RSA CONFERENCE
CHINA 2012
RSA信息安全大会2012

THE GREAT CIPHER

MIGHTIER THAN THE SWORD

伟大的密码胜于利剑

Practical Symbolic Execution

(实用符号执行)

Chaojian Hu

(忽朝俭)

NSFOCUS Information Technology Co., Ltd.

(绿盟科技)

RSA CONFERENCE
CHINA 2012
RSA信息安全大会2012

Outline

- 符号执行概述
  - 变异 or 生成
- 符号执行如何检测漏洞
- 符号执行遇到的主要问题
  - 路径爆炸，约束困顿，环境交互
  - 符号注入，约束爆炸
  - 约束收集
- 针对这些问题的可能缓解方法
- 实验与结论

什么是符号执行

- 通常表示为三元组<指令指针，路径函数，路径条件>
- 指令指针标识当前被分析的指令
- 路径函数表示程序路径中不同点处变量的值是变量初始值的函数，通常可表示为一个映射

$$f : D^n \rightarrow D^n$$

- 路径条件是路径在各分支点满足的分支条件的合取，通常可表示为一个布尔公式

$$c_1 \wedge c_2 \wedge \cdots \wedge c_n$$

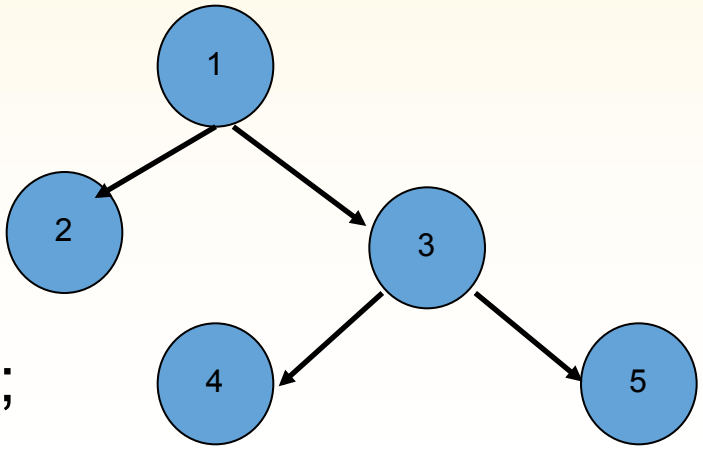
符号执行过程

- 初始化指令指针通常指向程序入口点，初始化的路径函数通常是恒等函数，初始化的路径条件通常为真
- 在符号执行中更新指令指针、路径函数和路径条件
  - 指令指针通常从当前指令转到其后继指令
  - 路径函数通常可通过复合路径上每条指令的语义函数（例如操作语义或者指称语义）得到更新
  - 路径函数的更新保证能够获取路径中每个分支的分支条件，通过合取路径中每个分支的分支条件能够获取路径条件

基于变异的符号执行

- 基于变异的符号执行通常混合具体与符号执行
- 首先使用一个具体输入向量驱使程序具体执行一条路径，同时使用符号执行更新路径函数和路径条件
  - 根据实际选择的分支目标依次在路径中的每个分支处收集相应的分支条件，并对所有的分支条件进行合取得到该路径的路径条件
- 然后系统地对路径条件中的每个分支条件取反以获取一组新的路径条件
- 对这些新的路径条件进行求解以获取一组新的具体输入向量，这组新的输入向量可驱使程序执行不同的路径
- 重复该过程，直到无法再变异

```
01 if(a < 100)
02   assert(1 && "a<100");
03 else if(b < 100)
04   assert(1 && "a<100&&b<100");
05 else assert(0 && "a>100&&b>100");
06 end：
```

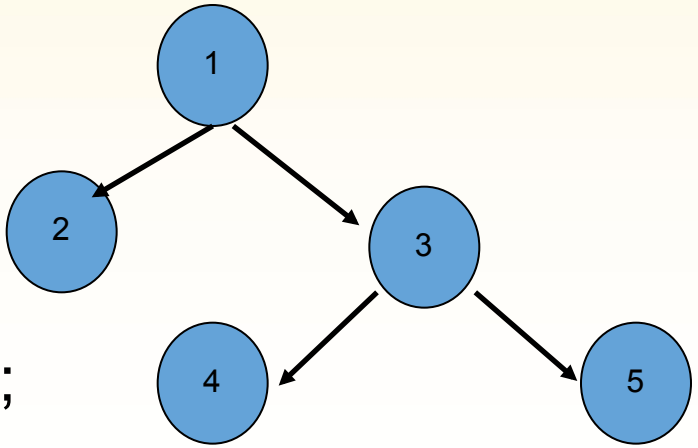


value	**path**	**path condition**
(0,0)	01-02-06	（a < 100）
新路径条件 ¬（a < 100）求解 a=100		
(100,0)	01-03-04-06	¬（a < 100） ∧ （b < 100）
新路径条件 ¬（a<100）∧ ¬（b<100）求解 a=100,b=100		
(100,100)	01-03-05-06	¬（a < 100）∧ ¬（b < 100）

基于生成的符号执行

- 基于生成的符号执行依次解释程序中的每条指令
- 对于非分支指令，其通常通过函数的复合更新路径函数
- 对于分支指令，其可能需要创建新的进程（执行一条新的路径）以探测每个分支目标，并将原路径条件与每个分支目标满足的分支条件取合取作为每个新路径的路径条件
- 当某个进程结束时，调用约束求解器求解路径条件以获取一个可驱使程序执行该路径的具体输入向量
- 当所有进程结束时，整个符号执行过程结束

```
01 if(a < 100)
02   assert(1 && "a<100");
03 else if(b < 100)
04   assert(1 && "a<100&&b<100");
05 else assert(0 && "a>100&&b>100");
06 end：
```



V	PC	branch condition
1	True	（a < 100）
2	（a < 100）	
3	¬（a < 100）	（b < 100）
4	¬（a < 100） ∧ （b < 100）	
5	¬（a<100）∧ ¬（b<100）	

符号执行如何检测内存越界

- 假定Q是程序中的一段程序片段，Q之后是一个内存读/写操作；P是执行程序片段Q之前的路径条件；R表示程序片段Q之后的内存读/写操作需要满足的安全约束，其形式如下

$$(V_{base} \le OP_{addr}) \wedge ((OP_{addr} + OP_{size}) \le (V_{base} + V_{size}))$$

- 对程序片段Q的符号执行过程以初始路径条件P开始，通过合取程序片段Q中的一条路径片段的路径条件
- $$c_1 \wedge c_2 \wedge \cdots \wedge c_n \rightarrow \quad P \wedge c_1 \wedge c_2 \wedge \cdots \wedge c_n$$
- 合取新路径条件与R的补可得如下所示的判定公式

$$P \wedge c_1 \wedge c_2 \wedge \cdots \wedge c_n \wedge \neg R$$

- 求解上述判定公式，如果可满足，则说明存在内存越界

符号执行技术实用化的主要障碍

- 路径爆炸问题
- 约束困顿问题
- 环境交互问题
- 基于生成的符号执行之符号注入
- 基于生成的符号执行之约束爆炸
- 基于变异的符号执行之约束收集

路径爆炸

杨子之邻人亡羊,
杨子曰：“嘻！亡
邻人曰：“多歧路。

约束困顿

- 从理论上讲，一阶谓词逻辑公式的可满足性是半可判定的，但是对于实际中的很多问题实例来说，高效率的约束求解工具却可以在用户可接受的时间内找到可满足解
- 随着软件规模越来越大，路径约束中的约束条件数量通常越来越多，因此需要的求解时间也越来越长
- 随着软件的内部结构越来越复杂，路径约束也越来越复杂，甚至超出了约束求解器的求解能力（约束求解器困顿）

环境交互

- 软件系统是一个多层的堆栈结构，其中每层均可看作一个模块。这些模块通常包括最上层的应用程序，中间层的C库（以及其它的第三方库），以及最下层的操作系统内核（实现系统调用）
- 应用程序功能的实现，多数情况下需要依赖C库（以及其它的第三方库）或者操作系统内核提供支持，这就不可避免要进行外部函数调用
- 程序调用外部函数后产生的效果通常是难以准确描述的，因此对环境交互问题的处理方式，直接影响到包括符号执行在内的绝大多数程序分析方法的实用性和准确性

符号注入与约束爆炸

- 基于生成的符号执行根据符号分支条件（具体分支条件要么永真，要么永假，不存在既可真又可假的情况）创建新符号进程，因此要解决好符号注入问题
- 在每个分支处，基于生成的符号执行都需要调用一次或者多次求解器，这是一个耗时的工作（约束爆炸）；在某些极端的情况下，一个简单的与符号相关的循环结构甚至都可能造成约束困顿

约束收集

- 基于变异的符号执行中，指令是运行在真实机器的CPU上的机器指令
- 如何恢复数量繁多的条件跳转指令依赖的各种各样的条件表达式，并容易地转换为主流约束求解器接收的格式，不仅面临巨大的工程挑战，且其可靠性值得怀疑

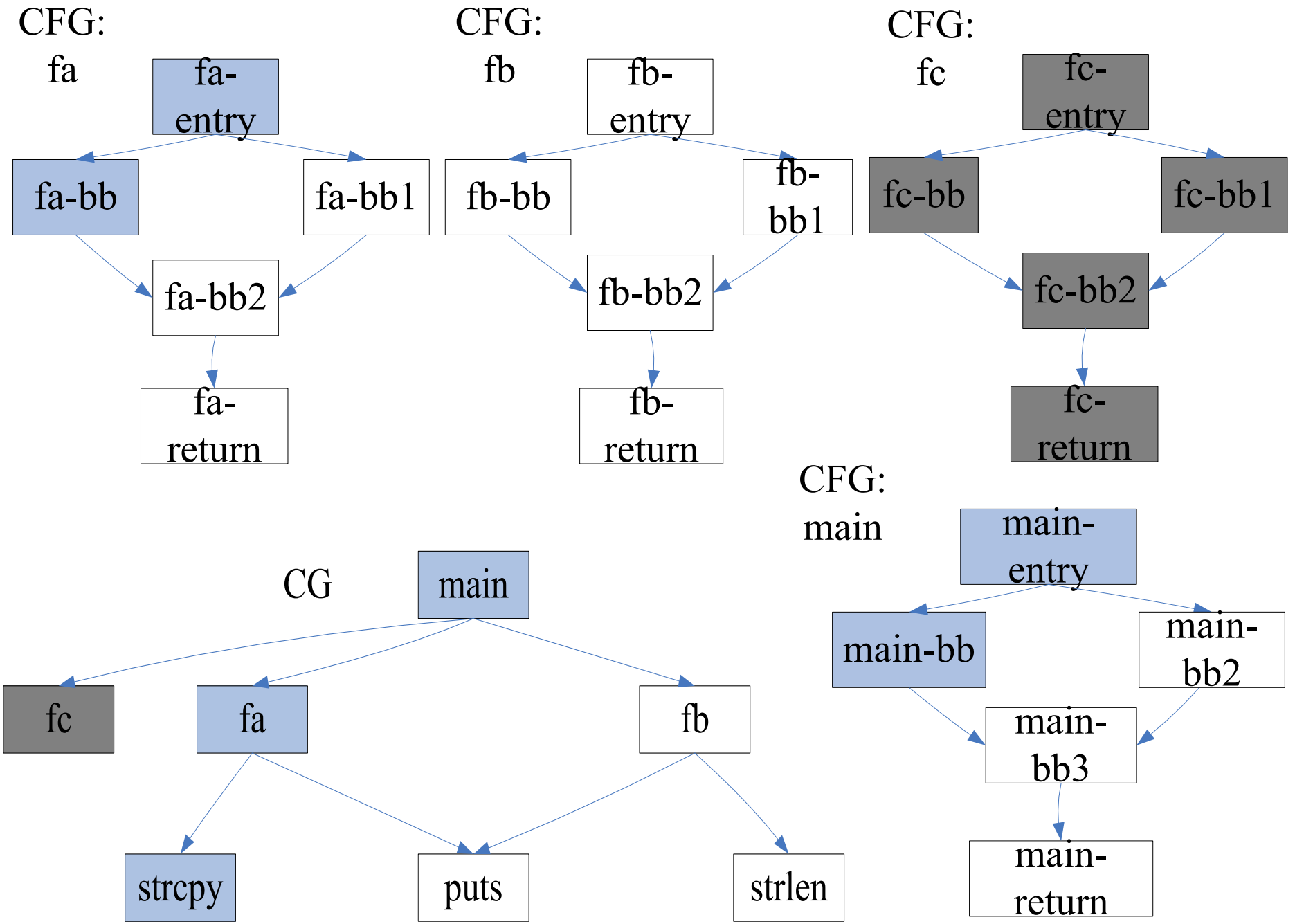
缓解路径爆炸问题之循环控制

- 路径爆炸问题的一个重要原因在于循环可能产生大量的新路径，在极端情况下，新生成的路径可能是无限的
  - 01 void *memset(void *s, int c, size_t n){
  - 02 unsigned char *p = (unsigned char *) s;
  - 03 while(n--) *p++ = (unsigned char) c;
  - 04 return s;
  - 05 }
- 循环控制的思想非常直接，通过减少与符号相关的循环可以创建的路径数，可以一定程度上缓解路径爆炸问题
  - 阈值控制
  - 2的幂次数控制

缓解路径爆炸问题之无关路径剪除

- 路径爆炸问题的另一个重要原因是程序的所有路径中，仅有一部分与特定的程序分析相关
- 无关路径剪除的思想是不探测与目标无关的路径
  - 首先使用污点分析着色所有感兴趣的基本块
    - 选取对字符串操作函数的调用点作为sink
    - 标记sink点的前驱基本块
    - 标记路径上的函数内部的基本块
  - 随后在符号执行过程中剪除执行到未着色基本块的路径

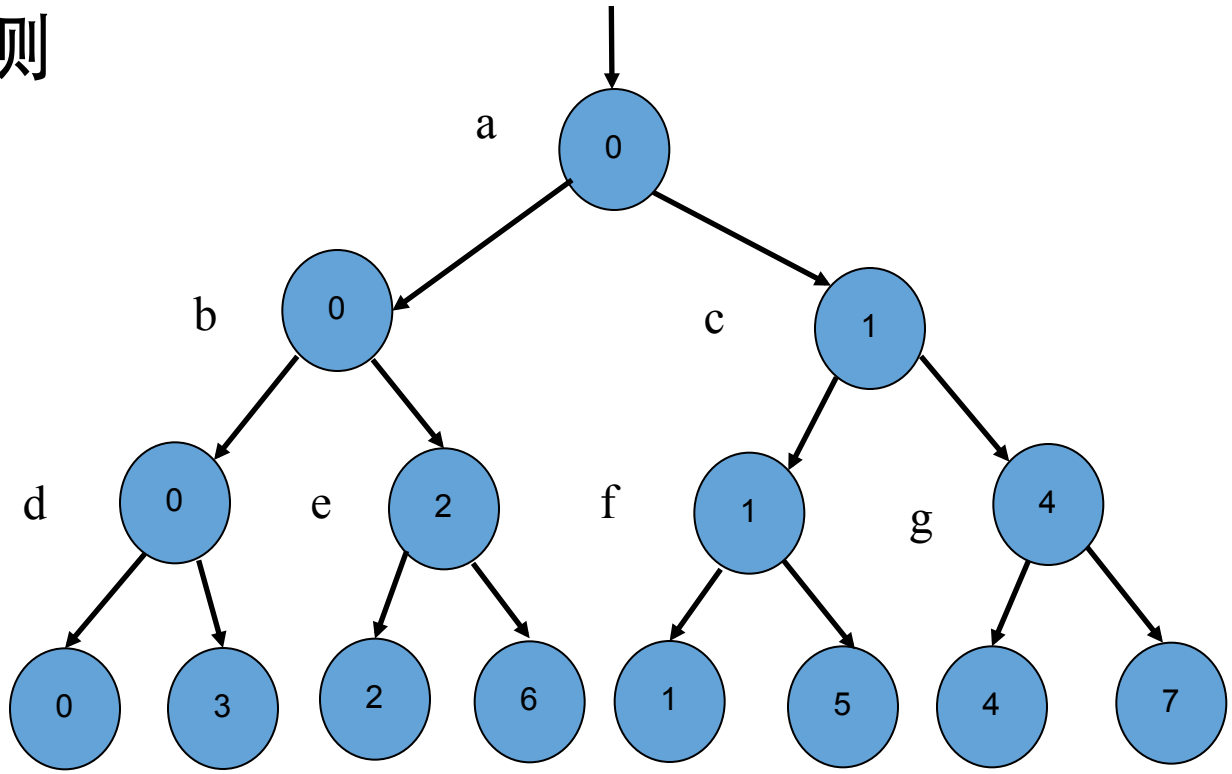
```
01 int fc(int c) {          19   return m;
02   if(c > 0)              20 }
03     return c + 1;        21 int fa(int a, char* sz) {
04   else                   22   char s[8];
05     return c - 1;        23   if(0== a % 2)
06 }                        24     strcpy(s, sz);
07 int main(){              25   else
08   int n, m;              26     puts("fa ");
09   char s[16];            24   return a;
10   make_symbolic(&n);     25 }
11   make_symbolic(s);      26 int fb(int b, char* sz) {
12   assume(s[15] == '0'); 27   int l = strlen(sz);
13   if(n < 32) {           28   if(l < 8)
14     n = fc(n);           29     puts("strlen(sz)<8");
15     m = fa(n, s);        30   else
16   }                      35     puts("strlen(sz)>8");
17   else                   31   return l;
18     m = fb(n, s);        36 }
```


CFG:
fa
fa-entry
fa-bb
fa-bb1
fa-bb2
fa-return
CFG:
fb
fb-entry
fb-bb
fb-bb1
fb-bb2
fb-return
CFG:
fc
fc-entry
fc-bb
fc-bb1
fc-bb2
fc-return
CG
main
fc
fa
fb
strcpy
puts
strlen
CFG:
main
main-entry
main-bb
main-bb2
main-bb3
main-return

缓解路径爆炸问题之路径选择

- 经过循环控制和无关路径剪除，需要探测的路径数目已经大大减少，但是如果期望在可接受时间内尽早检测到漏洞，则还需要某种路径选择策略，即如何能够尽可能早地遍历可能触发漏洞的路径
- 典型的路径选择策略：
  - 随机选择
  - 深度优先
  - 广度优先
  - 先生成的路径先探测
  - ……

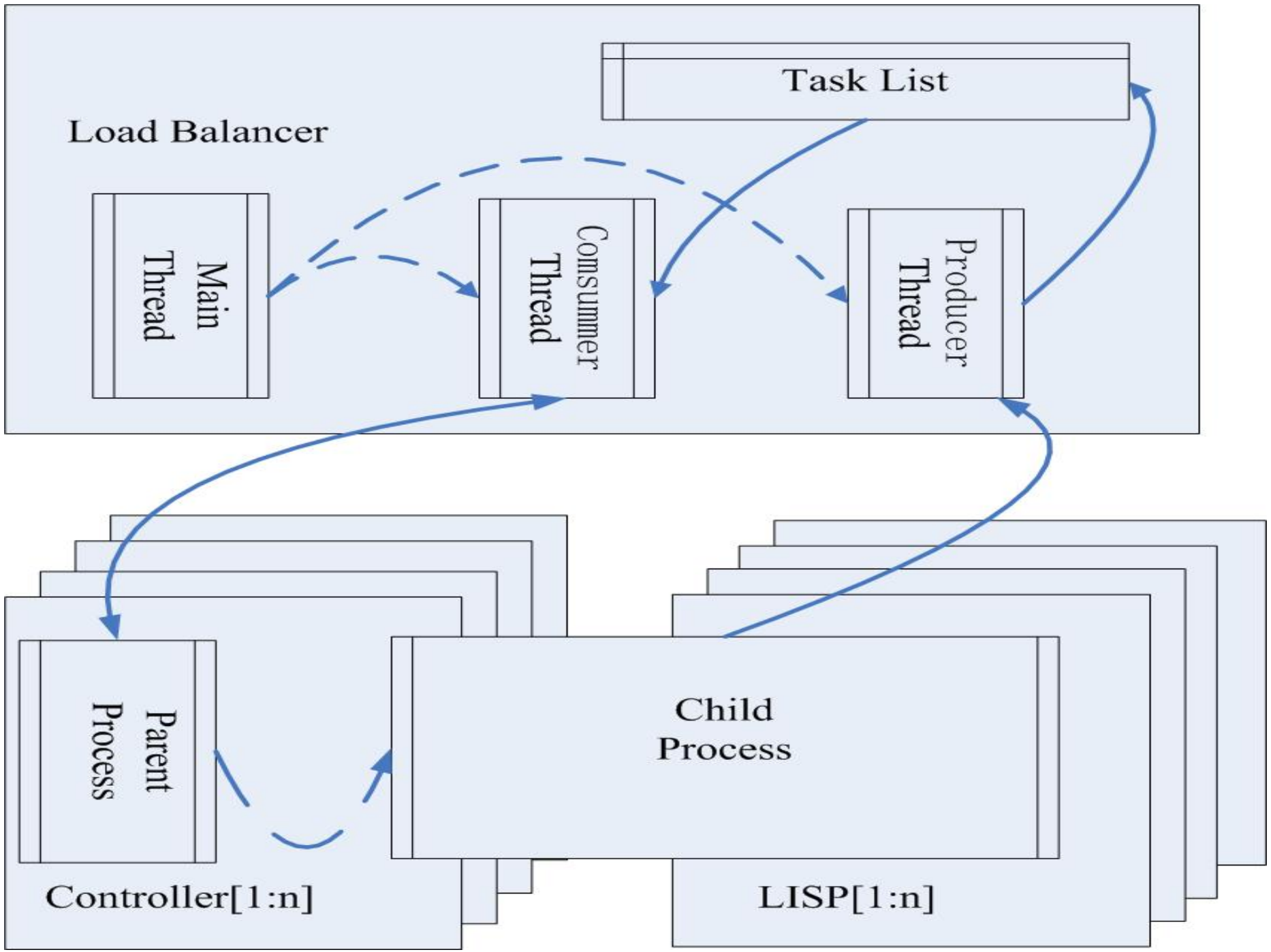
先生成的路径先探测



- 派生于广度优先和深度优先遍历策略
- 保证当前正在探测的路径尽量执行
- 保证所有路径整体向前推进

缓解路径爆炸之并行化符号执行

- 并行化符号执行包括3个组件
  - 负载均衡器，控制器，符号执行器
- 负载均衡器的消费者线程根据控制器的状态向其派发任务
- 每个控制器每次启动一个符号执行器符号地执行一条路径，并将结果返回给负载均衡器
- 在符号执行期间，符号执行器分解一个任务为一组任务并上传新生成的任务
- 负载均衡器的生产者线程负责收集符号执行器上传的任务


Load Balancer
Task List
Main Thread
Comsummer Thread
Producer Thread
Parent Process
Child Process
Controller[1:n]
LISP[1:n]

解决环境交互问题之环境建模

- 通过对C库和系统调用函数建模构建环境建模库，在分析开始前向待分析程序导入环境建模库可消除其对外部环境的依赖性
- 使用一个修改版本的uclibc库来模拟libc库，可使得待分析程序的外部函数调用范围缩小到操作系统调用
- 针对文件操作，根据用户提供的参数文件名判断文件是否是符号文件，具体文件由符号执行引擎负责读入待分析程序的内存空间，符号文件由符号文件环境负责维护
- 针对网络操作，尤其是网络输入函数，可以通过拦截对这些函数的调用，并使用环境建模库中的函数取代对实际函数的调用
- 对于read、write以及select等可同时被文件操作和网络操作使用的系统调用，处理的时候还需要对操作的对象进行判断，并针对不同的操作对象，采用不同的处理方式

各种符号注入方法

- 通过特殊标记可将任意的内存对象指定为符号
- 源代码改装
  - 操作源代码
- 静态字节码改装
  - 类似于源代码改装，操作字节码
- 动态字节码改装
  - 基于编译器Link机制对多个字节码模块进行链接
- 预加载机制
  - PRE_LOAD

缓解约束爆炸之符号变量常量化

- 调用求解器是一个耗时的工作，在某些极端的情况下，一个简单的与符号相关的循环结构甚至都可能造成约束困顿
  - 01 p = malloc(symlen);
  - 02 if(!p) return;
  - 03 memset(p, 0, symlen);
  - 04 void *memset(void *s, int c, size_t n){
  - 05 unsigned char *p = (unsigned char *) s;
  - 06 while(n--) *p++ = (unsigned char) c;
  - 07 return s;
  - 08 }
- 在符号执行过程中，通过使用一定规则对某些符号值计算唯一的常值，并使用该常值取代符号值，可有效遏制约束爆炸（例如：03行的symlen有唯一常值）

缓解约束爆炸之符号文件常量化

- 不使用符号文件模板，那么整个符号文件的内容都是符号，这对约束求解器的压力是相当大的
- 通过修改一个具体文件中特定的字段为预定义的值，即生成一个符号文件模版
- 当符号执行引擎读写加载了符号文件模版的符号文件时，由于只有其中的几个字段是符号，因此极大地缓解了约束求解器的压力

实验

program	file	function	line	Vulnerability-ID	KLEE
gzip	unlzh.c	make_table	152	CVE-2006-4335	N
gzip	unpack.c	build_tree	172	CVE-2006-4336	N
gzip	unlzh.c	make_table	182	CVE-2006-4337	N
gzip	unlzh.c	read_pt_len	228	*	N
minigzip	minigzip.c	file_compress	201	Bugtraq 22964	N
minigzip	minigzip.c	file_uncompress	232	*	N
zip	zip.c	main	697	CVE-2004-1010	Y
ncompress	compress42.c	comprexx	886	CVE-2001-1413	Y
zoo	parse.c	parse	42	Bugtraq 17126	N
rgb2ycbcr	rgb2ycbcr.c	tiffcvt	284	CVE-2009-2347	N
tiff2rgba	tiff2rgba.c	cvt_whole_image	338	CVE-2009-2347	N
htget	htget.c	ProcessURL	842	CVE-2004-0852	
iwconfig	iwconfig.c	get_info	67	CVE-2003-0947	

结论

- 目前，符号执行方法在理论上已经基本成熟，而阻碍其实用化的主要问题包括路径爆炸问题、约束困顿问题和环境交互问题等
- 上述两种符号执行方法都还有其特有的困难
  - 基于生成的符号执行方法面临的其它问题还有符号注入问题和约束爆炸问题
  - 基于变异的符号执行方法面临的另一个重要问题是约束收集问题
- 针对这些问题，实现有效的解决或者缓解方法，对于符号执行技术的实用化具有重要的意义

巨人背后的专家

谢谢

NSFOCUS

RSA CONFERENCE CHINA 2012
RSA信息安全大会2012