第46卷第1期 中国科学技术大学学报 Vol. 46, No. 1
2016年1月 JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA Jan. 2016

文章编号:0253-2778(2016)1-0021-07

控制流污点信息导向的符号执行技术研究

黄 晖,陆余良,刘林涛,赵 军

(合肥电子工程学院网络系,安徽 合肥 230037)

摘要:以快速生成能够覆盖可能存在缺陷程序点的测试用例为目标,结合基于生成的Fuzzing技术、静态程序控制流分析、静态污点分析等手段,提出一种导向式动态符号计算方法.通过Fuzzing生成能够到达包含缺陷程序点的函数的测试用例,作为种子输入驱动符号执行快速到达缺陷函数;在缺陷函数内利用静态控制流分析、静态污点分析计算出控制流污点可达程序切片,基于该切片进行朝向缺陷点的多路径动态符号执行.实验验证了方法能够有效减轻符号执行应用中广泛存在的路径爆炸问题,并且能生成触发目标缺陷的测试用例.

关键词:控制流分析;污点分析;导向式符号执行

中图分类号:TP18 **文献标识码**:A doi:10.3969/j.issn.0253-2778.2016.01.004

引用格式:HUANG Hui, LU Yuliang, LIU Lintao, et al. A research on control-flow taint information directed symbolic execution[J]. Journal of University of Science and Technology of China, 2016,46(1):21-27.
黄 晖,陆余良,刘林涛,等. 控制流污点信息导向的符号执行技术研究[J]. 中国科学技术大学学报,2016,46(1):21-27.

A research on control-flow taint information directed symbolic execution

HUANG Hui, LU Yuliang, LIU Lintao, ZHAO Jun

(*Electronic Engineering Institution of PLA, AnHui* 230037, *China*)

Abstract: Aiming at generation of test cases covering the potential vulnerable program points and combining generation base Fuzzing, static control flow analysis and static taint analysis, this paper proposes a directed dynamic symbolic execution method. By Fuzzing the test cases which could reach the function containing the vulnerable program points are generated, leading the symbolic execution fast towards the vulnerable functions along the denoted single path; By making a static control-flow analysis and a static taint analyses in the vulnerable functions, the control flow taint eachable slices are calculated directing the multi-path dynamic symbolic execution towards the desired vulnerable program points. Experiments prove effectiveness of the method in mitigating the path explosion problem common in symbolic execution applications and in generating test cases that trigger target vulnerability.

Key words: control flow analysis; taint analysis; directed symbolic execution

收稿日期:2015-03-29;**修回日期**:2015-04-22

作者简介:黄晖,男,1987年生,博士生,研究方向:信息、安全、程序分析. E-mail:hhui_123@163.com

通讯作者:陆余良,博士/教授. E-mail:luyuliang@ah165.net

0 引言

随着机器计算能力和约束求解能力的不断提升,近年来动态符号执行技术发展迅速.在整数、位向量理论的表达范围内描述程序的操作语义,以关于输入变元约束等价类的形式刻画程序的路径空间,符号执行技术能够实现对目标程序的精确分析、路径空间的高效覆盖,已成为面向大规模预发布的二进制软件缺陷发掘的最佳选择之一[1-3].

符号执行往往关注于目标程序的全部路径空间,然而该空间并不总能被完全覆盖[4].适度的搜索空间约减是使该技术实际可行的关键.尽管当前相关的研究(包括限制循环次数[5-6]、选择式符号执行[7]、符号执行并行化[8]等)在某些场景中被证明为有效,基于符号执行的缺陷发现技术依然不足以在实际应用中为大型程序的安全性分析提供支撑.

缺陷挖掘过程中常常关注于这样的应用场景:给定缺陷程序点θ,求解能到达θ并触发目标缺陷的输入实例.在这一情况下,分析引擎可以不关注于完整的路径空间,仅仅针对那些能够到达θ的路径集合进行安全分析.

由此,本文提出一种结合基于生成的Fuzzing测试[9]、静态控制流分析、静态污点分析、动态符号执行等技术的程序分析方法,通过有导向性的路径分析确定出目标程序点的污点控制流依赖关系,自动生成能够到达目标程序点并触发对应缺陷的输入实例.

1 系统框架

系统框架如图1所示,包括6个部件:良构输入生成器、执行监视器、静态控制流分析引擎、静态污点分析引擎、动态符号执行引擎和缺陷验证器.

在给定目标程序P和P中已知的缺陷程序点θ、程序良构输入对应的协议规约τ的情况下,良构输入生成器应用基于生成的Fuzzing技术生成满足τ的良构种子输入,交由执行监视器监视其在P中的执行轨迹,从中筛选出能够到达θ点所在函数f的良构输入集合Inputs;静态控制流分析引擎计算出函数f到程序点θ的所有静态可达的控制流路径集合CFG_{slc}.静态污点分析引擎基于CFG_{slc},结合动态污点分析得到的函数f入口点的污点状态,应用流敏感的控制流污点分析,筛选出能将污点输入传播到θ程序点处敏感操作数的路径集合$CFG_{t\text{-}slc}$.动态符号执行引擎综合Inputs执行轨迹和$CFG_{t\text{-}slc}$蕴含的路径信息,引导符号执行进行朝向θ的安全分析.缺陷验证器最终对获取的输入实例进行测试,确认是否能触发θ点处的目标缺陷.

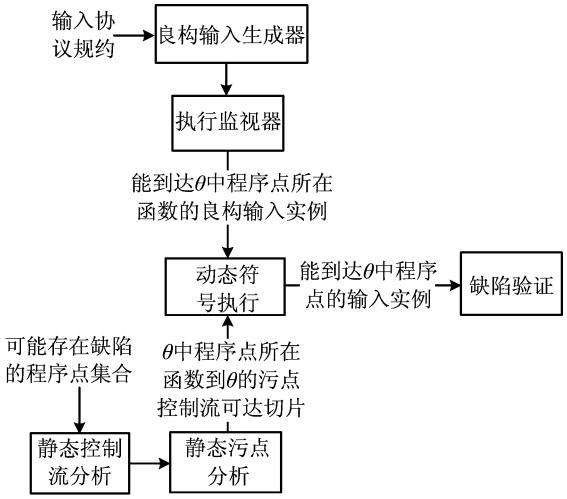

图1 系统框架图

Figure 1 System architecture

2 控制流污点信息计算

静态控制流污点信息计算包括两个部分:控制流信息计算和污点信息计算.这部分工作实现基于Binnavi二进制分析平台[10].

2.1 静态控制流分析

给定目标程序P和脆弱程序点θ,本文的静态控制流分析首先确定出包含θ的函数f.对包含θ的模块M,构造过程调用图CG和由每个函数$func_i$的过程内控制流图cfg_i构成的集合CFG.其间对于M中存在的间接跳转、间接调用等情况,本文采用跳转表识别、具体输入执行记录分析等手段补足控制流信息.

完成了CG和CFG的计算后,目标点θ静态可达路径信息的计算如下:

算法2.1 TargetCFGPath(CG,CFG,θ,f)

```
entry = f.entry;
new_cfg = CFG[f];
//依据调用图获取 f 调用的所有函数
passed_funcs=get_passed_func(CG,f);
if(passed_funcs ISNOT empty)
  foreach func in passed_funcs
  new_cfg = integrate_funcs( new_cfg,
    CFG[func]
                                    );
```

```
            end
        end
        preds = get_predecessors(new_cfg, θ);
        succs = get_successors(new_cfg, entry);
        nodes = preds ⋂ succs;
        return nodes;
    end
```

算法 2.1 首先利用模块 M 的调用图 CG 确定出函数 f 调用的所有函数 passed_funcs，然后应用 integrate_funcs 函数将各个函数的过程内控制流图内联到函数 f 的控制流图中. 内联过程如下：

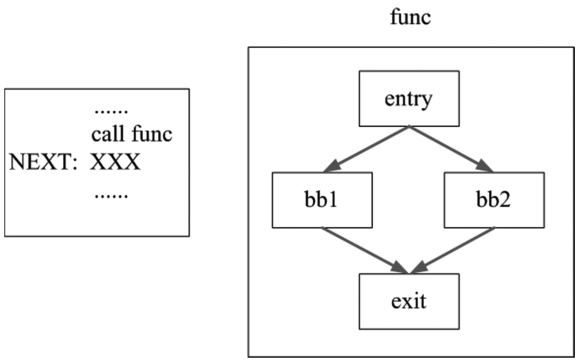

图 2　内联前的控制流图

Figure 2　Control flow graph before inlining

基本块 bb0 在切分前如图 2. integrate_funcs 函数首先将 bb0 沿着 NEXT 标记切分成 bb0-1 和 bb0-2 两个独立的基本块，其中 bb0 基本块的所有入口边的源节点全部设为基本块 bb0-1，bb0 基本块的所有出口边的源节点全部设为基本块 bb0-2. 随之构造两条边：一条边从 bb0-1 指向被调用函数 func 的入口基本块 entry，另一条从被调用函数 func 的 exit 基本块指向基本 bb0-2. 最终形成的控制流图如图 3 所示.

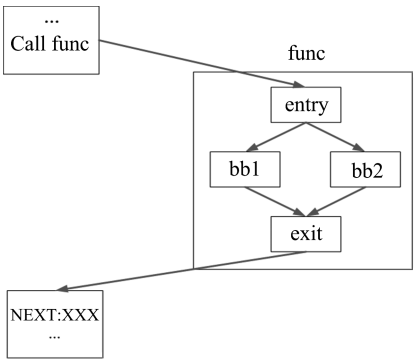

图 3　内联后的控制流图

Figure 3　Control flow graph after inlining

当完成了所有的内联操作后，应用图论算法求解 θ 的前驱节点集合和 entry 的后继节点集合，两者的交集即为函数 f 对 θ 程序点静态可达的控制流切片 CFG_{slc}.

2.2　污点信息计算

上节的算法能够计算出目标函数 f 入口点到目标程序点 θ 的静态控制流可达切片 CFG_{slc}. 然而，并非沿着该切片执行的所有路径都能够将外部输入相关信息（污点信息）传播到 θ 处的敏感操作数 λ. 需要计算出能够将污点传播到 λ 的基本块序列.

静态污点分析[11]一般以外部输入的读取位置（诸如 Linux 系统下的 read 函数调用点）作为污点引入点，将相关的读取缓冲区内容标记为污点操作数，沿着程序控制流图（一般就单个可执行程序模块而言）计算污点数据流信息，考察关键程序点（sink）处敏感操作数的污点状态. 然而在二进制分析背景下，污点数据的读取位置可能不在包含目标函数 f 的模块内，导致无法在对该模块的数据流分析中引入污点信息.

本文采用了一种动态污点分析和静态污点分析结合的方法. 首先由动态污点分析，以基于生成的 Fuzzing 筛选出的能够到达函数 f 的良构输入集合 Inputs 为种子输入，计算出在函数 f 入口点处的污点程序状态（包括寄存器、内存等）$State_f$，静态污点分析引擎随之将 2.1 节中计算出的目标函数 f 入口点到目标程序点 θ 的控制流可达切片 CFG_{slc} 标准化（确保 CFG_{slc} 中入口点 entry 只有一个入口边，出口点 exit 只有一个出口边），提升所有二进制指令为 vine 中间语言，并转换为静态单赋值形式，在图 4 所示的控制流污点格上进行流敏感的正向数据流分析.

如图 4，控制流污点格 $L=<V, \wedge, \leqslant>$ 由下列元素构成：

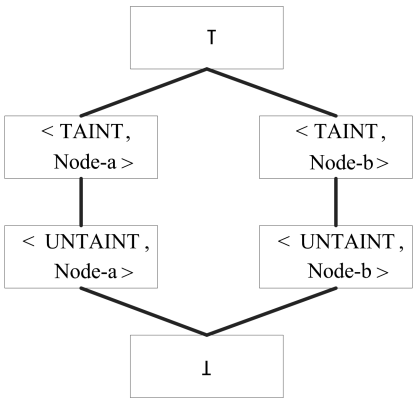

图 4　控制流污点格

Figure 4　Control-flow taint lattice

(Ⅰ)元素值集 V

定义程序中任一时刻某一定值 dv_j 的污点控制流信息如下：

$$ct_j = \{< bb_k, t_k > | bb_k \in \mathrm{CFG_{slc}}, t_k \in \mathrm{UNTAINT, TAINT}\}\}$$

其中 $t_k \in$ UNTAINT, TAINT}}表征该定值的污点状态，bb_k 表征能将 t_k 传递到 ct_i 的基本块. ct_i 元素的取值集合构成 L 的值集 V，定义为 $V = \{ct_i \mid dv_i \in DV(CFG_{slc})\}$.

(Ⅱ)交汇运算符 $\wedge$

$\wedge$ 在正向数据流分析中用于实现任一基本块所有入口边控制流污点信息的交汇运算. 对于 V 中任意两个元素 ct_i 和 $ct_{i'}$，定义函数 bbsame(ct_i, $\mathrm{ct_{i'}}$)抽取出两个元素中都包含的基本块集合，函数 bbunsameextract(ct_i, bblist)抽取出 ct_i 中不包含在 bblist 中的基本块及对应污点状态序列，交汇运算 $ct_i \wedge ct_{i'}$ 定义如下：

$ct_i \wedge ct_i' =$

$\{< bb_j, \mathrm{taint\_}merge(ct_i[bb_j], ct_i'[bb_j]) >$

$| bb_j \in \mathrm{bbsame}(ct_i, ct_i')\} \cup$

$\mathrm{bbunsameextract}(ct_i, \mathrm{bbsame}(ct_i, ct_i')) \cup$

$\mathrm{bbunsameextract}(ct_i, \mathrm{bbsame}(ct_i', ct_i'))$

其中，谓词函数 taint_merge 用于对两个不同的污点状态进行合并操作，定义如下：

算法 2.2 taint_merge(t_i, $t_{i'}$)

```
  if(t_i) ==TAINT or t_i' ==TAINT)
return TAINT
else if(t_i==UNTAINT or t_i' UNTAINT
)
return UNTAINT
else
   return ⊥
  end
end
```

(Ⅲ)偏序关系 $\leqslant$

$\leqslant$ 是 L 上的偏序关系，此处定义如下：

对于 V 中任意元素 t_i 和 $t_{i'}$，$t_i \leqslant t_{i'}$ 当且仅当 $t_i \wedge t_{i'} = t_{i'}$.

基于图 4 所示的污点格，静态控制流污点分析引擎以 $\mathrm{State_f}$ 为初始污点状态，进行正向数据流分析，确定出切片中各个基本块的入口污点状态和出口污点状态. 分析框架如算法 2 所示.

算法 2.3 ComputeCFGTaint($\mathrm{State_f}$, $\mathrm{CFG_{slc}}$)

```
OUT[]=
  patch(CFG_slc, entry);
foreach <State_f, CFG_slc bb in CFG_slc do
  if bb is not CFG_slc. eytry
    OUT[bb] = ⊥
  end
end
while(某个 OUT 值发生改变) do
  foreach bb in CFG_slc do
    if bb is not CFG_slc. entry
      IN[bb] = ∧_{P∈pred(bb)} OUT[P]
      OUT[bb] = f_bb(IN[bb])
    end
  end
end
```

其中，patch 函数首先将 $\mathrm{State_f}$ 中的污点信息结合 $\mathrm{CFG_{slc}}$ 的入口基本块，补足为格中的控制流污点信息. 随之对控制流切片中各个基本块进行迭代，求解出各个基本块的入口控制流污点状态(应用格的交汇算符 $\wedge$ 计算)和出口控制流污点状态(应用基本块状态迁移函数 f_bb 计算). 可以证明算法迭代过程中的程序抽象状态始终沿着定义的控制流污点格属性上升方向变更，算法的不动点计算一定能达到收敛状态[12]. 迭代结束时，控制流切片的 exit 基本块中的目标定值对应的控制流污点状态中维护的基本块序列即为能够将污点传播到该定值的基本块集合 φ_θ. 通过将 $\mathrm{CFG_{slc}}$-φ_θ 集合中的基本块从控制流切片中删除，基于得到的控制流污点可达切片 TCFG_θ 进行计算，可以进一步约减后端导向式符号执行的路径状态空间.

3 导向式符号执行

大型程序路径状态空间庞大，能到达目标程序点的路径数量则相对较小. 在限定计算资源的情况下分析引擎对这部分路径的覆盖概率较低. 导向式符号执行首先基于目标程序的控制流图计算出目标程序点控制流可达路径，仅仅对这部分路径进行分析. 由此确保了在资源限定的情况下，可以生成能够到达目标程序点的输入实例. 然而，在二进制程序分析背景下，当前导向式符号执行存在如下问题：

(Ⅰ)二进制程序运行时包含多个可执行模块，计算程序路径所需要的完整的控制流图往往并不可

得,一般只能关注于可执行程序自身模块的控制流计算. 对于目标程序点在其它动态可加载模块的情况并适用;

(Ⅱ)程序的执行路径不仅与程序的控制流相关,同时也与其数据流相关. 为便于计算可达路径,当前控制流分析一般将控制流图中循环结构的回边、调用图中递归结构的回边略去. 由此导致那些能够到达目标程序点同时触发缺陷的程序路径,由于在对循环等关键程序结构的遍历中并不遵从分析假定的简单模式而被丢弃.

针对上述问题,结合"缺陷程序点所在函数往往包含在良构输入的处理路径中"这一启发式原则[13],本文提出一种单路径具体-符号混合分析和多路径符号分析相结合的方法. 如图 5 所示(图 5 中实线表征符号执行实际分析的程序路径,虚线表示控制流可达而符号执行并未进行分析的程序路径),以包含目标程序点 θ 的函数 f 为界,在程序入口点到函数 f 入口点之间,利用基于生成的 Fuzzing 技术构造的能够到达函数 f 的良构种子输入 I 驱动符号执行引擎进行"严格遵循 I 执行轨迹"单路径符号执行,记录 f 入口点的程序机器状态(包含具体机器状态和符号机器状态)$\mathrm{SCState}_f$;在函数 f 入口点到目标程序点 θ 之间,则以 $\mathrm{SCState}_f$ 为初始状态,基于第 3 节中计算出的控制流污点可达切片进行"忽略不相关路径"的导向式多路径符号执行.

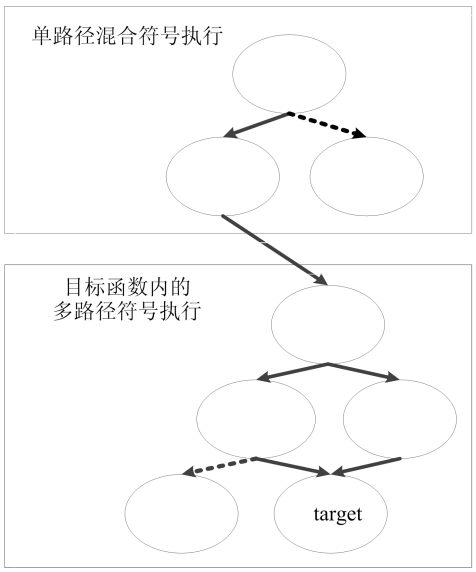

图 5　导向式符号执行

Figure 5　Directed symbolic execution

任意一条函数 f 内的程序路径 $\mathrm{path_n}$ 可由如下形式描述:$\mathrm{path_n}=[B_1,B_2,\cdots,B_n]$,其中 B_1 表示路径途经的基本块($i\in[1,n]$且 i 为整数),B_1 为路径的起始基本块,B_n 为路径的终止基本块(特别地,对于能够到达程序点 θ 的程序路径 path_θ,将其包含 θ 的基本块 B_θ 之后的基本块序列一律略去,使得 B_θ 即为该路径的终止基本块). 对于任意 $i\in[1,n]$,程序的控制流图中都存在着唯一的一条连接 B_1 和 B_{i+1} 的流程转移边. 定义 $\mathrm{Path}_{\mathrm{TCFG}_\theta}$ 表征控制流污点切片 TCFG_θ 所包含的程序路径集合. 可知如下条件恒成立:

$$\neg(\mathrm{path}_k\in\mathrm{Path}_{\mathrm{TCFG}_\theta})\rightarrow\exists_i((B_i\in\mathrm{path_k})\ \mathrm{and}\ (B_i\notin\mathrm{TCFG}_\theta))$$

由此,在计算出 TCFG_θ 的情况下,与仅仅顺从于 $\mathrm{Path}_{\mathrm{TCFG}_\theta}$ 中的程序路径执行轨迹不同,本文采用的"忽略不相关路径"的导向式多路径符号执行技术首先基于 TCFG_θ 计算出每个分支点 b_i 处的可能后继基本块集合 $B\mathrm{Set}_i$. 在动态符号执行对分支程序点 b_i 的语义计算时,在 b_i 的分支谓词为符号值的情况下,略去所有后续执行基本块不在 $B\mathrm{Set}_i$ 集合内的路径状态. 在对循环结构的迭代中采用固定迭代次数的分析策略. 由此实现了面向目标程序点的导向式路径分析.

4　实验分析

基于 Fuzzing 平台 Peach[14]、二进制静态分析平台 BinNavi 和全系统动态符号执行平台 S2E[15],本文构建了面向缺陷程序点的导向式二进制程序符号执行原型系统,以 Windows XP Professional 版本 2002 Service Pack 3 操作系统平台下实际公开的程序缺陷为数据源对方法有效性进行了验证实验,其中缺陷程序点通过人工分析缺陷成因确定. 缺陷的主要信息如下:

对上述给定缺陷,利用基于生成的 Fuzzing 技术构造出能够到达 bug 函数的种子输入,驱动符号执行沿单路径快速到达 bug 函数,进而在 bug 函数内朝向 bug 程序点进行导向式路径分析,成功生成能够到达缺陷程序点的测试用例.

本文从基本块约减、路径约减两方面对提出的"忽略不相关路径"的导向式多路径符号执行的计算性能进行分析. 表 2 给出了目标函数内基本块约减

表1 实验缺陷信息表

Table 1 Basic information for real world defects

Bug#	问题	Bug函数	Bug程序点
OSVDB 91256	内存非法访问	cam2pc.exe 0x591ff0	cam2pc.exe 0x5920b2
CVE 2010-3000	整数溢出	flvff.dll 0x613ecfb0	flvff.dll 0x613ed255
CVE 2010-0028	整数溢出	gdiplus.dll 0x4afc3844	gdiplus.dll 0x4afc39be
EDB 15034	内存非法访问	gdi32.dll 0x77ef9d95 NTAudioIn-forma-tion 2.dll	gdi32.dll 0x77f17216 NTAudioIn-forma-tion 2.dll
EDB 24880	内存非法访问	0x238bd30 CoolType.dll 0x8010d37	0x238be98 CoolType.dll 0x8010df0
CVE 2010-2862	内存非法访问		

情况.其中PreBlocks表征约减分析前目标函数内联基本块的数目,PostBlocks表征约减分析后目标函数内联基本块的数目.

表2 导向式符号执行计算目标函数基本块约减表

Table 2 Basic block reduction for directed symbolic execution

Bug#	PreBlocks	PostBlocks	Rate
OSVDB-91256	27	12	44.44%
CVE-2010-3000	84	35	41.67%
CVE-2010-0028	36	26	72.22%
EDB-15034	31	10	32.25%
EDB-24880	275	63	22.91%
CVE-2010-2862	59	9	15.25%

表3给出了目标函数内符号计算路径的约减情况.其中PrePaths表征约减分析前的可达路径数目,PostPaths表征约减分析后的可达路径数目.

表3 导向式符号执行目标函数路径约减表

Table 3 Path reduction for directed symbolic execution

Bug#	PrePaths	PostPaths	Rate
OSVDB-91256	60	21	35%
CVE-2010-3000	474	65	13.7%
CVE-2010-0028	124	15	12.1%
EDB-ID-15034	40	5	12.5%
EDB-ID-24880	1380	76	5.5%
CVE-2010-2862	185	2	1.1%

从表2、表3可见,应用本文方法对实际程序进行安全分析,程序中目标函数的符号计算规模得到显著约减.

此外,在对OSVDB-91256程序缺陷的验证实验中,系统还发现了未公开的除零缺陷,位于cam2pc.exe模块,虚拟地址0x560052,同时生成了2222字节的对应输入实例.

5 结论

本文对面向给定程序点的测试用例快速生成技术进行了研究,提出了一种导向式动态符号执行方法.主要工作包括:应用基于生成的Fuzzing技术生成种子输入,导向单路径符号执行快速到达脆弱函数;结合静态控制流分析和静态污点分析计算出脆弱函数到脆弱程序点的静态可达路径集合,导引多路径动态符号执行引擎仅仅关注于这些路径的符号分析.实验验证了方法作为路径爆炸问题减轻手段的有效性.

下一步的工作包括:多线程计算环境下程序缺陷的分析建模;将该方法与静态程序分析工具相结合,进而降低静态分析中广泛存在的误报率较高问题等.

参考文献(References)

[1] Goldfroid P, Klarund N. Sen K. DART: Directed automated random testing[C]// SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM Press, 2005: 213-223.

[2] Godefroid P, Levin M Y, Molnar D. Automated whitebox fuzz testing[C]// Proceedings of Network Distributed Security Symposium. San Diego, USA: The Internet Society, 2008: 151-166.

[3] Brumley D, Hartwig C, Kang M G, et al. BitScope: Automatically dissecting malicious binaries [R] Technical report CMU-CS-07-133, Carnegie Mellon University, 2007.

[4] Păsăreanu C S, Mehlitz P C, Bushnell D H, et al. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software [C]// Proceedings of the 2008 International Symposium on Software Testing and Analysis. Seattle, USA: ACM Press, 2008: 15-26.

[5] Hu C J, Li Z J, Ma J X, et al. File parsing vulnerability detection with symbolic execution[C]// Sixth International Symposium on Theoretical Aspects of Software Engineering. Beijing, China: IEEE Press, 2012: 135-142.

[6] Wang X, Chen H G, Jia Z H, et al. Improving integer security for systems with KINT[C]// Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. Berkeley, USA: USENIX-Association, 2012: 163-177.

[7] Chipounov V, GeorgescuV, Zamfir C, et al. Selective symbolic execution [C]// Proceedings of the 5th Workshop on Hot Topics in System Dependability. Lisbon, Portugal: ACM Press, 2009: 1-6.

[8] Siddiqui J H, Khurshid S. ParSym: Parallel symbolic execution [C]// 2nd International Conference on Software Technology and Engineering. San Juan, Puerto Rico: IEEE Press, 2010: 405-409.

[9] autodafe-fuzzer framework [EB/OL]. http://sourceforge.net/projects/autodafe/? source=directory [2015-1-31].

[10] BinNavi[EB/OL]. www.zynamics.com/binnavi.html, 2013.

[11] 王金锭, 王嘉捷, 程绍银, 等. 基于统一中间表示的软件漏洞挖掘系统[C]// 第三届信息安全漏洞分析与风险评估大会. 合肥, 2010: 42-52.

[12] Alfred V A, Monica S L, Ravi S. 编译原理[M]. 第 2 版, 赵建华, 郑滔, 戴新宇译, 北京: 机械工业出版社, 2009.

[13] 崔宝江, 梁晓兵, 王禹, 等. 基于回溯与引导的关键代码区域覆盖的二进制程序测试技术研究[J]. 电子与信息学报, 2012, 34(1): 108-114.
Cui B J, Liang X B, Wang Y, et al. The study of binary program test techniques based on backtracking and leading for covering key code area[J]. Journal of Electronics &Information Technology, 2012, 34(1): 108-114.

[14] PEACH FUZZE[EB/OL]. http://peachfuzzer.com/, 2012.

[15] Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems[J]. ACM SIGARCH Computer Architecture News, 2011, 39(1): 265-278.