

第五、六、七、八章 要点

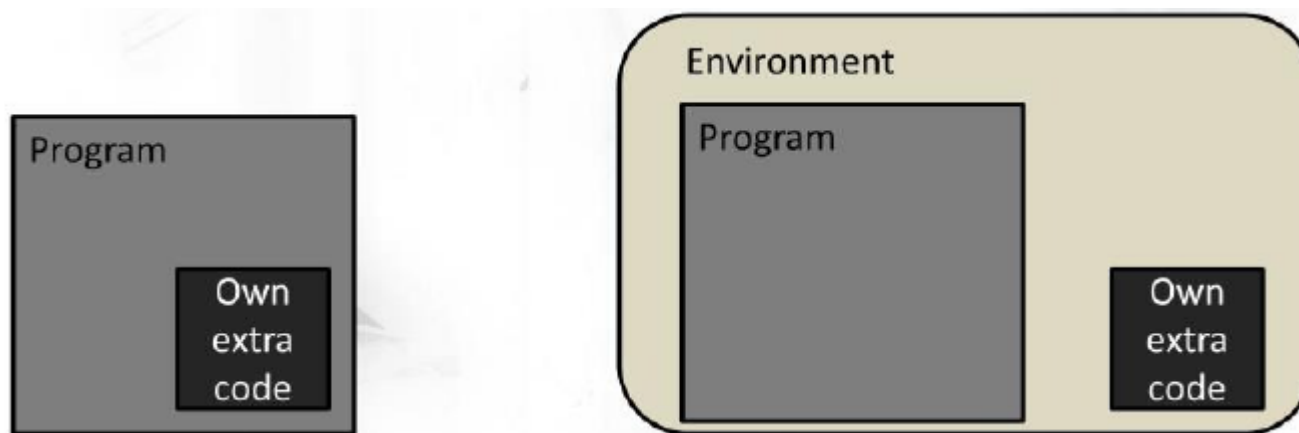
蔡彦

计算机科学国家重点实验室
中国科学院软件研究所

yancai@ios.ac.cn

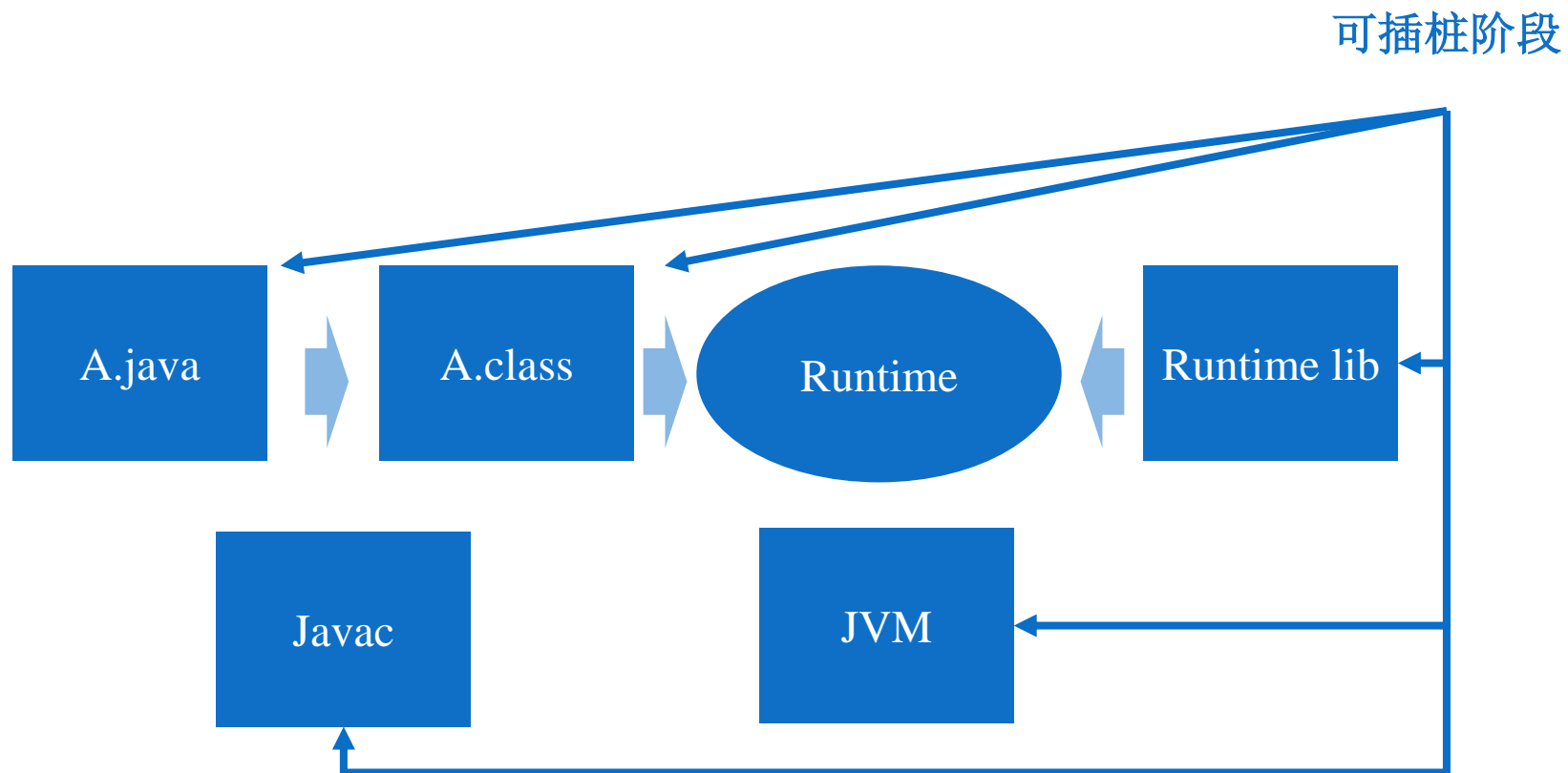
1.程序插桩技术

- 基本概念
 - **插桩 (Instrumentation)** 是一种用于动态软件测试等领域方法
 - 向目标程序或运行环境插入特定的代码片段 (称为**桩**、**探针**)
 - 程序运行时通过探针的执行获取程序关键信息



分类

- 按照插桩目标
 - 直接插桩
 - 间接插桩（解释器）
- 按照插桩阶段
 - 静态插桩：
 - 动态插桩
 - 即时模式
 - 解释模式
 - 探测模式
- 按插桩粒度
 - 指令（语句）级
 - 基本块级
 - 函数级等



1.2 常用插桩工具

- 源代码（字节码）插桩工具：
 - C/C++： Clang/LLVM
 - Java： ASM, WALA, Soot
 - Android： Soot, Dexlib2
- 二进制插桩工具： Qemu, Pintool, DynamoRIO

插桩工具比较

	GCC	Clang	LLVM	ASM	WALA	SOOT	Dexlib2	QEMU	Pintool	DynamoRIO
语言	C/C++		IR	Java			Dex	二进制		
插桩级别	源码		中间码	字节码		中间码		二进制		
成熟度	高								中	
难易度	难								中	难
静态分析	有			无	有		无			
	GCC	Clang	LLVM	ASM	WALA	SOOT	Dexlib2	QEMU	Pintool	DynamoRIO

对机器模型的知识

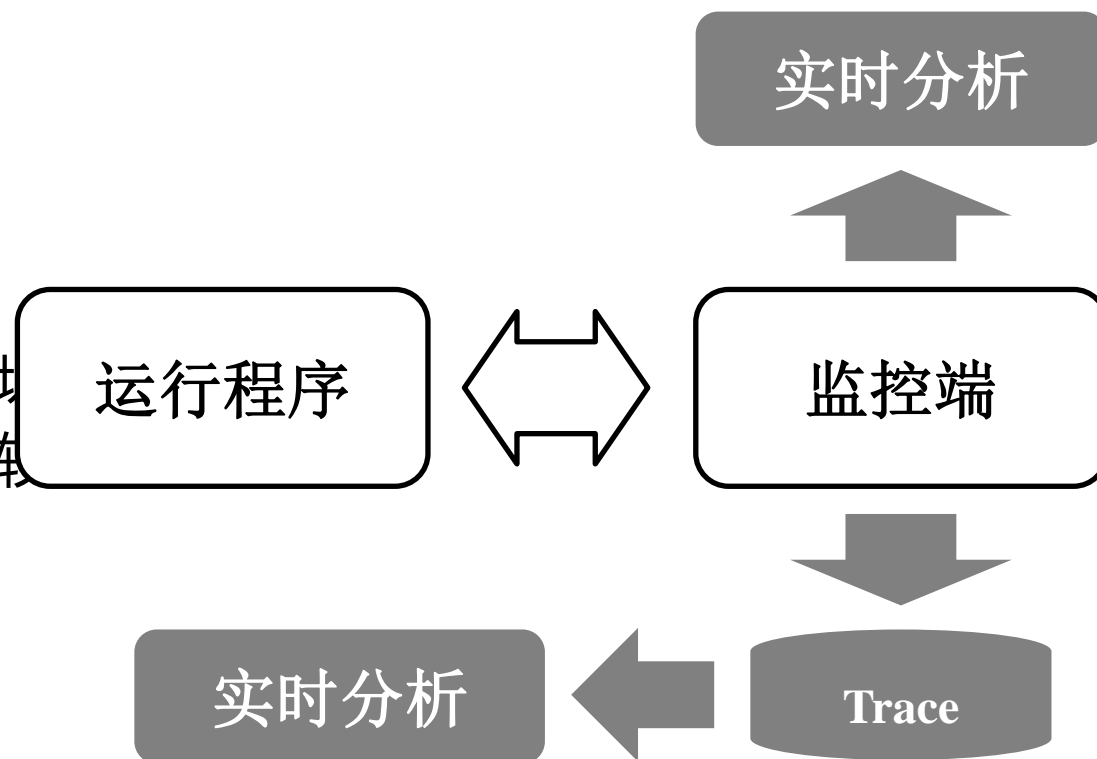
运行时监控

- 运行时监控/验证/检查 (Runtime Monitoring/Verification/Checking)
- Runtime Monitoring: a lightweight and **dynamic verification** technique that involves observing the internal operations of a software system and/or its interactions with other external entities, with the aim of determining whether the system **satisfies** or **violates** a correctness specification.
Cassar, Ian & Francalanza, Adrian & Aceto, Luca & Ingólfssdóttir, Anna. (2017). A Survey of Runtime Monitoring Instrumentation Techniques. Electronic Proceedings in Theoretical Computer Science. 254. 15-28. 10.4204/EPTCS.254.2.
- Runtime Verification: a computing system **analysis** and **execution based** on extracting information from a running system and using it to detect and possibly react to observed behaviors **satisfying** or **violating** certain properties.

运行时监控

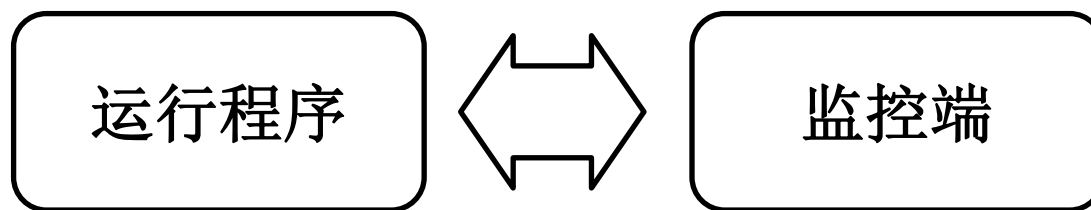
运行时/在线监控 vs 离线监控 (Runtime/Online vs Offline)

- 均需要记录程序运行 (trace)
- 离线监控
 - 需要完整 (连续) trace分析才可以的场
 - 运行时仅记录trace, 对程序运行影响较
- 在线监控
 - 仅需部分 (离散) trace分析的场景
 - 运行时无需显式记录trace,
 - 可能需要主动干扰程序运行



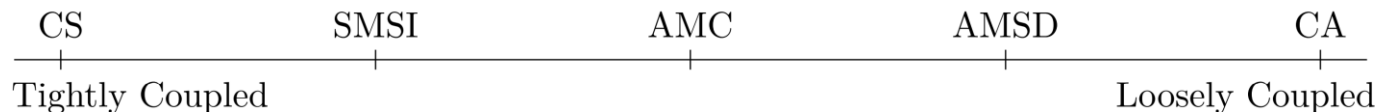
运行时监控

常见监控方法



已有的研究中，根据被监控程序和监控端的耦合关系

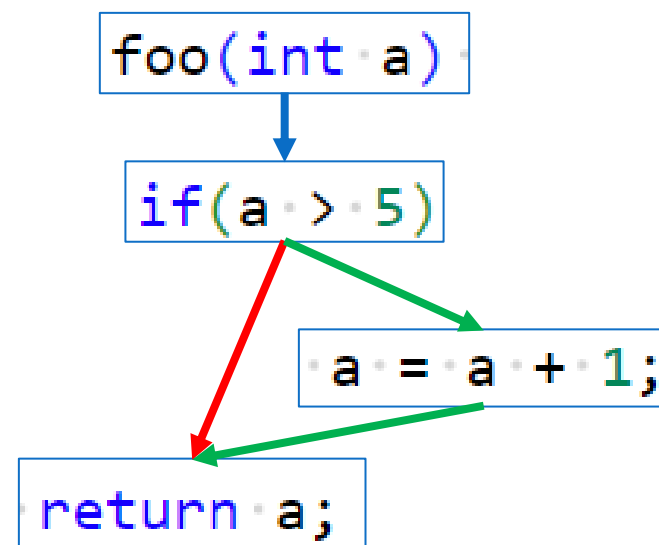
- 全局同步监控（CS: Completely Synchronous Monitoring）
- 同步分析同步监控（SMSI: Synchronous Monitoring with Synchronous Instrumentation）
- 带检查点的异步监控（AMS: Asynchronous Monitoring with Checkpoints）
- 同步检测异步监控（AMSD: Asynchronous Monitoring with Synchronous Detection）
- 全局异步监控（CA: Completely Asynchronous Monitoring）



代码覆盖率的计算

- 分支覆盖率
 - 执行到的分支数量与总分支数量的比例
 - 考虑每个分支判定条件为true或false的情况，即使没有else分支
 - 但不会考虑判定条件的子条件为true和false的情况
 - 下图中，若 $a = 6$ ，语句覆盖率为100%，而分支覆盖率为50%


```
int foo(int a) {  
    ... if(a > 5) {  
        ... a = a + 1;  
    ... }  
    ... return a;  
}
```



代码覆盖率的计算

- 条件覆盖率
 - 执行到的分支判定中每个子表达式的取值占所有可能取值的比例
 - 注意 或 操作时的短路问题

```
int foo(int a1, int a2) {  
    ... if(a1 || a2) {  
        ...  
    } else {  
        ...  
    }  
}
```



a1	T	F	F
a2	?	T	F

代码覆盖率的计算

- 实例

- 哪些测试用例能使右图代码覆盖率达到100%?
- 针对下面的测试用例计算其覆盖率
- Testcase1
 - $a = 6, b = 5$
- Testcase2
 - $a = 6, b = 5$
 - $a = 3, b = 11$
 - $a = 11, b = 11$
 - $a = 11, b = 13$

```
int foo(int a, int b) {  
    if (a < 10 || b < 10) {  
        return f1(a, b);  
    } else {  
        return f2(b, a);  
    }  
}
```

```
int f1(int a, int b) {  
    if (a * a > 25) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

```
int f2(int a, int b) {  
    if (a * a < 144) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

代码覆盖率的计算

- Testcase1
 - $a = 6, b = 5$
 - 语句覆盖率: 4/9
 - 分支覆盖率: 1/3
 - 条件覆盖率: 1/4

```
int foo(int a, int b) {  
    if (a < 10 || b < 10) {  
        return f1(a, b);  
    } else {  
        return f2(b, a);  
    }  
}
```

```
int f1(int a, int b) {  
    if (a * a > 25) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

```
int f2(int a, int b) {  
    if (a * a < 144) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

代码覆盖率的计算

- Testcase2
 - a = 6, b = 5
 - a = 3, b = 11
 - a = 11, b = 11
 - a = 11, b = 13
 - 语句覆盖率: 100%
 - 分支覆盖率: 100%
 - 条件覆盖率: 7/8
 - 如何修改测试用例使得条件覆盖率为100%?

```
int foo(int a, int b) {  
    if (a < 10 || b < 10) {  
        return f1(a, b);  
    } else {  
        return f2(b, a);  
    }  
}
```

```
int f1(int a, int b) {  
    if (a * a > 25) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

```
int f2(int a, int b) {  
    if (a * a < 144) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

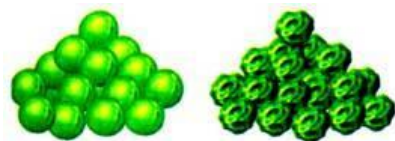
错误定位

- 关键点

程序中的 Faults (root cause)一般并不和failure强相关

- 1、Fault 一般隐藏很深
- 2、统计定位中噪音大量存在
- 3、是否有最好的基于统计的比较方法？
- 4、Faults并非一定会导致failure

背景——从生物变异开始



豌豆的圆粒与皱粒



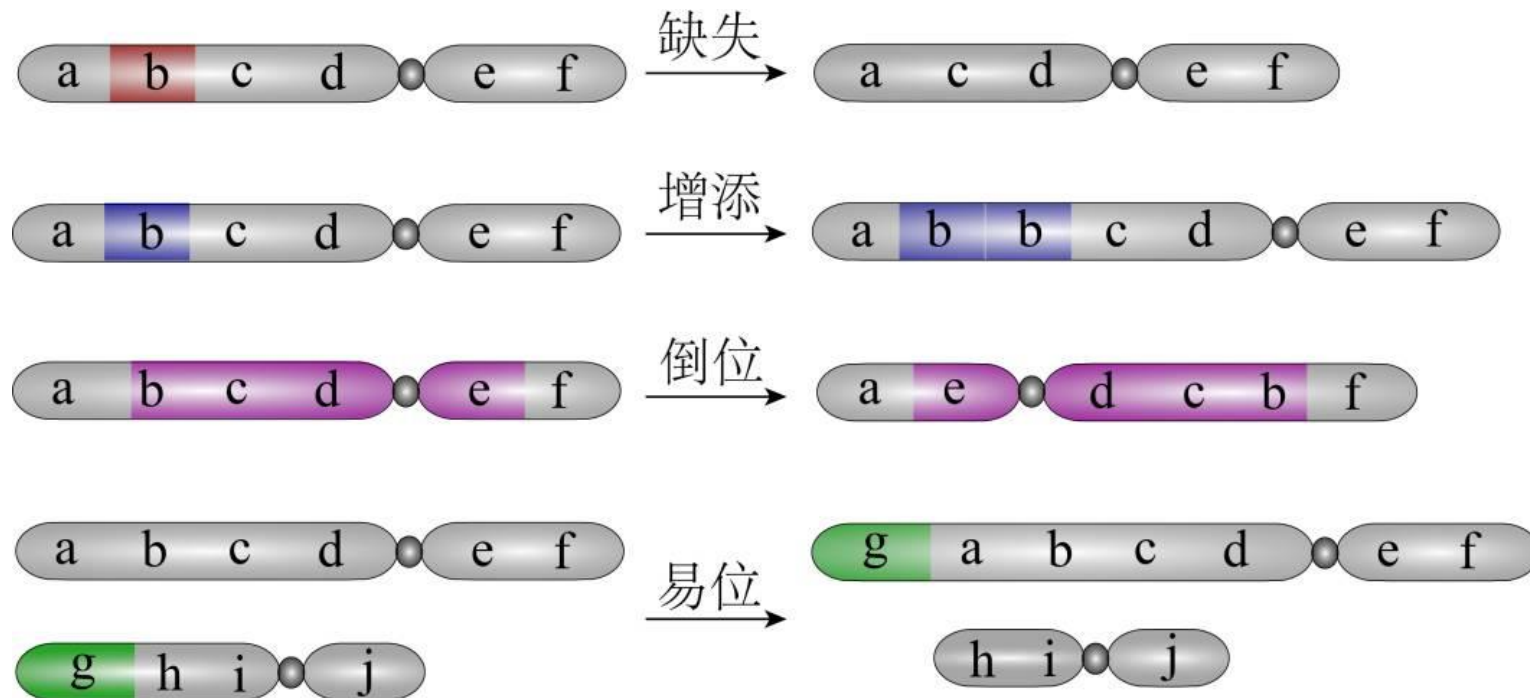
番茄的红果与黄果



兔的白毛与黑毛



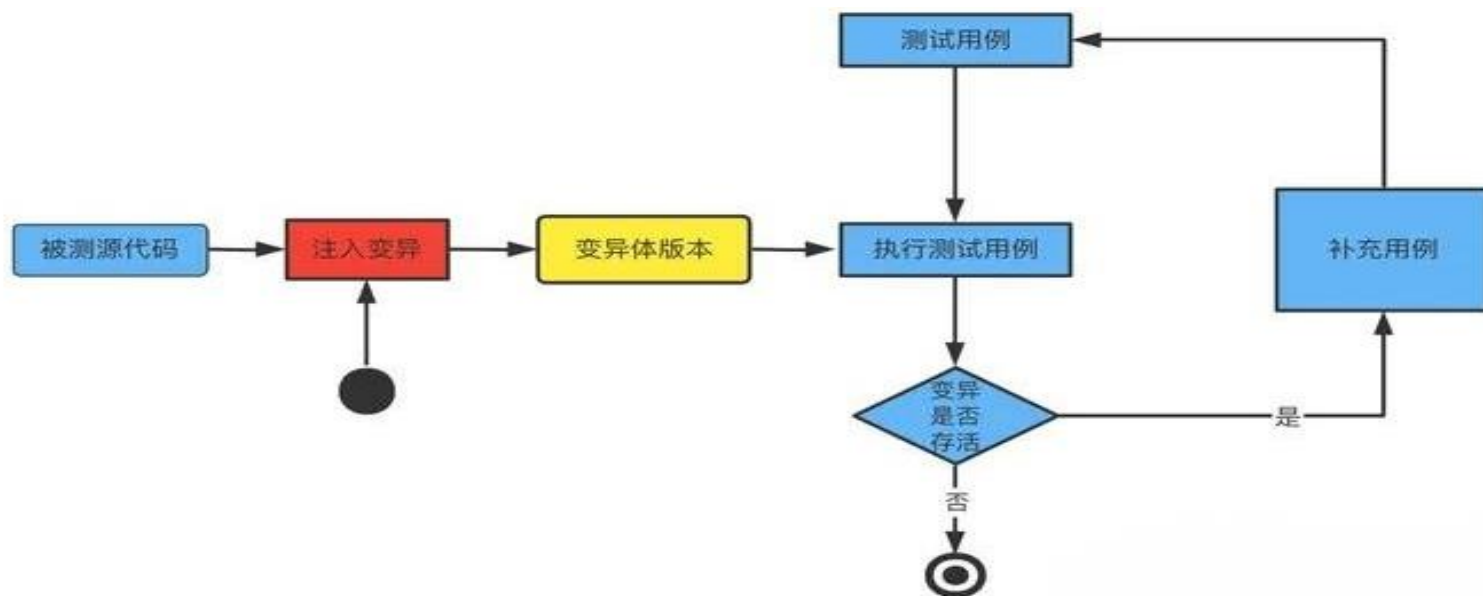
鸡的玫瑰冠与单冠



软件测试→变异测试

背景——基本思想

- 变异测试是一种基于故障注入的测试技术，将错误代码插入到被测代码中，以验证当前测试用例是否可以发现注入的错误。该测试手段理论上属于白盒测试范畴。
- 变异测试的主要目的是为了验证测试用例的有效性，在注入变异后，测试用例能发现该错误，则表明用例有效的；反之，表明测试用例是无效的，需要补充该变异的测试用例。
- 变异测试有助于评估测试用例的质量，以帮助测试人员编写更有效的测试用例。测试人员设计的测试用例发现的变异体越多，表明其设计的测试用例质量就越高。

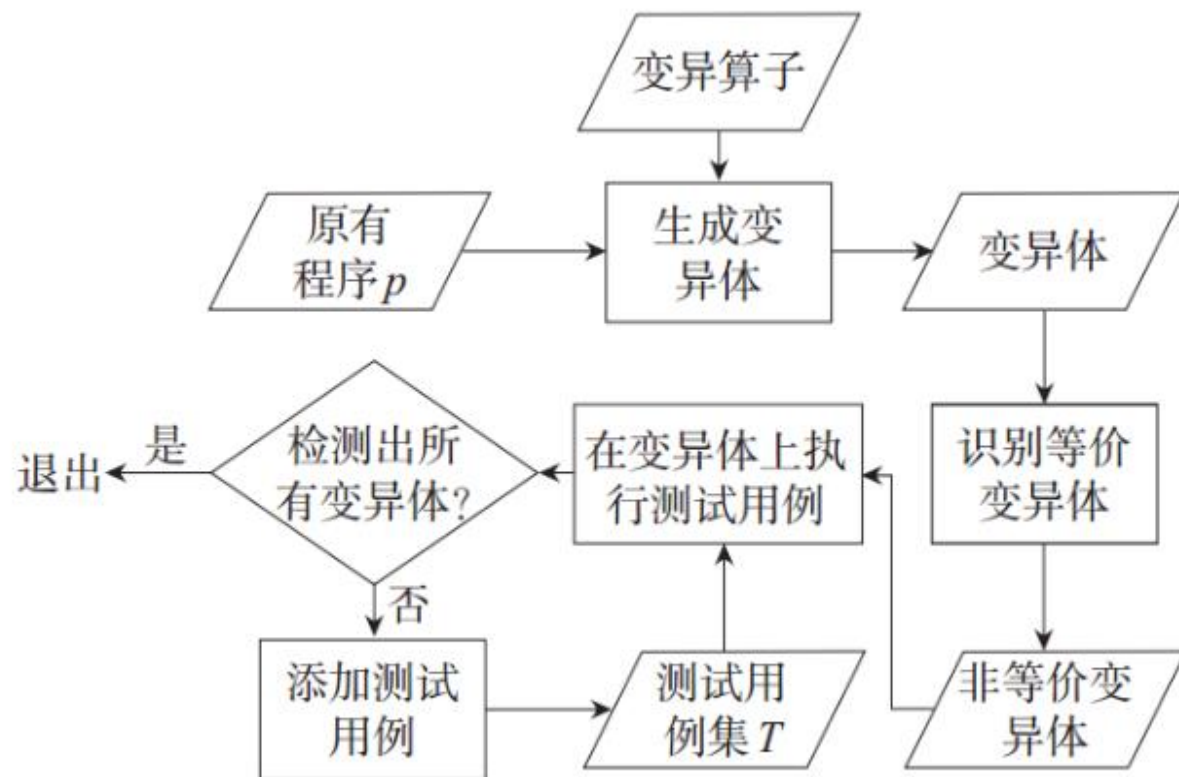


背景——基本假设

- 两大基本假设：变异测试旨在找出有效的测试用例，发现程序中真正的错误。在一个软件中，潜在BUG的数量是巨大的，通过生成突变体来全面覆盖所有的错误是不可行的。所以，传统的变异测试旨在寻找这些错误的子集，能尽量充分地近似描述这些BUG。于是，变异测试的理论基于两条假设：Competent Programmer Hypothesis (CPH) 和 Coupling Effect(CE)。
 - ❑ CPH：假设编程人员是有能力的，他们尽力去更好地开发程序，达到正确可行的结果，而不是搞破坏。它关注的是程序员的行为和意图。
 - ❑ CE：(耦合效应)更加关注在变异测试中错误的类别。一个简单的错误产生往往是由于一个单一的变异（例如句法错误），而一个庞大复杂的错误往往是由于多出变异所导致。复杂变异体往往是由诸多简单变异体组合而成。

背景——概念定义

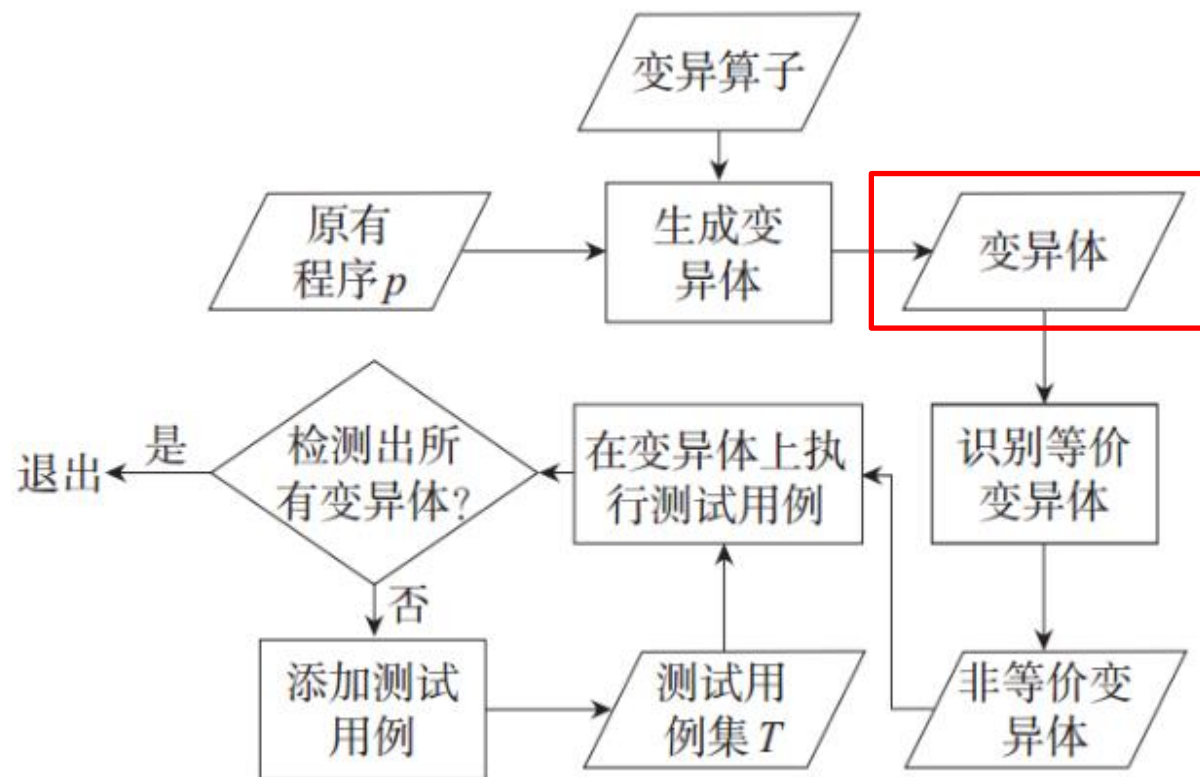
给定一个程序 P 和一个测试数据集 T ，通过**变异算子**为 P 产生一组**变异体** M_i （合乎语法的变更），对 P 和 M 都使用 T 进行测试运行，如果某 M_i 在某个测试输入 t 上与 P 产生不同的结果，则该 M_i 被**杀死**；若某 M_i 在所有的测试数据集上都与 P 产生相同的结果，则称其为**活的变异体**。接下来对活的变异体进行分析，检查其是否**等价**于 P ；对不等价于 P 的变异体 M 进行进一步的测试，直到**充分性度量**达到满意的程度。



原理——变异体

在完成变异算子设计后，通过在原有被测程序上执行变异算子可以生成大量变异体 M ，在变异测试中，变异体一般被视为含缺陷程序。

根据执行变异算子的次数，可以将变异体分为一阶变异体和高阶变异体等。



变异测试应用——case1

C语言源程序P:

```
1. int foo (int x, int y){  
2.     return (x-y);  
3. }
```

return (x+y);

- case1: 考虑左侧这个程序实例
该程序本身的目的应该是计算并返回两数之和，那么显然此时的代码是错误的；

假设foo已经由测试集合T测试通过，T包含以下两个测试用例：

{ t1: <x=1, y=0>, t2: <x=-1, y=0> }

注意：foo在每一个测试用例上都返回了正确的期望值，而且对于基于控制流和数据流的充分性准则来说T是充分的。

变异测试应用——case1

C语言源程序P:

```
1. int foo (int x, int y){  
2.     return (x-y);  
3. }
```

return (x+y);



- case1: 考虑左侧这个程序实例
该程序本身的目的应该是计算并返回两数之和，那么显然此时的代码是错误的；

考虑foo生成的三个变异体:

变异体M1:

```
1. int foo (int x, int y){  
2.     return (x+y);  
3. }
```

变异体M2:

```
1. int foo (int x, int y){  
2.     return (x-0);  
3. }
```

变异体M3:

```
1. int foo (int x, int y){  
2.     return (0+y);  
3. }
```

变异测试应用——case1

考虑foo生成的三个变异体:

变异体M1:

```
1. int foo (int x, int y){
2.     return (x+y);
3. }
```

变异体M2:

```
1. int foo (int x, int y){
2.     return (x-0);
3. }
```

变异体M3:

```
1. int foo (int x, int y){
2.     return (0+y);
3. }
```

使用T执行每个变异体，直到变异体被杀死或执行完所有的测试。

T: {t1: <x=1, y=0>, t2: <x=-1, y=0>}

Test(t)	foo(t)	M1(t)	M2(t)	M3(t)
t1	1	1	1	0
t2	-1	-1	-1	0
		LIVE	LIVE	KILLED

执行完所有的变异体后，2个存活变异体，1个被杀死，计算变异数需要对所有活的变异体判断等价。

变异测试应用——case1

考虑源程序P和两个存活的变异体：

变异体M1:

```
1. int foo (int x, int y){  
2.     return (x+y);  
3. }
```

变异体M2:

```
1. int foo (int x, int y){  
2.     return (x-0);  
3. }
```

C语言源程序P:

```
1. int foo (int x, int y){  
2.     return (x-y);  
3. }
```

关注变异体M1:

一个测试数据如果能够区分M1和源程序P，其一定满足条件：

$$x-y \neq x+y \quad \text{即} \quad y \neq 0$$

于是产生新的一组测试数据：t3: <x=1, y=1>

使用t3执行foo得到foo(t3)=0，然而根据需求应该得到foo(t3)=2. 因此，t3使得M1与源程序P产生区别，同时发现了错误。

变异测试成本——弱变异

- 传统的变异测试系统如Mothra比较的是原始程序和变异体的最终输出结果，需要完全执行完整整个程序，因此，被称为**强变异(strong mutation)**。
- 弱变异**由于避免了完全执行整个程序，从而节省了计算开销，提高了变异测试的效率，改进了变异测试实用性。
- 主要思想：判断杀死变异体时比较的是原始程序和变异体的内部状态，即比较点紧接着程序的变异部分之后，只需运行部分语句，不需要执行完整整个程序。弱变异有四个比较点：
 - 1) 表达式级比较点(expression weak)，在对变异部分之后最内层表达式第一次求值时进行比较。
 - 2) 语句级比较点(statement weak)，在第一次执行完变异语句之后进行比较。
 - 3) 基本块级比较点1(basic block weak / 1)，在第一次执行完包含变异语句的基本块之后进行比较。
 - 4) 基本块级比较点2(basic block weak / n)，这种情况主要针对于带有循环的变异，在第一次循环时还无法将变异体杀死。因此，需要在每次执行完包含变异语句的基本块后都进行比较，直到将变异体杀死。

2.技术原理

- 基本原理：
 - 模糊测试（fuzz testing, fuzzing）是一种软件测试技术
 - 核心思想
 - 将自动或半自动生成的随机数据输入到一个程序中
 - 监视程序异常，如崩溃，断言（assertion）失败
 - 以发现可能的程序错误
 - 常常用于检测软件或计算机系统的安全漏洞

2.1 基本原理

- 模糊测试的分类
 - 黑盒测试 (Black-box)
 - 不关心程序内部逻辑，只观察程序行为（输入-输出对应关系）。将程序视为黑盒。
 - 代表工作：Peach 2004
 - 白盒测试
 - 通过分析程序内部逻辑，系统性地探索程序行为，通常要根据程序语义进行推理
 - 符号执行 (Symbolic Execution)、动态符号执行 (Concolic Testing)
 - 代表工作：KLEE 2008, SAGE 2008

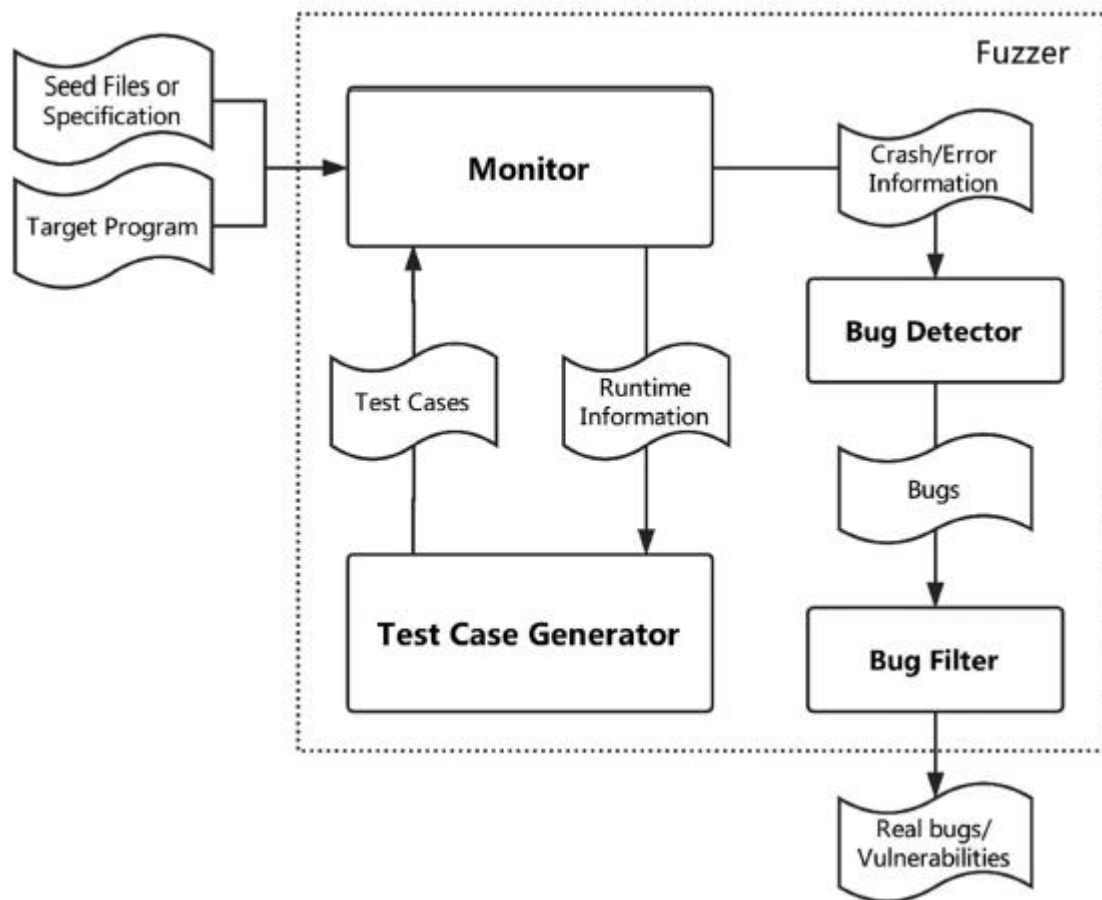
2.1 基本原理

– 灰盒测试

- 通过插桩等方式获取程序的部分内部信息，将其作为反馈来指导输入的生成
- 不对程序的完整语义进行推理
- 代表工作：AFL 2013, libfuzzer 2015

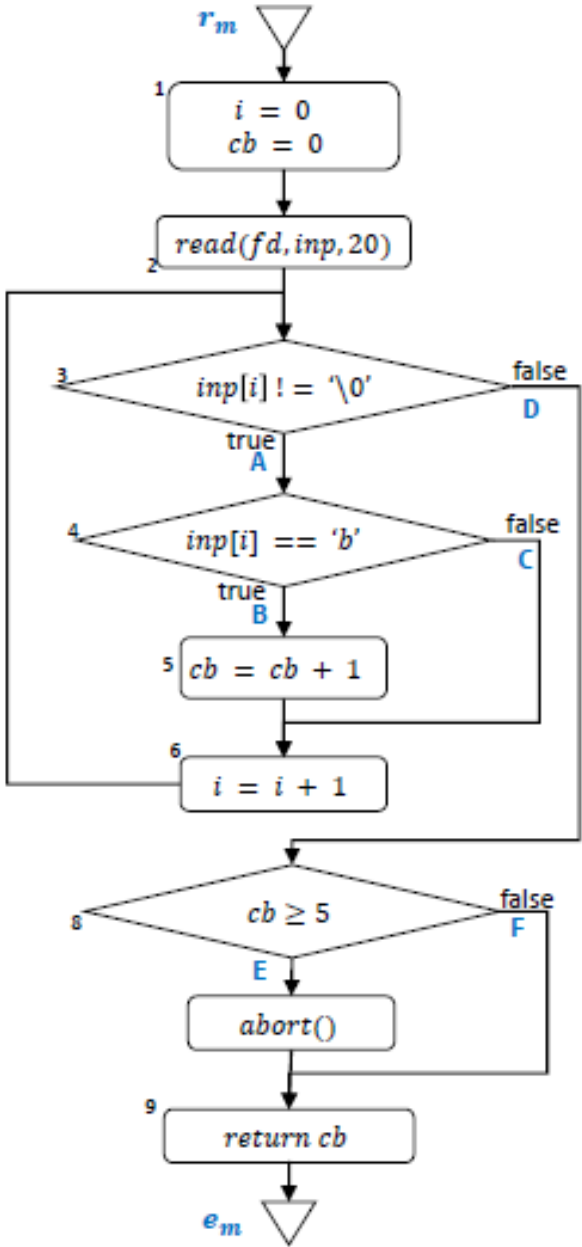
2.1 基本原理

- 模糊测试的基本过程：



Grey-box fuzzing – Working example

① "a"

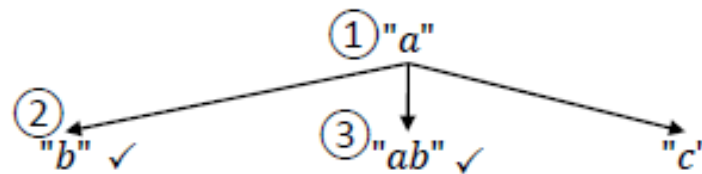
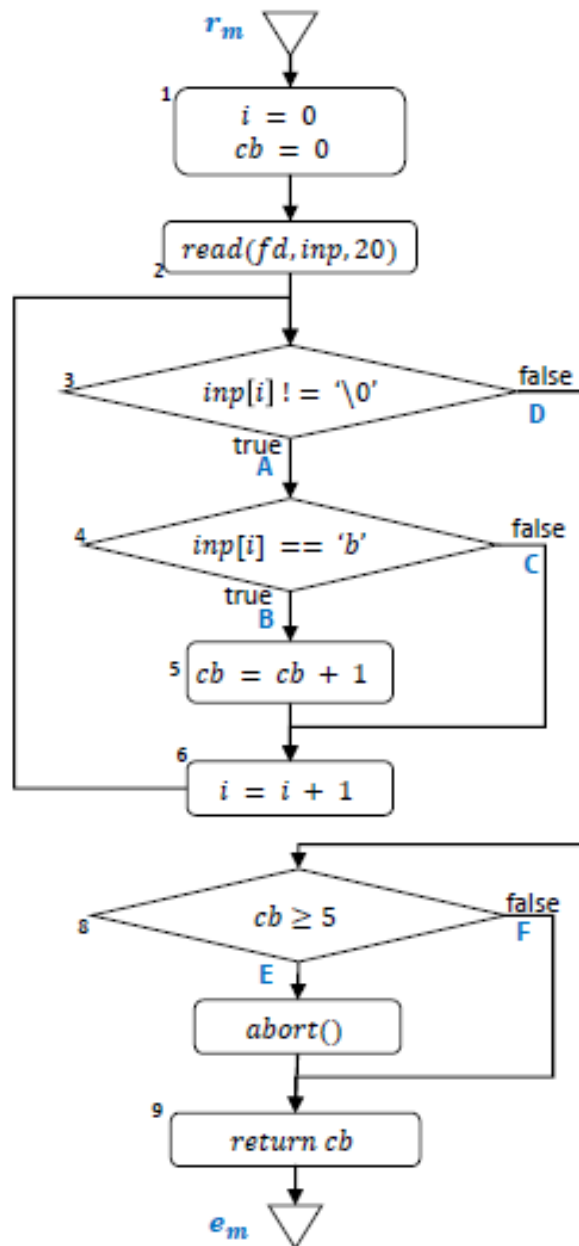


Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1

来源:

https://z14120902.github.io/teaching/ict_alg/fuzz_mc.pdf

Grey-box fuzzing – Working example

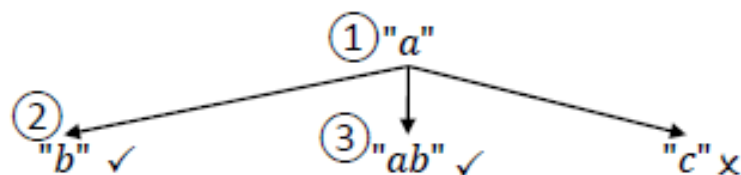
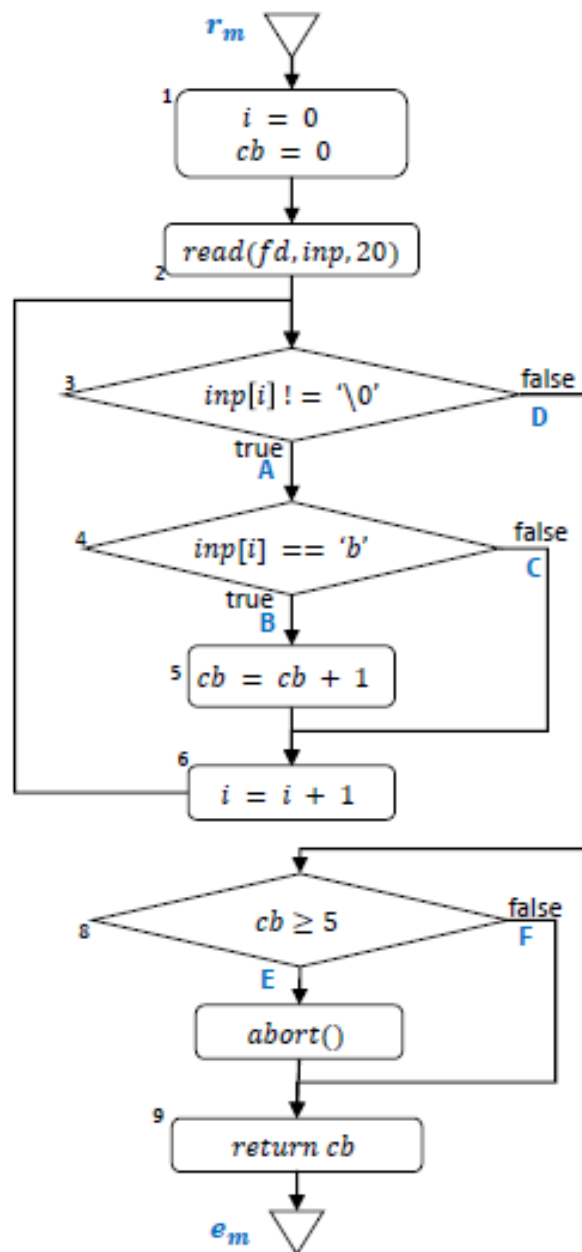


Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
	"c"		1				1		1

来源:

https://z14120902.github.io/teaching/ict_alg/fuzz_mc.pdf

Grey-box fuzzing – Working example

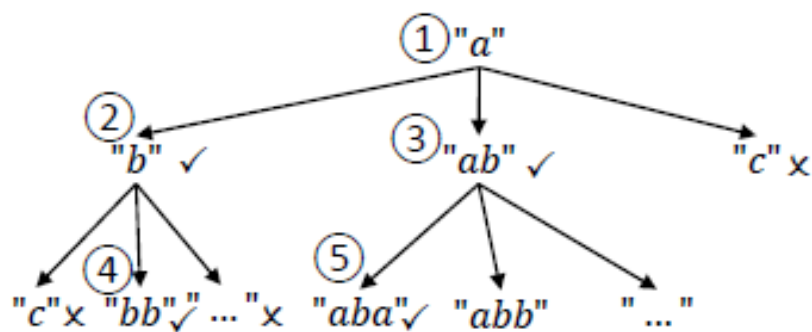
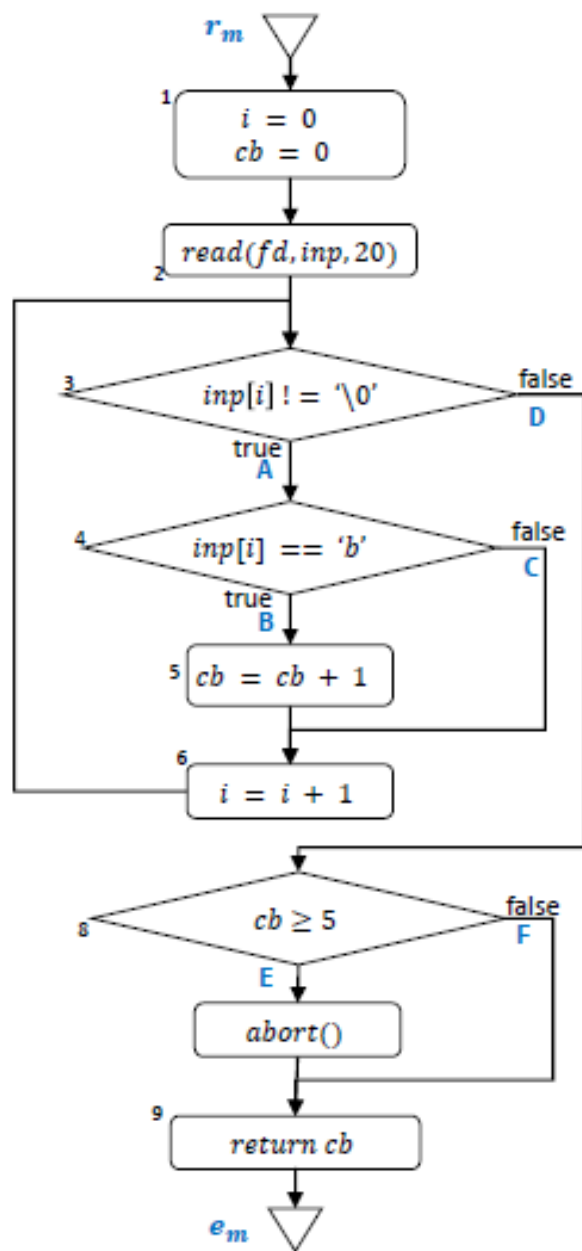


Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
	"c"		1				1		1

来源:

https://z14120902.github.io/teaching/ict_alg/fuzz_mc.pdf

Grey-box fuzzing – Working example



Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
4	"bb"	2		1		1			1
5	"aba"	1	2	1	1		1		1
	"abb"	2	1	1	1	1			1

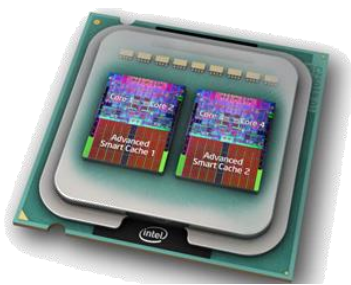
来源:

https://z14120902.github.io/teaching/ict_alg/fuzz_mc.pdf

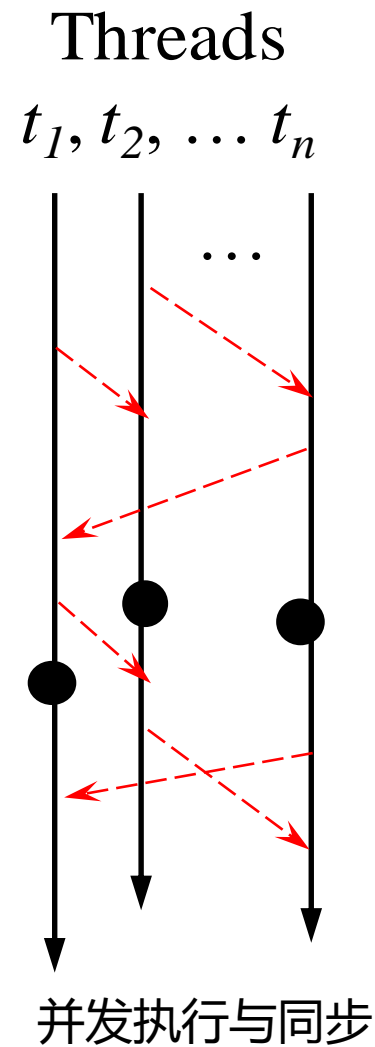
并发程序及其发展现状

➤ 并发程序的基本特点:

- 共享系统资源: 并发程序的各个线程共享系统中的各种资源, 如CPU、内存、I/O设备等。
- 共享数据: 并发程序中存在共享变量, 一个线程对共享变量的修改对于其他线程是立即可见的。
- 任务同步: 通过同步操作协调多个线程, 共同完成工作。
- 并发执行: 在同步操作之外, 各个线程可以并发执行, 互不干扰。
- 原子性: 并发程序中存在原子操作, 即一个指令序列, 要么全部执行成功, 要么全部不执行, 不会出现部分执行的状态。



•Thread 1
•Thread 2
•Thread 3
•Thread 4



基本概念

- 数据竞争：多个线程同时访问共享数据，且其中至少一个操作为写操作。
- 左图中，line 2和4构成对指针p的数据竞争，在右图的线程交错中，会导致UAF
- 引起的危害：
 - 破坏数据一致性
 - 进一步发展为UAF、DF等漏洞

	t ₁	t ₂
1	p = malloc()	p[0] = 0
2		
3	free(p)	
4	p = null	



	t ₁	t ₂
1	p = malloc()	
2	free(p)	
3	p = null	
4		p[0] = 0

if(p!=NULL) p[0]=0;

动态方法-LockSet

➤ 朴素的LockSet算法:

- 对每个线程 t , 维护一个 $\text{locks_held}(t)$, 记录 t 当前持有的锁的集合（持有是指加锁之后还未解锁）
- 对每一个共享变量 v , 维护一个锁集 $C(v)$, 记录着所有对 v 进行访问的操作共同持有的锁的集合
- 初始状态下, $C(v)$ 包含所有的锁, $\text{locks_held}(t)$ 为空
- 当线程 t 对锁 l_1 加锁时, $\text{locks_held}(t) = \text{locks_held}(t) \cup \{l_1\}$; 对锁 l_1 解锁时, 将 l_1 从 $\text{locks_held}(t)$ 中移除
- 当线程 t 对共享变量进行访问时（读或者写）, 更新 $C(v) = C(v) \cap \text{locks_held}(t)$
- 如果执行过程中, $C(v)$ 变成空集, 就认为此处存在并发缺陷（数据竞争）

动态方法-LockSet

共享变量a, 锁 l_1 和 l_2

初始: $C(a) = \{l_1, l_2\}$, $locks_held(t_1) = locks_held(t_2) = \{\}$

	t_1	t_2	
$locks_held(t_1) = \{l_1\}$	1 $lock(l_1)$		
$C(a) = C(a) \cap locks_held(t_1) = \{l_1\}$	2 $a = 0$		
$locks_held(t_1) = \{\}$	3 $unlock(l_1)$		
	4	4 $lock(l_2)$	$locks_held(t_2) = \{l_2\}$
	5	5 $a = 1$	$C(a) = C(a) \cap locks_held(t_2) = \{\}$
	6	6 $unlock(l_2)$	$locks_held(t_2) = \{\}$

RACE!



数据竞争: line 2, line 5

动态方法-LockSet

共享变量a, b, 锁 l_1 和 l_2

初始: $C(a) = C(b) = \{l_1, l_2\}$, $locks_held(t_1) = locks_held(t_2) = \{\}$

	t_1	t_2	
$locks_held(t_1) = \{l_1\}$	1 $lock(l_1)$		
$C(a) = C(a) \cap locks_held(t_1) = \{l_1\}$	2 $a = 0$		
$locks_held(t_1) = \{\}$	3 $unlock(l_1)$		
	4	$lock(l_2)$	$locks_held(t_2) = \{l_2\}$
	5	$a = 1$	$C(a) = C(a) \cap locks_held(t_2) = \{\}$
	6	$unlock(l_2)$	$locks_held(t_2) = \{\}$
$C(b) = C(b) \cap locks_held(t_1) = \{\}$	7 $b = 0$		

RACE!



误报: 只有一个线程进行过访问!

动态方法-LockSet

共享变量a, 锁 l_1 和 l_2

初始: $C(a) = \{l_1, l_2\}$, $locks_held(t_1) = locks_held(t_2) = \{\}$

	t_1	t_2
$locks_held(t_1) = \{l_1\}$	1 $lock(l_1)$	
$C(a) = C(a) \cap locks_held(t_1) = \{l_1\}$	2 $x = a + 1$	
$locks_held(t_1) = \{\}$	3 $unlock(l_1)$	
	4	$lock(l_2)$ $locks_held(t_2) = \{l_2\}$
	5	$y = a + 1$ $C(a) = C(a) \cap locks_held(t_2) = \{\}$
	6	$unlock(l_2)$ $locks_held(t_2) = \{\}$

误报: 只有读操作!



动态方法-LockSet

共享变量a, 读写锁 l_1

初始: $C(a) = \{l_1\}$, $locks_held(t_1) = locks_held(t_2) = \{\}$

	t_1	t_2
$locks_held(t_1) = \{l_1\}$	1 $rdlock(l_1)$	
$C(a) = C(a) \cap locks_held(t_1) = \{l_1\}$	2 $x = a + 1$	
$locks_held(t_1) = \{\}$	3 $unlock(l_1)$	
	4	$rdlock(l_1)$ $locks_held(t_2) = \{l_1\}$
	5	$a = 1$ $C(a) = C(a) \cap locks_held(t_2) = \{l_1\}$
	6	$unlock(l_1)$ $locks_held(t_2) = \{\}$

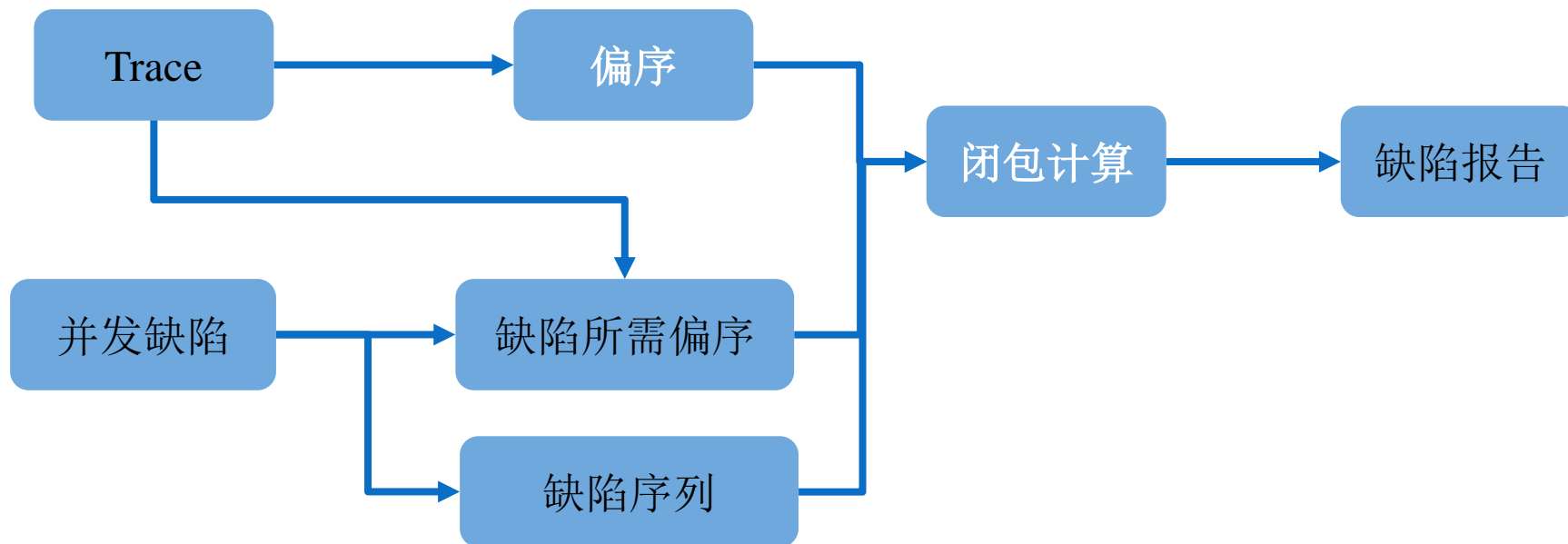
漏报: 写操作只持有读锁!

动态方法-LockSet

- 朴素的LockSet约束无法处理的情况：
 - **Initialization:** 共享变量初始化之后，其他线程未访问共享变量，此时即使没有锁保护，也不会产生数据竞争。
 - **Read-shared data:** 共享变量只有在初始化的时候被写访问过，其他后续线程对这个共享变量只有读操作，那么也不会产生数据竞争（数据竞争至少要有有一个写操作）。
 - **Read-Write locks:** 读写锁（允许多个线程获取读锁，只允许一个线程获得写锁；获得了读锁之后，写锁将会被阻塞，直到所有获得读锁线程释放；而获得写锁之后，所有读锁将会被阻塞，直到获得写锁的线程释放），这时候允许多个线程获得读锁并对共享变量进行访问。

动态方法-图算法

- 利用图算法取代向量时钟算法，可以探索更大的线程交错空间
- 算法流程：给定执行序列
 - 首先提取偏序（偏序种类增多，精细度提高）
 - 然后找出潜在的并发缺陷（例如可能构成数据竞争的事件对），构建触发数据竞争的序列片段以及所需的相应偏序
 - 以事件为结点，以偏序关系为有向边，构建有向图
 - 结合执行序列偏序、数据竞争序列片段、数据竞争所需偏序，在图上执行闭包计算，检查是否有环路



自动程序修复

自动程序修复 APR (Automated/Automatic Program Repair/Fixing)

- 自动化的软件维护方法，利用程序分析技术和人工智能来发现、定位和修复程序中的错误
- 可以帮助开发人员快速修复大量缺陷报告，减少程序调试时间，提高软件的可信赖性和安全性

程序错误 (program error)

- 程序的预期行为和实际执行时所发生情况之间存在的一种偏差
- 程序的预期行为：又称为**程序规约**(program specification)，可以是自然语言书写的文本、形式化的逻辑公式，或者测试集等

自动程序修复就可以解释为将不符合程序规约的错误语义自动转换为符合程序规约的一种技术

自动程序修复

自动程序修复的过程一般分为三个阶段。

- **缺陷定位：**通过静态分析或动态测试方法确定程序可能的缺陷位置
- **补丁生成：**选择一个缺陷位置，使用特定的补丁生成技术尝试生成修复补丁
- **补丁评估：**利用测试集等验证生成补丁的正确性



APR的分类

按照自动化程度

- 全自动修复
- 半自动修复（为开发人员提供候选补丁）

按照修复策略

- 基于状态的修复（修改程序运行时的寄存器等执行状态）
- 基于行为的修复（直接改变程序代码）

按照程序规约

- 不完全规约的修复（基于测试集）、
- 完全规约的修复（对特定错误的完全修复）、
- 半完全规约的修复（手工编写规约等）

经典的APR方法是基于动态测试的程序自动修复

Q & A

问答

A black and white illustration of a hand-drawn pen writing the words "Thank you" in a cursive script. The pen is positioned at the end of the word "you", and the entire phrase is enclosed within a thin rectangular border.