基于动态符号执行的二进制程序缺陷发现系统

黄 晖, 陆余良, 夏 阳
(解放军电子工程学院 网络系, 合肥 230037)

摘 要: 以对二进制程序进行自动化缺陷发现为目标,基于软件虚拟机的动态二进制翻译机制和污点传播机制,对符号计算需要关注的程序运行时语义信息提取、中间语言符号计算等机制进行了研究,改进了传统动态符号执行的路径调度部分,分析了程序缺陷的符号断言表达形式,构建了一个在线式的动态符号执行系统检测二进制程序中的缺陷。实验验证了该方法在实际程序缺陷发现中的有效性。

关键词: 语义提取; 动态符号执行; 路径调度; 二进制程序缺陷发现
中图分类号: TP311 **文献标志码**: A **文章编号**: 1001-3695(2013)09-2810-03
doi:10.3969/j.issn.1001-3695.2013.09.064

Dynamic symbolic execution based defect detection system for binary programs

HUANG Hui, LU Yu-liang, XIA Yang
(*Dept. of Networks, Electronic Engineering Institution of PLA, Hefei* 230037, *China*)

Abstract: Aiming towards automatic defect detection for binary programs, based on software virtual machine's dynamic binary translation and taint propagation, this paper studied mechanisms necessary for symbolic execution including program's runtime semantics' extraction, intermediate language based symbolic calculation, enhanced the path-scheduling mechanism in traditional dynamic symbolic execution, analyzed symbolic asserts' expressions for common program defects, with an online dynamic symbolic execution system built up detecting defects in binary programs. Experiments prove the method's effectiveness in defect detection for real binary programs.

Key words: semantic extraction; dynamic symbolic execution; path scheduling; defect detection for binary programs

动态符号执行技术通过在程序运行过程中隐式地为程序输入引入符号变元,求解每条指令的符号计算语义,在路径敏感的分析中收集分支谓词,对目标程序所有输入相关的路径以约束条件等价类的形式进行特征刻画。近年来动态符号执行技术在软件漏洞挖掘领域得到了广泛的应用[1~3]。

以对二进制程序进行安全性分析为目标,本文构造了一个动态符号执行的 x86 平台下二进制程序缺陷发现系统。

1 动态符号执行系统框架

系统的设计实现基于伯克利学院的 BitBlaze 项目[4],该项目包括 temu 虚拟机[5] 和 vine 中间语言[6] 两个部分。前者提供了污点标记维护、动态污点传播、目标系统 API 挂钩等适于监视目标程序对外部数据处理流程的分析机制;后者则用于将二进制程序提升至中间语言层面进行静态分析。本文禁用了 temu 内置于动态二进制翻译中的污点传播机制。该机制实质上是在符号计算利用污点标记维护引擎维护程序运行时符号机器状态的过程中完成。

架构于 temu 虚拟机及其污点标记、维护机制之上,结合 vine 中间语言,本文构建的动态符号执行系统如图 1 所示,由二进制程序运行时语义信息抽取和中间语言符号计算两部分内容构成。前者负责提取出必要的程序运行时信息进行符号计算、进程上下文过滤;后者则关注于程序指令在抽象机器域下的符号语义计算。

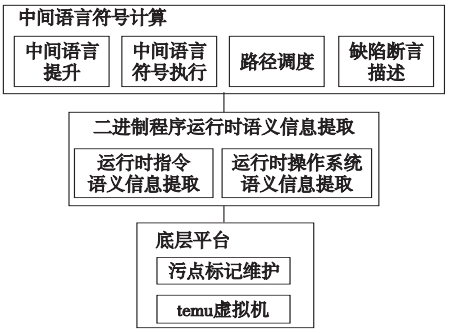

图1 动态符号执行系统框架

2 程序运行时语义信息抽取

2.1 运行时指令语义信息提取

指令语义信息包含符号机器下的指令语义和具体机器下的指令语义。符号语义关注于指令本身是否涉及污点操作数的计算。非流程转移类指令的执行一般包含操作数的读取和写入动作。可以认为,只要其读取的操作数或读取的操作数的地址为被污染的,或写入的目标操作数的地址为污点相关,该指令即为污点传播相关的指令。而对于流程转移类的指令,在符号计算的背景下,只需关注条件跳转相关的分支谓词计算是否与污点操作数相关即可。具体语义则关注于指令计算前某些操作数的数值信息。

图 2 给出了单一指令的具体计算和符号计算流程。可见

收稿日期: 2012-11-19; **修回日期**: 2013-01-04

作者简介:黄晖(1987-),男,江苏海门人,博士研究生,主要研究方向为信息安全、程序验证(hhui_123@163.com);陆余良(1964-),男,教授,博导,主要研究方向为信息安全、网络安全;夏阳(1977-),男,副教授,博士,主要研究方向为网络安全.

对于一条污点计算相关的指令,系统将分别对其具体机器意义下的具体语义和符号机器意义下的抽象语义进行计算。而两者并不是同时进行的:在软件虚拟机对目标指令仿真过程完成具体计算后才进行符号计算。由于这种不同步性,导致分析引擎只有在具体计算期间对目标指令进行必要的具体操作数信息获取,并在符号计算前进行必要的上下文恢复,才能确保符号计算的分析正确性。Temu 虚拟机对于机器体系结构的软件仿真使得指令计算信息的提取变得容易。本文在二进制的动态翻译过程中通过对操作数获取、计算相关的微指令进行挂钩,实现了上述的运行时信息提取。

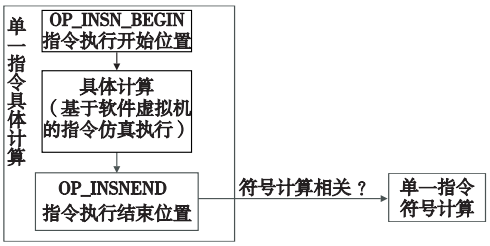

图2　单一指令的具体计算和符号计算

2.2　运行时客户操作系统语义信息提取

对目标程序运行状态的监控,必须实时了解客户操作系统的相关信息。Temu 虚拟机对此虚拟出了一个对应端口号为 0x68 的硬件设备,客户端系统中的内核模块可经该端口与虚拟机的监控模块通信。由此,通过在客户端系统内加载定制的内核模块,提取了诸如进程 PID、进程页表物理地址、进程描述符地址之类的进程相关信息。利用定义具体操作系统相关的解析模块,进一步得到线性区描述符 VAD 等信息,为诸如符号地址解析等问题的解决提供了必要的上下文。系统中关键 API 函数的挂钩也以类似的方式实现。

3　中间语言符号执行引擎

中间语言符号执行引擎包括中间语言提升、符号计算、路径调度和缺陷断言描述四个部分。中间语言提升依托于 vine 本身的提升套件实现。在完成转换后,借助符号计算、路径调度和缺陷断言描述,分析引擎实现了对目标程序在路径覆盖基础上的自动化缺陷发现。

3.1　基于中间语言的符号计算

目标程序指令确定为污点计算相关后,将被提升为中间语言表示进行抽象属性的分析计算。本文采用的 vine 中间语言在设计上具有如下特点:每条语句都仅仅对应于单一计算语义的描述;指称操作数的 exp 型非终结符上通过数据类型指明了具体操作数的位长。这些特点为与 STP[7] 之类的基于位向量的约束求解器相结合、构造语法制导的抽象计算规则提供了便利。

Vine 的文法主要由表征表达式的 exp 型节点和表征语句的 stmt 型节点两部分构成。本文对于 exp 型的非终结符,定义左值属性和右值属性两个属性。左值属性指该非终结符在抽象语法树的上一级节点的语义计算中作为地址操作数时的属性;右值属性指该非终结符在抽象语法树的上一级节点的语义计算中作为具体操作数时的属性。在设计的计算模型中,左值属性有以下几种:通用寄存器索引、EFLAGS 寄存器中标记位的索引、临时变量在临时符号表中的索引以及具体内存虚拟地址。右值属性则有关于输入的符号表达式和具体值两种。对于 stmt 型的非终结符,依据具体的指令意义定义计算规则。特别地,对分支跳转相关的 cjmp 类型的 stmt 非终结符,翻译规则将抽取出对应当前路径的分支谓词,以此更新当前路径的符号约束条件。

3.2　路径调度

为快速发现目标程序中可能存在的缺陷,本文默认的路径调度策略为深度优先。在调度新路径执行时,传统的动态符号执行[1~3] 往往采用生成测试用例重新执行的方法。如图 3 所示,假定 entry 节点对应于目标程序的入口点。目标程序运行至 A 分支节点时的路径约束为 p。在分支节点 A 依据分支谓词 $x=1$? 生成朝向节点 B 和 C 的两条新的路径状态(假定两个路径状态对应的路径约束都是可满足的),并随后立即调度朝向节点 B 的路径执行。当传统的路径调度引擎调度朝向节点 C 的路径执行时,首先依据对应的路径约束 P AND ¬ ($x==1$)生成满足条件的输入 x 的具体值 3,然后从程序的入口点开始对新路径同时进行具体计算和符号计算。但由于节点 B 和 C 对应的路径状态有很强的相关性(从 entry 节点到节点 A 的分析结果中的符号机器状态可以在完成与 $x=1$? 分支谓词相关部分的更新后应用于 B 和 C 中,而具体机器状态可以依据符号机器状态进行恢复)。事实上,该方法对 C 路径从程序入口点开始进行完全分析的策略造成了很大的分析浪费,尤其在目标程序规模较大的情况下,显然该策略将直接影响到分析性能。

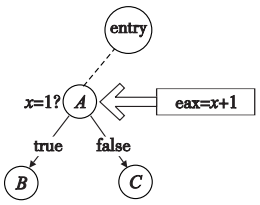

图3　分支点处具体机器状态的动态更新

通过与软件虚拟机快照机制(添加对符号机器状态的保存)相结合,本文改进的路径调度机制如下:如图 3 所示,在分支点 A 处首先利用快照机制保存当时的具体机器状态和符号机器状态。当重新调度执行朝向节点 C 的路径时,更新当前路径约束为 P AND ¬($x==1$),并以此计算满足条件的符号变元 x 的具体值(假定此处计算为 3);以该值更新符号机器状态中与 x 相关的 eax 寄存器的状态(由 $x=3$ 计算出 $x+1=4$,将具体值 4 写入 temu 中 eax 的具体存储位置);朝向节点 C 的路径状态的调度执行将从分支节点 C,而不是 entry 节点开始。由此,新策略实现了分支节点处父节点的符号机器状态在子节点路径状态分析过程中的实时重用,在节约了分析开销的同时,保证了新路径分析中符号机器状态、具体机器状态、路径约束三者之间的一致性。

3.3　程序缺陷的符号断言描述

基于 vine 中间语言,本文以断言形式对关注的程序缺陷进行描述。通过在缺陷可能存在的程序位置,结合当时的符号机器状态对相关断言的可满足性进行判定,确定对应的缺陷是否存在于当前路径约束所描述的执行路径中。具体断言形式有:

a)中间语言二元运算相关语句的翻译规则中的缺陷判定断言。这一类型断言用于检测二元运算中可能出现的除零、除法整数溢出等缺陷。

b)中间语言内存访问运算语句对应翻译规则中的缺陷判定断言。面向地址受污点操作数影响而导致的非法内存访问问题,综合得到的操作系统语义相关的进程虚拟地址描述符信息。本文关注以下两种类型的断言描述:(a)综合符号地址表达式和当前映射的内存区中与当前的程序访存模式不匹配的

线性地址范围得到的断言条件;(b)综合符号地址表达式和不在当前映射的内存区之内的所有线性地址范围所得到的断言条件。前者用于检测内存访问违例型缺陷,后者则用于检测内存访问越界型缺陷。

c)目标操作系统中敏感的系统调用服务,作为可能的 sink 点[8]而进行的参数校验。大部分整数溢出[8]可以归纳为这种类型的程序缺陷。

d)对于缓冲区溢出型程序缺陷,本文为每组程序敏感输入引入符号长度变量,基于 IDA pro [9] 识别出目标程序中如 strcpy、strcat 等不安全函数,并利用 temu 提供的 hookapi 机制进行了动态挂钩,结合目标程序运行时调用栈等信息构造断言,进行动态检测。

4 实验

本文以 WindowsXP 原版系统中 wmplayer 媒体播放器中存在的 CVE 编号为 CVE-2009-3201 的一个程序漏洞的验证,来说明本文方法在实际缺陷发现中的有效性。实验目标系统为 WindowsXPSP2 原版系统。对应的 wmplayer 的版本为 9.0.0.3250。以良构的 expl. mid 文件作为输入,本文对 wmplayer 媒体播放器进行测试。文件的内容如图 4 所示。

```
0000h: 4D 54 68 64 00 00 00 06 00 00 00 01 00 78 4D 54
0010h: 72 6B 00 00 00 3A 00 FF 58 04 04 02 18 08 00 FF
0020h: 59 02 00 00 00 FF 51 03 07 A1 20 00 90 56 64 1E
0030h: 56 00 00 56 64 81 70 56 00 00 56 64 81 70 56 00
0040h: 00 48 64 1E 48 00 00 4A 64 1E 4A 00 00 FF 2F 00
```

图4 良构测试用例

不进行符号计算,以图 4 所示的文件作为输入。输入文件并未触发程序漏洞。启动虚拟机中的客户操作系统,调度分析插件,在虚拟机中以 expl. mid 文件为输入启动 wmplayer. exe 媒体播放器。经过 1 h 的符号计算,发现在 quartz. dll 模块中虚拟地址 0x7d062021 处的 div ecx 指令的执行中,ecx 在符号机器中对应于如下形式的符号约束:

```
(LET let_k_0 = (0hex0000 @ (expl. mid__13 @ expl. mid__12)),
let_k_1 = (0hex00000000 | ((0hex000000 @ (let_k_0[31:24]@
(let_k_0[23:16]@ let_k_0[15:8] @ let_k_0[7:0])))[7:0]) @
0hex00)[31:0]),
let_k_2 = (let_k_0[31:24] @ (let_k_0[23:16] @ (let_k_0[15:
8] @ let_k_0[7:0]))),
let_k_3 = (0hex00 @ let_k_2[31:8]),
let_k_4 = ((let_k_1[31:24]@ (let_k_1[23:16] @(let_k_1[15:
8]@let_k_1[7:0])))
|(let_k_3[31:24] @ (let_k_3[23:16] @ (let_k_3[15:8] @ let_k_
3[7:0])))),
let_k_5 = (let_k_4[31:24] @ (let_k_4[23:16] @ (let_k_4[15:
8] @ let_k_4[7:0]))),
let_k_6 = (let_k_5[31:24] @ (let_k_5[23:16] @ (let_k_5[15:
8] @ let_k_5[7:0]))),
let_k_7 = BVMULT(32, (let_k_6[31:24] @ (let_k_6[23:16] @
(let_k_6[15:8] @ let_k_6[7:0]))), 0hex000003E8),
let_k_8 = let_k_7[31:24] @ (let_k_7[23:16] @ (let_k_7[15:
8] @ let_k_7[7:0]))),
let_k_9 = (let_k_8[31:24] @ (let_k_8[23:16] @ (let_k_8[15:
8] @ let_k_8[7:0])))
IN
(0hex00000000 @ (let_k_9[31:24] @ (let_k_9[23:16] @ (let_k_9
[15:8] @ let_k_9[7:0]))))
```

可见,ecx 在 0x7d062021 处的除法指令中,对应于输入文件的第 12 和 13 号字节(字节从 0 号开始计数)相关的符号表达式。基于除零缺陷对应的漏洞断言,利用约束求解器求解出的问题测试文件样例如图 5 所示。

以图 5 所示生成的测试用例作为 wmplayer 的输入文件,成功触发除零缺陷。此外,应用 3.3 节论述的缺陷断言,给定良构的输入文件,分析系统能够正确检查出 Fox Player 1.7 栈缓冲区溢出缺陷、Google Chrome 9.0.597.84 的 libpng 组件中存在的整数溢出缺陷。目前正使用该系统对其他应用软件进行安全性测试。

```
0000h: 4D 54 68 64 00 00 00 06 00 00 00 01 00 00 4D 54
0010h: 72 6B 00 00 00 3A 00 FF 00 00 00 FF 2B 02 00 00
0020h: 00 FF 51 03 00 00 00 7E A0 00 00 06 00 00 15 00
0030h: 00 FC 01 00 00 27 00 00 82 71 00 00 63 00 00 0E
0040h: 00 00 58 00 00 09 00 00 00 FF 2F 02 00 00 00 00
```

图5 能够触发异常的输入文件

5 结束语

基于开源项目 BitBlaze,本文构建了一个面向程序缺陷发现的动态符号执行系统。将二进制程序提升为中间表示,在运行时抽取重要的计算语义信息,依托于语法制导翻译规则求解程序在抽象符号机器中的计算语义。为提升分析性能,对传统符号执行的路径调度部分进行了修改。通过引入断言形式的属性检查确定程序是否违背安全规约,实现动态路径覆盖过程中的缺陷发现。下一步的工作包括:对缓冲区溢出型漏洞的深度建模、路径调度策略的优化以及对控制流与输入的间接依赖关系[10,11]进行摘要分析等。

参考文献:

[1] CADAR C, DUNBAR D, ENGLER D. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs[C]// Proc of the 8th Symposium on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008: 209-224.

[2] GOLDFROID P, KLARUND N, SEN K. DART: directed automated random testing[C]//Proc of Programming Language Design and Implementation. New York: ACM Press, 2005: 213-223.

[3] GODEFROID P, LEVIN M Y. MOLNAR D A. Automated white box fuzzing[C]//Proc of Network Distributed Security Symposium. San Diego: The Internet Society, 2008: 151-166.

[4] SONG D, BRUMLEY D, YIN Heng, *et al*. BitBlaze: a new approach to computer security via binary analysis[C]//Proc of the 4th International Conference on Information Society Security. [S. l.]: Springer-Verlag, 2008: 1-25.

[5] YIN Heng, SONG D. TEMU: binary code analysis via whole system layered annotation execution, Technical Report UCB/EECS-2010-3 [R]. Berkeley: University of California, 2010.

[6] Vine[EB/OL]. [2012-11-19]. http://www. bitblaze. cs. ber-edu/vine. html.

[7] GANESH V, DILL D L. A decision procedure for bit-vectors and arrays[C]//Proc of the 19th International Conference on Computer Aided Verification. Berlin: Springer-Verlag, 2007: 519-531.

[8] WANG Xi, CHEN Hao-gang, JIA Zhi-hao, *et al*. Improving integer security for systems[C]//Proc of the 10th USENIX Symposium on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2012: 163-177.

[9] EAGLE C. IDA Pro 权威指南[M]. 石华耀, 段桂菊, 译. 北京: 人民邮电出版社, 2007: 77-96.

[10] SAXENA P, POOSANKAM P, McCAMANT S, *et al*. Loop extended symbolic execution[C]//Proc of the ACM/SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM Press, 2009: 225-236.

[11] GOLDFROID P, LUCHAUP D. Automatic partial loop summarization in dynamic test generation[C]//Proc of International Symposium on Software Testing and Analysis. New York: ACM Press, 2011: 23-33.