、第 16 行).需要指出的是,虽然相关工作[3,7]实验结果表明,运行这类用例不仅会消耗大量测试时间,而且对缺陷的触发几乎没有贡献,但在极端情况下,这种忽略特定测试用例的做法可能会导致漏报.算法运行过程中,需要中断危险路径定位过程,转而进行危险路径遍历的情形有如下两种:

(1) 危险路径定位过程执行到新的危险操作,需要调用遍历过程对该操作相关路径进行遍历检查.对此,

危险路径定位算法针对被调度的用例实施污点分析,更新危险操作和可影响危险操作的字节集合.如果集合 D_{op} 内发现新的危险操作,或者 *DByte*(*dop*)内存在可以影响危险操作的新的字节,则将当前用例加入危险路径遍历的测试用例列表 *eWorkList*(第 3 行~第 6 行),然后将控制流程转交给遍历算法(第 17 行).

(2) 危险路径定位过程以新的上下文环境执行已知危险操作,需要分析当前执行路径中是否引入了危险操作相关约束项新的组合.如果有,则调用遍历过程对这些组合进行检查.对此,算法首先分析新用例是否由翻转危险操作相关约束项然后求解所生成:若是,则分析该用例是否可以覆盖新的基本块,借此来判断是否引入未被检测的危险约束项组合.

如果两者均满足,则将当前测试用例添加到列表 *eWorkList*(第 13 行、第 14 行),流程转至遍历算法(第 17 行).

算法 4. 危险路径定位算法.

输入:*lWorklist*; *eBlocksCovered*.

```
1.  lBlocksCovered+=eBlocksCovered;                    //更新当前已被覆盖基本块列表
2.  while (lWorklist!=∅) {
3.    input=PickFirstItem(lWorklist);
4.    (D_op,DByte(dop))+=TaintAnalysis(T_op,input);
5.    if new D_op or DByte(dop) detected {             //如 input 执行新危险操作,则转至遍历算法
6.      eWorkList+=newInput; break;}
7.    lBlocksCovered+=TraceBlocks(newInput);
8.    pc=PCCollection(input);
9.    for (i=input.lBound; i<|pc|; i++) {
10.     if (pc[0...(i~1)] and not(pc[i])) has a solution I {
11.       newInput=SeedInput+I; newInput.lBound=i;
12.       newInput.lScore=lScore();                    //计算新测试用例对基本块覆盖的贡献
13.       if ((SByte(dop)∩Use(pc_i)!=∅) && newInput.lScore!=0)   //若为未检测相关约束,则转至遍历
14.         eWorkList+=newInput;
15.       else if newInput.lScore!=0
16.         lWorkList+=newInput}}
17.   if eWorkList!=∅ break;}
```

3 实现和评估

3.1 原型工具实现

为了评估面向危险操作的动态符号执行方法的有效性和效率,实现了原型工具 CrashFinder,包含初始输入选择模块和动态符号执行模块两部分.该工具基于动态插装平台 Valgrind[30],将程序执行中的二进制指令翻译为中间语言 VEX.在此基础上,实现部分控制流敏感的细粒度污点分析,识别危险操作和可影响危险操作的输入字节,为初始输入选择模块提供支持.动态符号执行模块同样分析被执行指令的 VEX 语言,维护程序执行的真实上下文与符号上下文环境,将执行路径中与符号化污点数据相关的指令转换成对输入数据的约束条件.CrashFinder 采用 STP[31]约束求解器对翻转的路径约束进行求解,构造新的测试用例.

为了方便与相关工作比较,CrashFinder 的目标缺陷设置为内存读写错误、整型溢出与除数零错误.相应的关注安全敏感操作集合 T_{op} 包含内存读写操作、内存分配函数(*malloc* 等)以及除操作.为了在测试过程中通过符号化的方式有针对性地检测错误,需要对关注的缺陷定义相应的缺陷模型.表 1 为上述危险操作对应的缺陷触发条件,其中,*base* 代表缓冲区基地址,*len*(*buf*)为缓冲区长度.这些信息是由 CrashFinder 通过插装内存分配函数以及分析函数调用栈获得.当待测程序执行至危险操作时,CrashFinder 在已收集的路径约束基础上,按照表 1

规则添加缺陷触发约束并进行求解.若可解,则生成满足危险语句执行条件和缺陷触发条件的测试用例.CrashFinder 监控所有新测试用例的执行,如果产生关注缺陷类型相关的信号中断,则定位并输出相关数据以辅助程序员重现缺陷.

Table 1 Constraints of defects

表 1 缺陷触发条件

危险操作类型	缺陷触发条件
r=dividend/divisor	*divisor*==0
**p*	*p*==*Null*; *p*<*base*∨*p*>*base*+*len*(*buf*)
malloc(*arg*)等内存分配函数	*arg*>max(*int*) (32 位系统为 0xffffffff)

3.2 性能评估

我们选择一系列开源软件,对 CrashFinder 的初始输入评分算法的有效性、面向危险操作的路径选择策略的有效性和危险路径定位算法中测试用例评估方法的效率等方面进行测评.实验运行的环境为:CPU Intel Core i5-750,主频 2.67GHz,内存 4G,系统 Unbuntu12.10.实验中如没有特殊要求,每个测试任务均被赋予足够的运行时间,直至工作列表中没有能覆盖新基本块的测试用例存在.为了分辨不同用例是否触发同一缺陷,对于测试过程中生成的可触发缺陷或致使待测程序崩溃的测试用例,均通过人工分析来确定缺陷根源.

3.2.1 初始输入评分算法的有效性

为了验证 CrashFinder 初始输入评分算法的有效性,我们选择一组基准程序,并为每个程序提供一组备选初始输入.首先利用 CrashFinder 的初始输入选择模块对各个备选输入进行评分,然后选取各个分数段内的备选输入作为动态符号执行模块的初始输入,对基准程序进行测试,并记录每组实验结果,用来验证不同备选初始输入的评估得分与它们的缺陷检测表现之间的联系.实验采用的基准程序一部分选自相关工作中的开源软件,包括文献[8]中的 readelf 2.23.52、文献[9]中 libjpeg7 软件包内 cjpeg 程序与 swftool 0.9.0 以及文献[23]中的 nginx 0.6.32.这些基准程序被研究人员用于评价各自的方法,具有一定的代表性.基准程序还包括 Linux 平台常用软件 ImageMagic 6.7.7-10 软件包内的 convert 程序.针对每个基准程序,实验所使用的备选初始输入集合中,包含相关工作实验所使用的初始输入文件以及程序自带测试用例集合中部分合法格式输入.需要说明的是,由于实现平台限制,实验中 nginx 经过修改,其接收数据方式为从文件读入.

实验结果见表 2:第 1 列和第 2 列分别描述了基准程序名称和合法输入的格式;第 3 列和第 4 列描述了备选初始输入的编号和大小;第 5 列和第 6 列描述了初始输入选择模块的分析时间和评分结果;第 7 列~第 10 列分别描述了使用备选输入测试基准程序初始所覆盖的基本块数量、完成测试所用时间、产生的测试用例数量和发现的缺陷数.

Table 2 Effectiveness of initial input scoring algorithm in CrashFinder

表 2 CrashFinder 初始输入评分算法的有效性

Program	Format	备选初始输入							
		ID	Size	EvalTime (s)	Scores	BlocksAtStart	TestTime (s)	TestCases	Defects
cjpeg	BMP	ABmp	712Bytes	65	5 720	3 073	12 243	1 321	1
		Bmp1	10KB	121	10 720	3 293	16 224	1 732	3
		Bmp2	9.05KB	145	15 112	3 334	15 235	1 654	3
nginx	HTTP	Http1	4Bytes (uri)	29	4 529	5 423	7 492	1 832	0
		Http2	10Bytes (uri)	36	6 744	5 346	7 982	1 932	1
		Http3	20Bytes (uri)	39	9 843	5 543	8 321	2 324	1
readelf	ELF	CElf	2.93KB	17	9 023	4 386	3 492	3 596	0
		Elf1	56.6KB	42	12 031	4 425	4 735	4 214	3
		Elf2	29.4KB	32	15 202	4 391	4 243	4 712	3
convert	JPG	Jpg1	49KB	132	6 953	13 834	4 134	376	1
		Jpg2	45.0KB	131	13 812	13 390	4 120	335	1
		Jpg3	69KB	143	15 224	13 206	4 430	415	1
swftool	SWF	ASwf	712Bytes	20	934	2 770	9 620	2 987	2
		Swf1	3.46KB	32	1 120	3 427	13 849	3 408	2
		Swf2	11.2KB	45	2 231	3 424	15 393	4 126	2

1) 由 EvalTime (s)列可知,对备选初始输入进行评分所引入的时间开销相对较小,实验中,评分工作多在一分钟左右即可完成.
2) 由 Defects 列和 Scores 列可知,对于每个基准程序,评分高的备选初始输入触发的缺陷数量不低于评分低的备选初始输入.也就是说,我们的初始输入评分算法能够很好地刻画备选初始输入的缺陷发掘能力.例如,在对 cjpeg 进行动态符号执行测试时,选自文献[9]的输入文件 ABmp 与其他备选初始输入均能触发位于 jmemmgr.c:406 的除数零错误.然而出于测试开销考量,文献[9]删除了 *ABmp* 中大量图片内容相关信息而仅保留满足 BMP 格式要求的数据.因此,使用 *ABmp* 为初始输入进行的动态符号执行过程中,由于图片内容缺失,测试流程不能覆盖与图片内容相关的内存分配函数,因此不能触发该函数由于参数发生整型溢出而引起的内存访问越界缺陷.相对而言,在采用高评分初始输入 *Bmp*2 后,我们在 cjpeg 中额外发现了两个内存访问越界错误(rdbmp.c:212,jmemmgr.c:428).类似地,初始输入选择模块不仅可以对 nginx 的备选初始输入按照其 *uri* 长度进行排序评分,而且相对于文献[8]采用文件 *CElf* 为初始输入测试 readelf 时没有发现缺陷,我们采用高评分的初始输入 *Elf*2 在 readelf 中发现了 3 个内存读写错误.
3) 对比 Size 列、BlocksAtStart 列以及 Scores 列可见,部分实例中(例如 *Elf*1 与 *Elf*2、*Bmp*1 与 *Bmp*2),备选初始输入的缺陷触发能力与其文件大小或初始所能覆盖的基本块数量并没有直接联系.即相对于传统黑盒测试中普遍采用的通过简单计算备选输入文件大小或者它们所覆盖的基本块数量来选定初始输入的方法[32],初始输入评分算法在协助动态符号执行技术选择初始输入方面更加精确.

3.2.2 面向危险操作的路径选择策略的有效性

Avalanche[9]为 Linux 平台针对内存访问错误,空指针解引用以及除数零错误进行检测的动态符号执行工具,它依据新用例产生的新增基本块数量对其进行调度.现将 CrashFinder 与 Avalanche 的缺陷检测能力进行比较,用以反映面向危险操作的路径选择策略和传统基于新增基本块的路径选择策略的缺陷检测能力的差异.为了方便比较,这里选取 Avalanche 工作实验中采用的 swftool 0.9.0,libjpeg7 软件包内的 cjpeg 程序、libquicktime-1.1.2 内的 qtdump 程序、libmpeg3-1.8 内的 mpeg3dump 程序以及 speex-1.2rc 作为分析对象.

表 3 为采用相同初始输入分别运行 CrashFinder 和 Avalanche 的实验结果.第 1 列为测试对象,第 2 列~第 5 列和第 7 列~第 10 列分别描述两个工具产生的测试用例数、缺陷检测数量、触发第 1 个缺陷所用时间以及触发所有不同缺陷所用时间.Switch 列为 CrashFinder 测试过程中危险路径遍历与危险路径定位两个阶段之间的切换次数.

Table 3 Comparison of defect detection ability between CrashFinder and Avalanche

表 3 CrashFinder 与 Avalanche 缺陷检测能力比较

Program	CrashFinder					Avalanche			
	TestCases	Defects	T_{first} (s)	T_{total} (s)	Switch	TestCases	Defects	T_{first} (s)	T_{total} (s)
qtdump	230	1	124	124	1	172	1	369	369
speexenc	76	1	293	293	1	44	1	454	454
swftool	3 408	2	180	408	2	3 278	2	616	871
mpeg3dump	23 569	2	2	66	4	20 433	2	2	135
cjpeg	1 654	3	214	434	2	1 365	3	211	723

1) 对比两个工具的 Defects 列和 T_{total}(s)列可知,在保证发现所有缺陷的前提下,CrashFinder 比 Avalanche 花费更少的时间,这反映了面向危险操作的路径选择策略在加速触发缺陷方面的优势.
2) 对比两个工具的 T_{first}(s)列可知,对于多数程序,CrashFinder 比 Avalanche 能够更快地触发第 1 个缺陷.两个例外情况是,在 cjpeg 与 mpeg3dump 内分别存在一个除数零错误(jmemmgr.c:406)与空指针解引用错误(mpeg3ifo.c:509).由于两个工具均能结合初始输入产生的路径约束与上文缺陷触发模型中对应的缺陷触发条件,然后借助约束求解器直接生成可以触发这两个缺陷的测试用例,因此这两例程序对应的 T_{first} (s)相差不大.
3) 由 Switch 列可知,对于多数程序,仅依据初始输入识别的危险操作以及相关路径信息,并不能覆盖待

测程序内所有危险操作.因此,相对于传统的结合细粒度污点分析与危险操作的测试方法[21−23],面向危险操作的路径选择策略可以对待测程序内危险操作进行更全面的检测.

3.2.3 危险路径定位算法的效率

如第 2.4.2 节所描述,危险路径定位算法中,测试用例评估方法在完全覆盖评估方法的注重精度而消耗资源与部分覆盖评估方法的节约资源而牺牲精度之间取折中,加速动态符号执行对待测程序状态空间的探索和对新危险操作的定位.表 4 描述了分别采用危险路径定位算法中测试用例评估方法、部分覆盖评估方法与完全覆盖评估方法来评估和调度测试用例,对待测程序进行动态符号执行的实验结果.其中,第 1 列描述了待测程序名称,第 2 列~第 5 列、第 6 列~第 9 列和第 10 列~第 13 列分别详细描述了使用 3 种评估调度策略的测试时间、测试过程中不同阶段所占时间百分比(测试用例评估/路径约束收集/约束求解以及错误检查)、基本块覆盖总数以及测试用例数.

Table 4 Efficiency of critical paths locating method

表 4 危险路径定位算法的效率

Program	危险路径定位中测试用例评估方法				部分覆盖评估方法				完全覆盖评估方法			
	Total	PCT	Blocks	Case	Total	PCT	Blocks	Case	Total	PCT	Blocks	Case
qtdump	3 552	6/53/41	6 698	211	5 029	5/50/45	6 678	337	4 427	11/48/41	6 687	162
speex	2 048	2/3/95	4 487	44	2 533	1/6/93	4 432	74	2 171	8/3/89	4 468	44
swftool	12 943	21/30/49	4 329	3 415	14 082	18/42/40	4 301	4 608	13 786	46/41/13	4 231	3 158
mpeg3	32 837	30/2/68	5 138	25 573	38 340	24/4/72	5 086	29 365	37 403	54/1/45	5 132	19 033
cjpeg	13 223	2/22/76	3 795	1 475	15 618	2/20/78	3 789	2 004	14 216	7/21/72	3 839	1 252
readelf	4 082	18/9/73	5 165	4 023	5 177	18/6/76	4 993	5 087	5 766	38/2/60	4 884	2 563
convert	3 930	8/37/55	17 997	287	4 477	6/48/46	18 019	632	4 394	19/32/49	17 973	156

1) 由不同策略的 Total 列可得,相对于其他两种评估方法,危险路径定位算法中的测试用例评估方法可以更快地完成对待测程序的测试任务.
2) 对比不同策略的 PCT 列和 Case 列可得,相对于部分覆盖评估方法,危险路径定位中测试用例评估方法可以剪除大量冗余测试用例;相对于完全覆盖评估方法,该评估方法虽然由于损失部分精度,导致完成测试所需测试用例数量增加,但会大幅减少评估测试用例的过程所消耗的时间比重,从而加速对待测程序的探索.

3.2.4 缺陷统计

如表 5 所示,CrashFinder 在上述实验程序以及常见开源软件中总共发现了内存访问越界、空指针解引用以及除数零错误这 3 种类型总共 20 个缺陷.正如第 3.2.1 节所分析,相对于其他针对同样程序进行实验的相关动态符号执行工作,CrashFinder 利用初始输入选择模块评估和选择高质量的初始输入,因此在 cjpeg,readelf 以及 convert 等程序中发现了更多缺陷.经过人工核查后,其中包含 10 个未知缺陷,我们已将相关信息以及对应测试用例发送给相关软件作者.

Table 5 Defects detected by CrashFinder

表 5 CrashFinder 发现的缺陷

Programs	cjpeg	readelf	convert	swftool	swftool	qtdump	speexenc	mpeg3dump	mupdf
Version	7	2.23.52	6.7.7	0.9.0	0.9.2	1.1.2	1.2rc	1.8	0.1
#Defects	3	3	1	2	5	1	1	2	2

4 总 结

提出了面向危险操作的动态符号执行方法,基于缺陷发生与危险操作密切相关的思想,分析和识别危险操作以及与危险操作相关的路径约束项.一方面将这些信息用于协助动态符号执行选择高效的初始输入;另一方面用于指导动态符号执行测试用例的生成和调度,对待测程序中易发生缺陷的路径进行优先系统地探测.对原型系统 CrashFinder 的实验结果表明,面向危险操作的动态符号执行方法能够快速、有效地检测更多的缺陷.

References:

[1] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. In: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). New York: ACM Press, 2005. 213−223. [doi: 10.1145/1065010.1065036]

[2] Kim Y, Kim Y, Kim T, Lee G, Jang Y, Kim M. Automated unit testing of large industrial embedded software using concnlic testing. In: Proc. of the 28th ACM/IEEE Int'l Conf. on Automated Software Engineering (ASE). New York: ACM Press, 2013. 518−528. [doi: 10.1109/ASE.2013.6693109]

[3] Godefroid P, Levin MY, Molnar D. Automated whitebox fuzz testing. In: Proc. of the 2008 Network and Distributed Systems Security (NDSS). San Diego: The Internet Society, 2008. 151−166.

[4] Cadar C, Sen K. Symbolic execution for software testing: three decades later. Communications of the ACM, 2013,56(2):82−90. [doi: 10.1145/2408776.2408795]

[5] Lu XC, Li G, Lu K, Zhang Y. High-Trusted-Software-Oriented automatic testing for integer overflow bugs. Ruan Jian Xue Bao/Journal of Software, 2010,21(2):179−193 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3785.htm [doi: 10.3724/SP.J.1001.2010.03785]

[6] Godefroid P, Levin MY, Molnar DA. Active property checking. In: Proc. of the 8th ACM Int'l Conf. on Embedded Software (EMSOFT). New York: ACM Press, 2008. 207−216. [doi: 10.1145/1450058.1450087]

[7] Molnar D, Li XC, Wagner DA. Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proc. of the 18th USENIX Security Symp. San Diego: USENIX Association, 2009. 67−82.

[8] Chen B, Zeng QK, Wang WG. Crashmaker: An improved binary concolic testing tool for vulnerability detection. In: Proc. of the 29th Annual ACM Symp. on Applied Computing (SAC). New York: ACM Press, 2014. 1257−1263. [doi: 10.1145/2554850.2554875]

[9] Isaev IK, Sidorov DV. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. ACM Journal of Programming and Computing Software, 2010,36(4):225−236 [doi: 10.1134/S0361768810040055]

[10] Chen K, Feng DG, Su PR. Dynamic overflow vulnerability detection method based on finite CSP. Chinese Journal of Computers, 2012,35(5):898−909 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2012.00898]

[11] Majumdar R, Sen K. Hybrid concolic testing. In: Proc. of the 29th Int'l Conf. on Software Engineering (ICSE). Washington: IEEE Computer Society, 2007. 416−426. [doi: 10.1109/ICSE.2007.41]

[12] Bounimova E, Godefroid P, Molnar D. Billions and billions of constraints: Whitebox fuzz testing in production. In: Proc. of the 2013 Int'l Conf. on Software Engineering (ICSE). Piscataway: IEEE Press, 2013. 122−131. [doi: 10.1109/ICSE.2013.6606558]

[13] Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. In: Proc. of the 36th Int'l Conf. on Software Engineering (ICSE). New York: ACM Press, 2014. 435−445. [doi: 10.1145/2568225.2568271]

[14] Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. Exe: Automatically generating inputs of death. In: Proc. of the 13th ACM Conf. on Computer and Communications Security (CCS). New York: ACM Press, 2006. 322−335. [doi: 10.1145/1180405.1180445]

[15] Cadar C, Dunbar D, Engler D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation (OSDI). Berkeley: USENIX Association, 2008. 209−224.

[16] Marinescu PD, Cadar C. Make test-zesti: A symbolic execution solution for improving regression testing. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). Piscataway: IEEE Press, 2012. 716−726. [doi: 10.1109/ICSE.2012.6227146]

[17] Burnim J, Sen K. Heuristics for scalable dynamic test generation. In: Proc. of the 23rd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Washington: IEEE Computer Society, 2008. 443−446. [doi: 10.1109/ASE.2008.69]

[18] Cui ZQ, Wang LZ, Li XD. Target-Directed concolic testing. Chinese Journal of Computers, 2011,34(6):953−964 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2011.00953]

[19] Su T, Fu ZL, Pu GG, He JF, Su ZD. Combining symbolic execution and model checking for data flow testing. In: Proc. of the 2015 Int'l Conf. on Software Engineering (ICSE). Piscataway: IEEE Press, 2015. 654−665. [doi: 10.1109/ICSE.2015.81]

[20] Babić D, Martignoni L, McCamant S, Song D. Statically-Directed dynamic automated test generation. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis (ISSTA). New York: ACM Press, 2011. 12−22. [doi: 10.1145/2001420.2001423]

[21] Ganesh V, Leek T, Rinard M. Taint-Based directed whitebox fuzzing. In: Proc. of the 31st Int'l Conf. on Software Engineering (ICSE). Washington: IEEE Computer Society, 2009. 474−484. [doi: 10.1109/ICSE.2009.5070546]

[22] Wang TL, Wei T, Gu GF, Zou W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Proc. of the 2010 IEEE Symp. on Security and Privacy (S&P). Washington: IEEE Computer Society, 2010. 497−512. [doi: 10.1109/SP.2010.37]

[23] Haller I, Slowinska A, Neugschwandtner M, Bos H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: Proc. of the 22nd USENIX Conf. on Security (SEC). Berkeley: USENIX Association, 2013. 49−64.

[24] CVE statistics for integer vulnerabilities. http://www.cve.mitre.org/cgi-bin/cvekey.cgi?keyword=integer

[25] Wang WG, Zeng QK. Evaluating initial inputs for concolic testing. In: Proc. of the 2015 Int'l Symp. on Theoretical Aspects of Software Engineering (TASE). Piscataway: IEEE Computer Society, 2015. 47−54. [doi: 10.1109/TASE.2015.14]

[26] Chen S, Kalbarczyk Z, Xu J, Iyer RK. A data-driven finite state machine model for analyzing security vulnerabilities. In: Proc. of the 2003 IEEE Int'l Conf. on Dependable Systems and Networks (DSN). Piscataway: IEEE Computer Society, 2003. 605−614. [doi: 10.1109/DSN.2003.1209970]

[27] Bao T, Zheng YH, Lin ZQ, Zhang XY, Xu DY. Strict control dependence and its effect on dynamic information flow analyses. In: Proc. of the 19th Int'l Symp. on Software Testing and Analysis (ISSTA). New York: ACM Press, 2010. 13−24. [doi: 10.1145/1831708.1831711]

[28] CVE-2009-2629: Buffer underflow vulnerability in nginx. 2009. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2629

[29] Marinescu PD, Cadar C. KATCH: High-Coverage testing of software patches. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). New York: ACM Press, 2013. 235−245. [doi: 10.1145/2491411.2491438]

[30] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). New York: ACM Press, 2007. 89−100. [doi: 10.1145/1250734.1250746]

[31] Ganesh V, Dill DL. A decision procedure for bit-vectors and arrays. In: Proc. of the 19th Int'l Conf. on Computer Aided Verification (CAV). Berlin: Springer-Verlag, 2007. 519−531. [doi: 10.1007/978-3-540-73368-3_52]

[32] Rebert A, Cha SK, Avgerinos T, Foote J, Warren D, Grieco G, Brumley D. Optimizing seed selection for fuzzing. In: Proc. of the 23rd USENIX Conf. on Security Symposium (SEC). Berkeley: USENIX Association, 2014. 861−875.

附中文参考文献:

[5] 卢锡城,李根,卢凯,张英.面向高可信软件的整数溢出错误的自动化测试.软件学报,2010,21(2):179−193. http://www.jos.org.cn/1000-9825/3785.htm [doi: 10.3724/SP.J.1001.2010.03785]

[10] 陈恺,冯登国,苏璞睿.基于有限约束满足问题的溢出漏洞动态检测方法.计算机学报,2012,35(5):898−909. [doi: 10.3724/SP.J.1016.2012.00898]

[18] 崔展齐,王林章,李宣东.一种目标制导的混合执行测试方法.计算机学报,2011,34(6):953−964. [doi: 10.3724/SP.J.1016.2011.00953]

王伟光(1984－),男,河北衡水人,博士生,主要研究领域为信息安全,软件安全测试.

孙浩(1987－),男,博士生,主要研究领域为信息安全,程序分析.

曾庆凯(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为信息安全,分布计算.