博士论文答辩报告

融合程序分析与测试的内存安全漏洞检测技术研究

报 告 人：文　成

指导老师：秦胜潮、许智武

目　录

CONTENTS

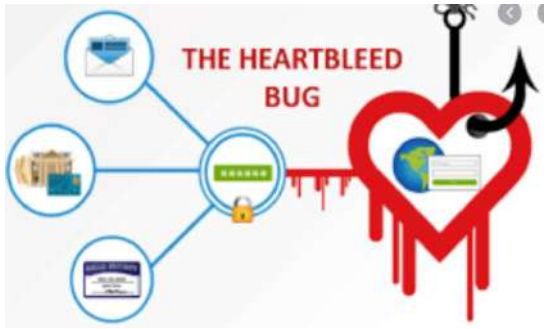
深圳大学 SHENZHEN UNIVERSITY

软件错误/漏洞时有发生

Intel Pentium漏洞导致声誉和巨额经济损失

Ariane5火箭升空数秒后爆炸（损失：85亿美元）

软件漏洞导致Toyota回收120万辆Prius



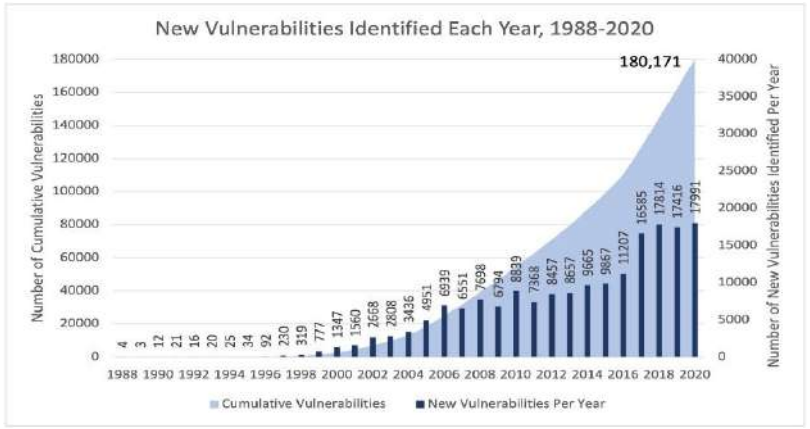
Openssl的“心脏滴血”漏洞致使2亿网民面临信息泄漏的风险

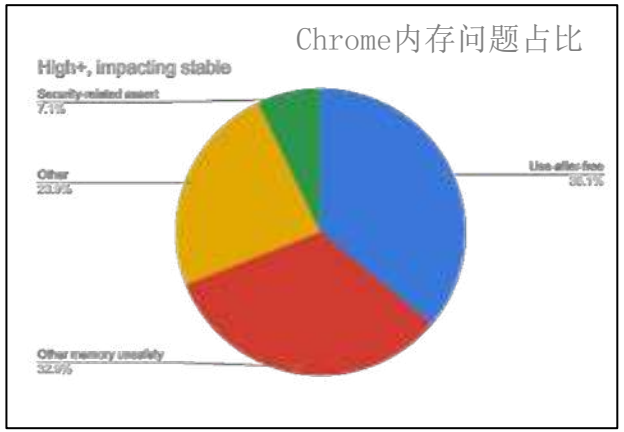
软件数据竞争问题导致美国东北部大面积停电（2003）

证券交易所软件失误：骑士资本一夜之间蒸发4.4亿美元（2012）

软件并发漏洞导致Therac25放疗仪使用超过量的放射物，至少6名患者死亡，多人受伤

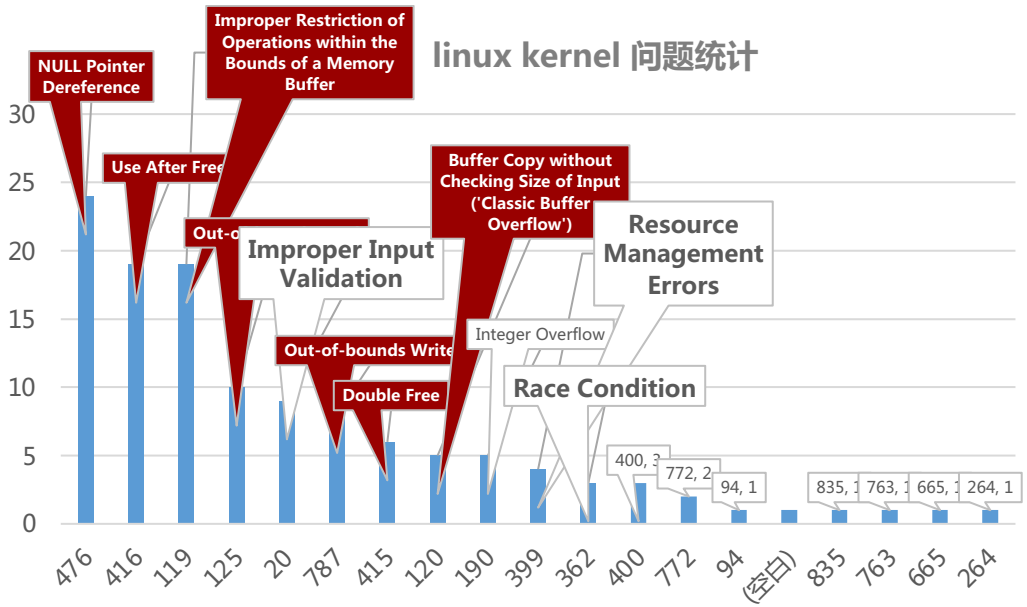
内存安全问题当前依然是软件漏洞的最主要来源





Total CVE Count	Memory Unsafety Bugs	Percentage	Release
44	36	81.8%	10.14.6
45	40	88.9%	10.14.5
38	20	52.6%	10.14.4
23	22	95.7%	10.14.3
13	11	84.6%	10.14.2
71	40	56.3%	10.14.1
64	4		

Mac OS 14各版本中内存问题占比



内存安全问题在操作系统，浏览器，应用软件等领域均广泛存在，并持续造成大量功能安全问题和信息安全问题；

- 微软系列产品中**超过70%漏洞**均由内存安全问题导致
- Chrome浏览器项目中大约**70%漏洞**是内存安全问题
- MacOS/iOS中**60-80%漏洞**由内存安全问题导致

内存安全问题一直是软件漏洞的主因，至今仍未被解决，且问题数量不收敛。
内存安全漏洞的严重程度很高，一般至少是系统Crash

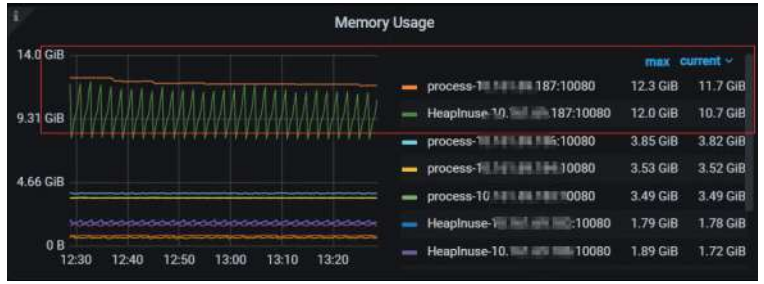
内存安全错误/漏洞

内存安全漏洞：

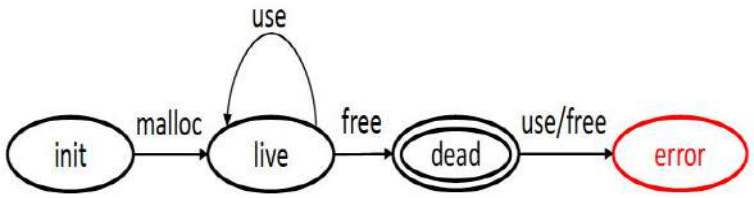
- 在软件中对内存进行不恰当的分配、释放、读取和写入等操作导致程序错误

内存安全漏洞特点：

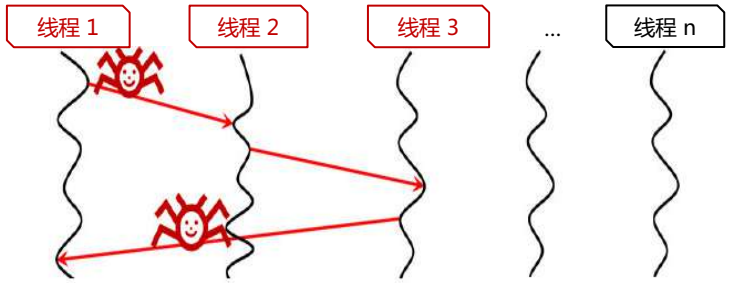
- 严重程度高、危害大
- 漏洞类型多种多样
- 部分类型的漏洞隐藏深、难发现



内存消耗漏洞：软件没有合理地控制有限内存资源的分配和维护，从而使参与者可以影响消耗的内存量，最终导致可用内存的耗尽。



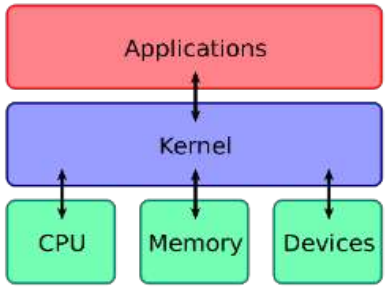
内存时序漏洞：软件在进行内存操作时违反了内存使用的安全时序规则，在释放内存后继续使用内存或再次释放内存，可能导致未定义行为或程序崩溃。



内存并发漏洞：多个线程因线程交错，交互地作用于一个或多个共享内存，引起程序崩溃或挂起，或产生与串行执行不同的结果。

现有的技术手段难以彻底规避内存安全错误/漏洞

现实应用软件场景特点：**软件规模大，对安全性和可靠性要求高**



操作系统内核

浏览器

音视频处理软件

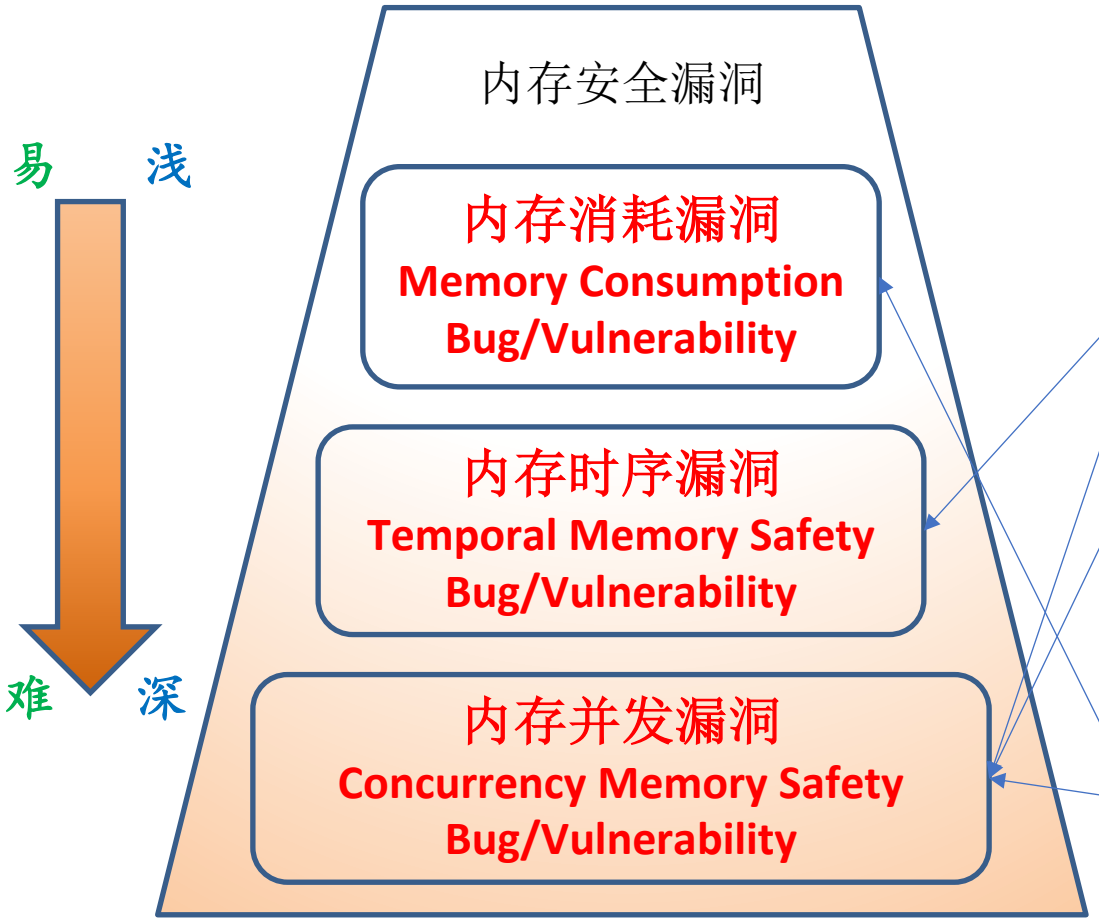
数据库软件

解压缩软件

目前主流技术**难以满足**这种复杂场景下的内存安全性需求：

技术路线	代表工具	准确率	覆盖率	可扩展性	评价
静态分析	Cppcheck/Infer	低	高	较好	分析成本低但误报率较高
动态测试	AFL/Asan	高	低	好	分析结果准确但漏报率较高
形式化证明	RefinedC/Frama-C	高	高	差	仅适用于不计成本的高安场景
	本文提出的方法	高	较高	较好	现实应用软件场景下成本效益的平衡

三类隐藏深、难发现、危害大的内存安全漏洞



Rank	ID	Name	Score
1	CWE-787	Out-of-bounds Write	64.20
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11
4	CWE-20	Improper Input Validation	20.63
5	CWE-125	Out-of-bounds Read	17.67
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53
7	CWE-416	Use After Free	15.50
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56
11	CWE-476	NULL Pointer Dereference	7.15
12	CWE-502	Deserialization of Untrusted Data	6.68
13	CWE-190	Integer Overflow or Wraparound	6.53
14	CWE-287	Improper Authentication	6.35
15	CWE-798	Use of Hard-coded Credentials	5.66
16	CWE-862	Missing Authorization	5.53
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42
18	CWE-306	Missing Authentication for Critical Function	5.15
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85
20	CWE-276	Incorrect Default Permissions	4.84
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57
23	CWE-400	Uncontrolled Resource Consumption	3.56
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32

2022 CWE Top 25 Most Dangerous Software Weaknesses

本文的主要研究问题

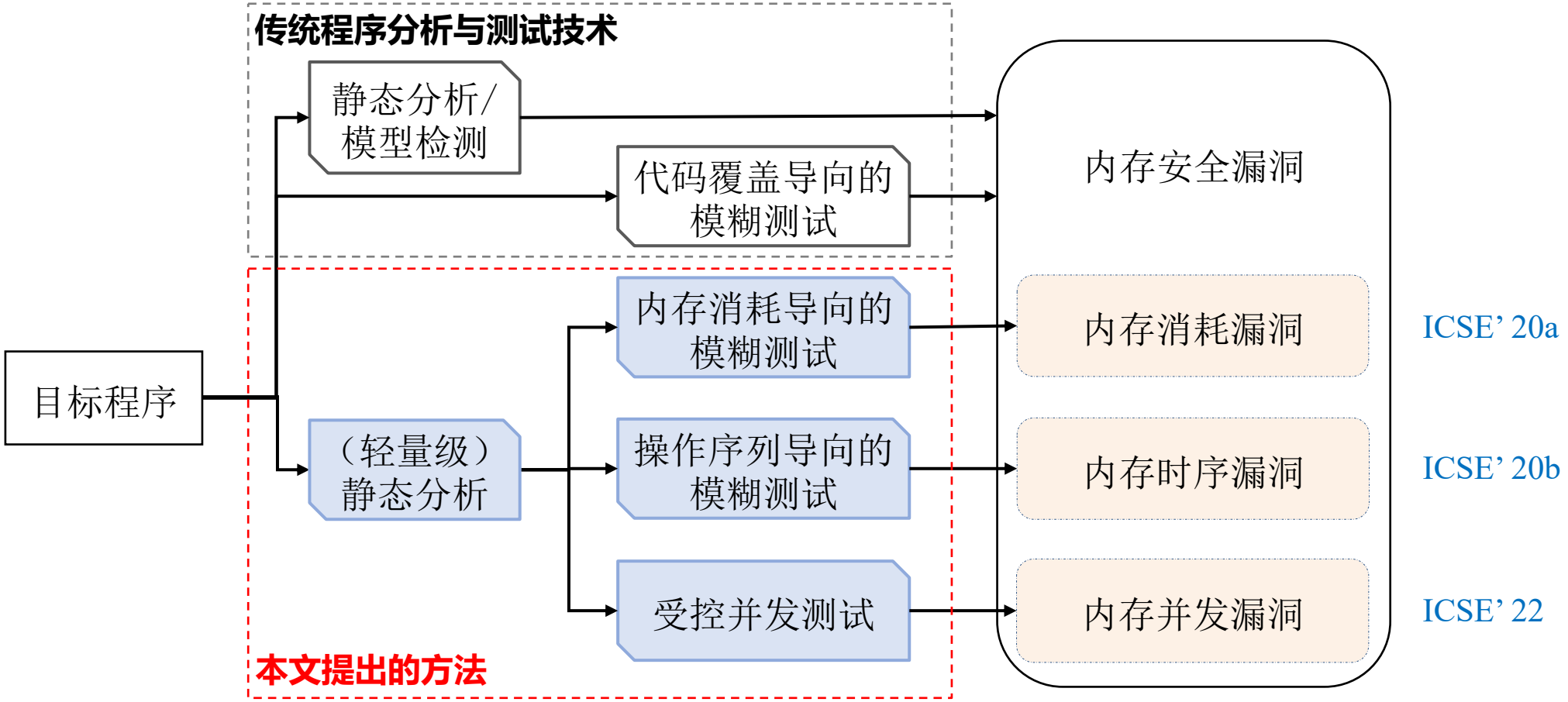
- 针对内存消耗漏洞、内存时序漏洞、内存并发漏洞提出有效实用的分析方法和检测技术。
- 发现业界实际应用软件中的内存安全问题，提升软件质量。

本文中的分类	漏洞名称	英文名称	相关 CWE 编号
内存消耗漏洞	不受控的递归调用	Uncontrolled Recursion	CWE-400、CWE-674
	不受控的内存分配	Uncontrolled Memory Allocation	CWE-400、CWE-789
	内存泄露	Memory Leak	CWE-400、CVE-401
内存时序漏洞	释放后使用	Use-after-free（UaF）	CWE-416
	双重释放	Double-free（DF）	CWE-415
内存并发漏洞	竞态条件	Race Condition	CWE-362
	并发释放后使用	Concurrency Use-after-free	CWE-362、CWE-416
	并发双重释放	Concurrency Double-free	CWE-362、CWE-415
	并发空指针解引用	Concurrency Null-pointer dereference	CWE-362、CWE-476
	死锁	Deadlock	CWE-833

本文主要研究的内存安全漏洞类型

主要研究方法与手段

- 一套通用可扩展的内存安全漏洞检测框架，融合了轻量级程序分析技术与自动化测试技术。
- 对内存消耗漏洞、内存时序漏洞、内存并发漏洞的检测采取分而治之的策略。



本文所提方法的特点

- **错误发现**：侧重更有效地发现错误，防止误报，降低漏报
- **自动化**：侧重于自动化的分析与测试，需要人工干预的任务较少
- **可扩展**：能适用于大规模场景（十万行-百万行代码）
- **针对特定性质**：针对特别需要保障的某个内存安全性质开展分析与测试
- **有效**：相比业界先进的方法发现的漏洞数量要多于20%-30%，且效率提升到2倍以上
- **实用**：新发现并报告了大量未知的、安全攸关的软件漏洞（36 CVEs）

高成本原子弹（形式化证明）

地毯式轰炸（传统程序分析或测试）

精准定位打击（本文提出的方法）

本文的主要创新点

- 本文提出了一套通用可扩展的内存安全漏洞检测框架，即先采用轻量级程序分析技术定位潜在的漏洞，后指导自动化测试技术限定检测的范围，从而有方向、有目标地发现各类内存安全漏洞。
  - 首次提出增加内存消耗这个新维度来指导模糊测试，使得测试过程中能自动化地逐步生成能造成过量内存消耗的测试用例，弥补了当前模糊测试技术在检测内存消耗漏洞方面的盲区。
  - 首次提出将违反内存使用的安全时序规则的操作序列用于引导模糊测试，提升了模糊测试在发现内存时序漏洞方面的有效性和效率。
  - 创新性地提出了一种系统性的并发程序测试方法，使用一种基于周期性执行的调度方案来主动控制线程调度，能根据历史执行信息循序渐进地高效探索调度空间，极大程度地提升了发现并发漏洞的效率。

目　录

CONTENTS

深圳大学 SHENZHEN UNIVERSITY

内存消耗漏洞

内存消耗漏洞：软件没有合理地控制有限内存资源的分配和维护，从而使参与者可以影响消耗的内存量，最终导致可用内存的耗尽。

典型的内存消耗漏洞：

```
// CWE-674: 不受控的递归调用
void recur (int depth) {
        int a[10];
        return recur(depth-1);
}

int main (void) {
        int depth = get_user_input();
        recur(depth);
        return 0;
}
```

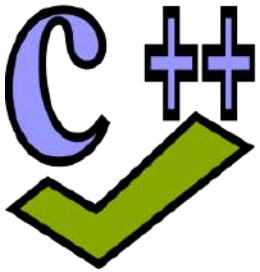
```
// CWE-789: 不受控的内存分配
void main (void) {
        int size = get_user_input();
        char *p = malloc(size);
        use(p);
        free(p);
}
```

```
// CWE-401: （不受控的）内存泄漏
void func (int size) {
    char *p = malloc(size * sizeof(char))
    use(p);
}
```

内存消耗漏洞检测的难点以及现有方法的局限性

内存消耗漏洞：软件没有合理地控制有限内存资源的分配和维护，从而使参与者可以影响消耗的内存量，最终导致可用内存的耗尽。

- 难点：这类漏洞的触发不仅依赖于程序的执行路径，还依赖于执行路径上的内存消耗，这要求对程序的内存消耗进行建模和分析
- 现有方法局限性：
  - 内存模型缺乏对内存消耗的建模，大部分静态分析工具不检查这种错误；分析循环/递归结构的精度低；误报和漏报较高（静态）；
  - 以代码覆盖率作为覆盖评估的标准，难以生成能触发过量内存消耗的输入（动态）

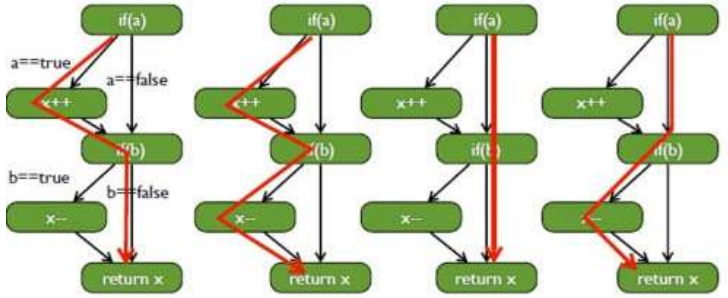


CBMC: Bounded Model Checking for ANSI-C

CBMC

Version 1.0, 2010

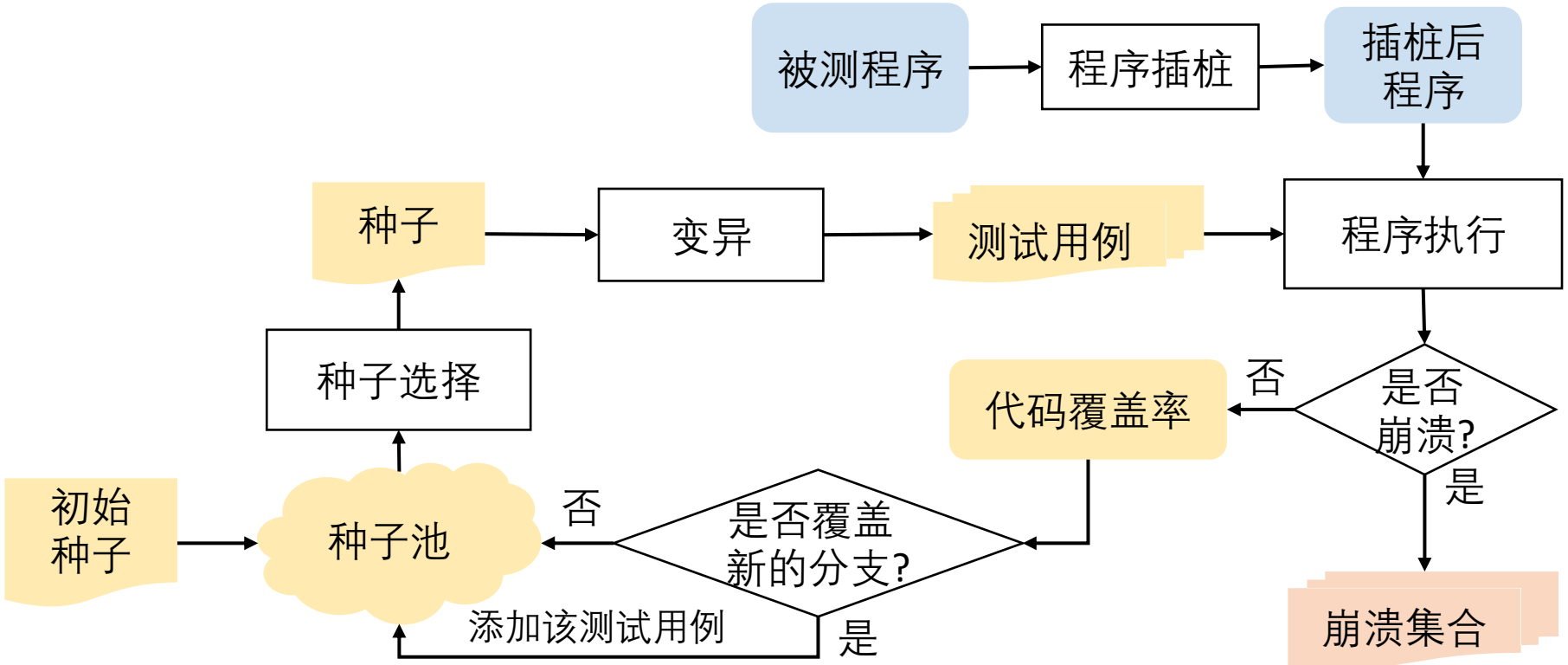
静态分析工具（例如Infer，Cppcheck，CBMC）不检查这类漏洞



动态测试工具（例如AFL）关注代码覆盖率而不关注已覆盖路径上的内存消耗

代码覆盖导向的模糊测试

- 一种有效的自动化软件测试技术，能自动生成海量的输入来执行程序，通过监视程序运行过程中的异常来检测软件缺陷/漏洞。
- 在工业界被广泛使用（例如谷歌公司的AFL工具已经发现了上千个软件漏洞）



该技术能有效发现大部分浅层的内存安全漏洞，但对于内存消耗漏洞的检测仍存在盲区

启发性示例：不受控的递归调用漏洞

CVE-2018-17985漏洞（本文新发现的漏洞）

```
1  struct demangle_component *
2  cplus_demangle_type (struct d_info *di) {
3
4    // "peek" is a single character extracted from the input directly
5    char peek = d_peek_char (di);
6
7    switch (peek){
8      ...
9      case 'P':
10       ret = d_make_comp (di,
11         DEMANGLE_COMPONENT_POINTER,
12         cplus_demangle_type (di), NULL);
13       break;
14     case 'C':
15     ...
16   }
17   ...
18 }
```

递归函数

该代码片段来自GNU开源工具集 *Binutils v2.31* 中的 *c++filt* 程序

输入	递归调用深度	种子保留状态
p	1	Interesting
pp	2	Interesting
pppp	4	Interesting
pppp...ppp	128	Interesting
pppp...ppppp	260	**Discard**
pppp...pppppp	…	**Discard**
pppp…ppppppp	**35000+**	**Trigger the bug**

启发性示例：不受控的内存分配/内存泄漏漏洞

CVE-2018-4866漏洞

```
1  class EXIV2API DataBuf {
2  public:
3    // Constructor with an initial buffer size
4    explicit DataBuf(long size): pData(new byte[size]), size(size) {}
5    ...
6    byte* pData; // Pointer to the buffer
7    size_t size; // The current size of the buffer
8  };
9
10 void Jp2Image::readMetadata() {
11   while (io_->read((byte*)&subBox, sizeof(subBox)) ==
     ↪  sizeof(subBox) && subBox.length ) {
12     subBox.length = getLong((byte*)&subBox.length, bigEndian);
13     DataBuf data(subBox.length); // Allocation without checking
14     ...
15     io_->seek(position - sizeof(box) + box.length, BasicIo::beg);
16   }
17 }
```

内存分配操作

subBox来自用户的输入

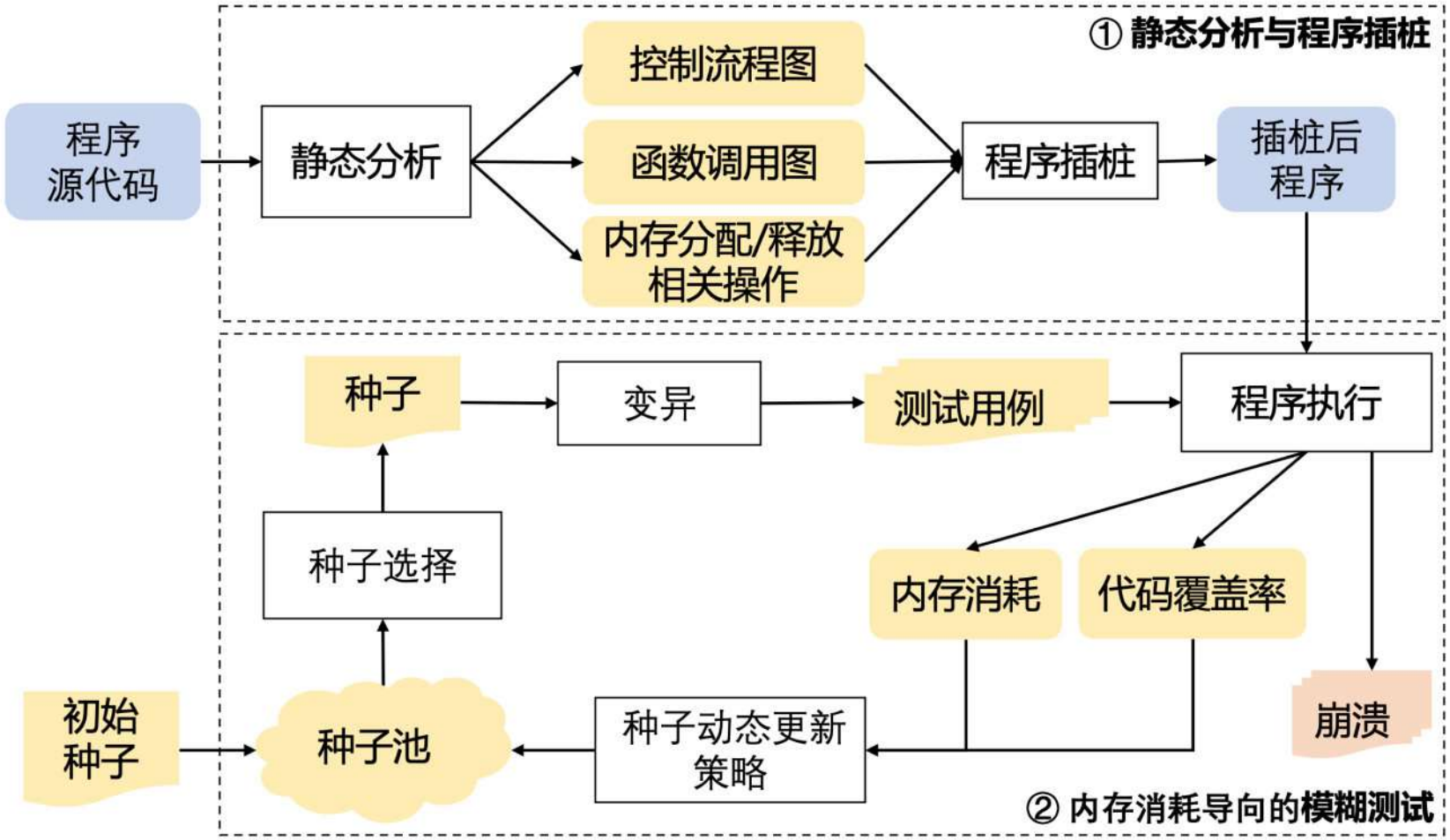
在未进行安全检查的情况下直接分配内存空间

该代码片段来自开源工具 *Exiv2 v0.26* 的 *jp2image.cpp* 文件

输入（size）	种子保留状态
2048	Interesting
8192	**Discard**
61440	**Discard**
2097152	**Discard**
53687092	**Discard**
…	**Discard**
4294967296+	**Trigger the bug**

本文提出的内存消耗漏洞检测方法：MemLock

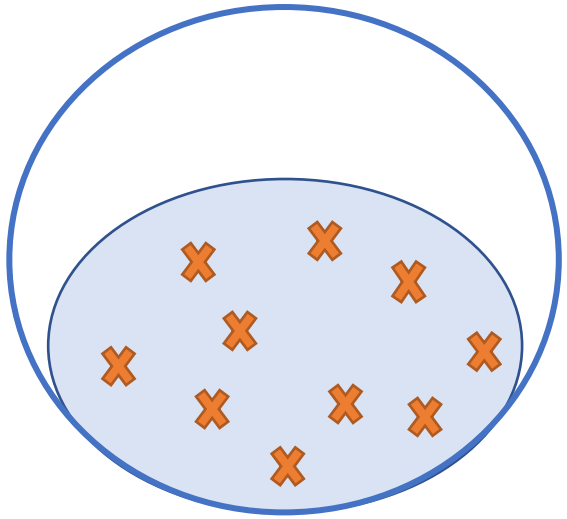
- 采用轻量级的程序静态分析技术识别能影响内存消耗的内存操作语句
- 使用内存消耗导向的模糊测试技术自动化地生成能造成过量内存消耗的输入



本文提出的内存消耗漏洞检测方法：MemLock

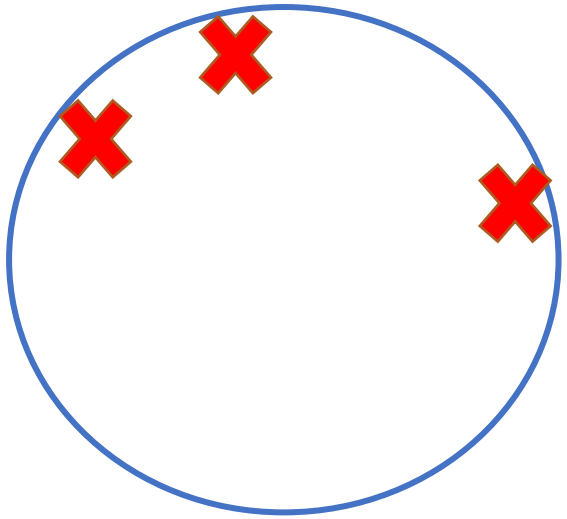
- 采用轻量级的程序静态分析技术识别能影响内存消耗的内存操作语句
- 使用内存消耗导向的模糊测试技术自动化地生成能造成过量内存消耗的输入

与代码覆盖导向的模糊测试技术区别：



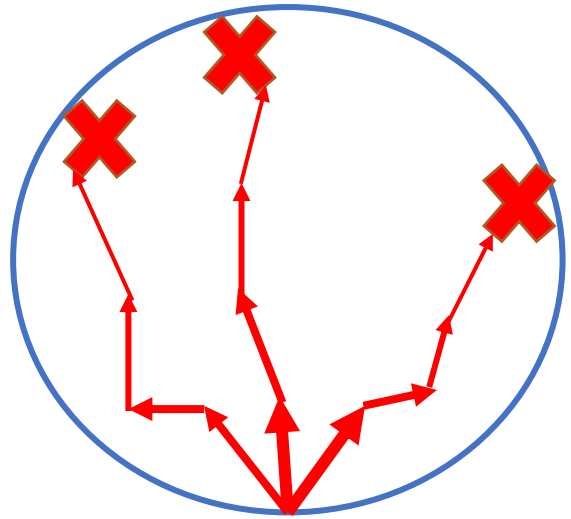
局部海域，遍地撒网

代码覆盖导向的模糊测试



精准定位，指明方向

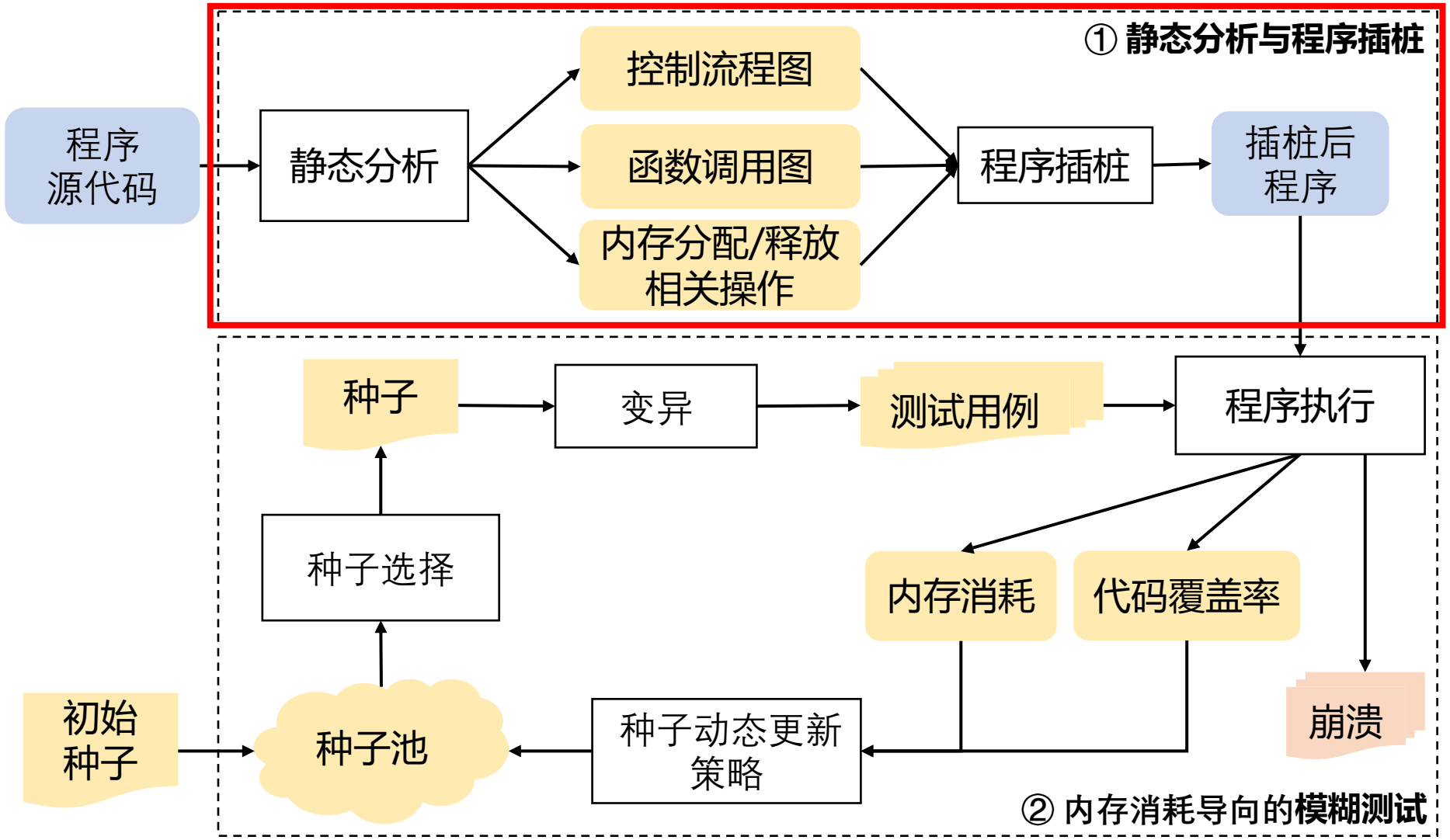
轻量级静态分析



自动导航，足下践行

内存消耗导向的模糊测试

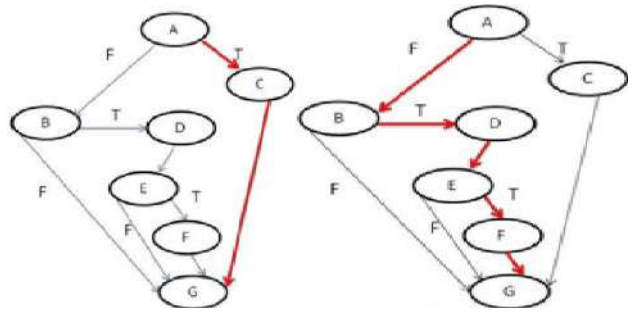
MemLock的整体流程



静态分析与程序插桩

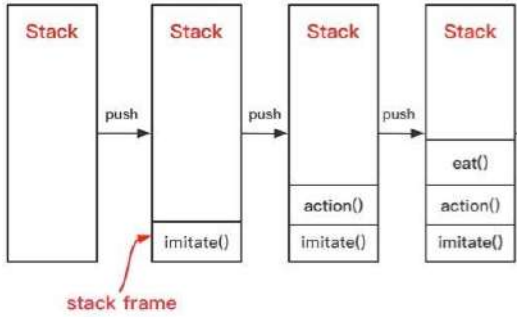
分析三类静态信息，通过程序插桩在运行时收集动态信息

- 控制流程图
  - 对控制流程图中的每条边插桩，收集分支覆盖率信息



执行路径覆盖的控制流边

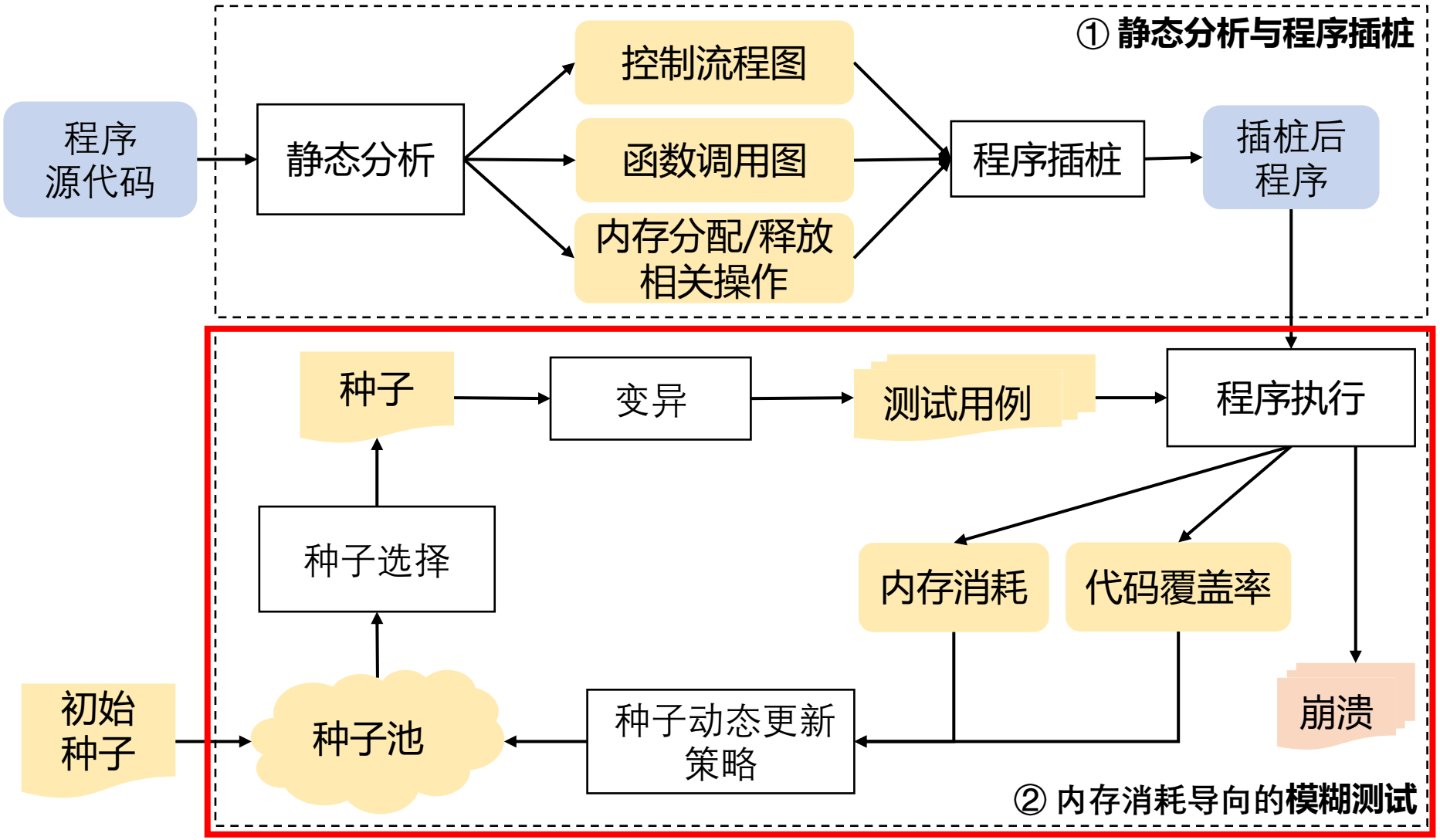
- 函数调用图
  - 函数入口插桩，函数调用意味着压栈，栈内存消耗增加
  - 函数出口插桩，函数返回意味栈空间被回收，栈内存消耗减少



栈帧示意图

- 内存分配/释放操作
  - 在每个内存分配操作前进行插桩，计算已分配的堆内存空间的大小
  - 在每个内存释放操作前进行插桩，在释放内存后更新堆内存空间的消耗量

MemLock的整体流程

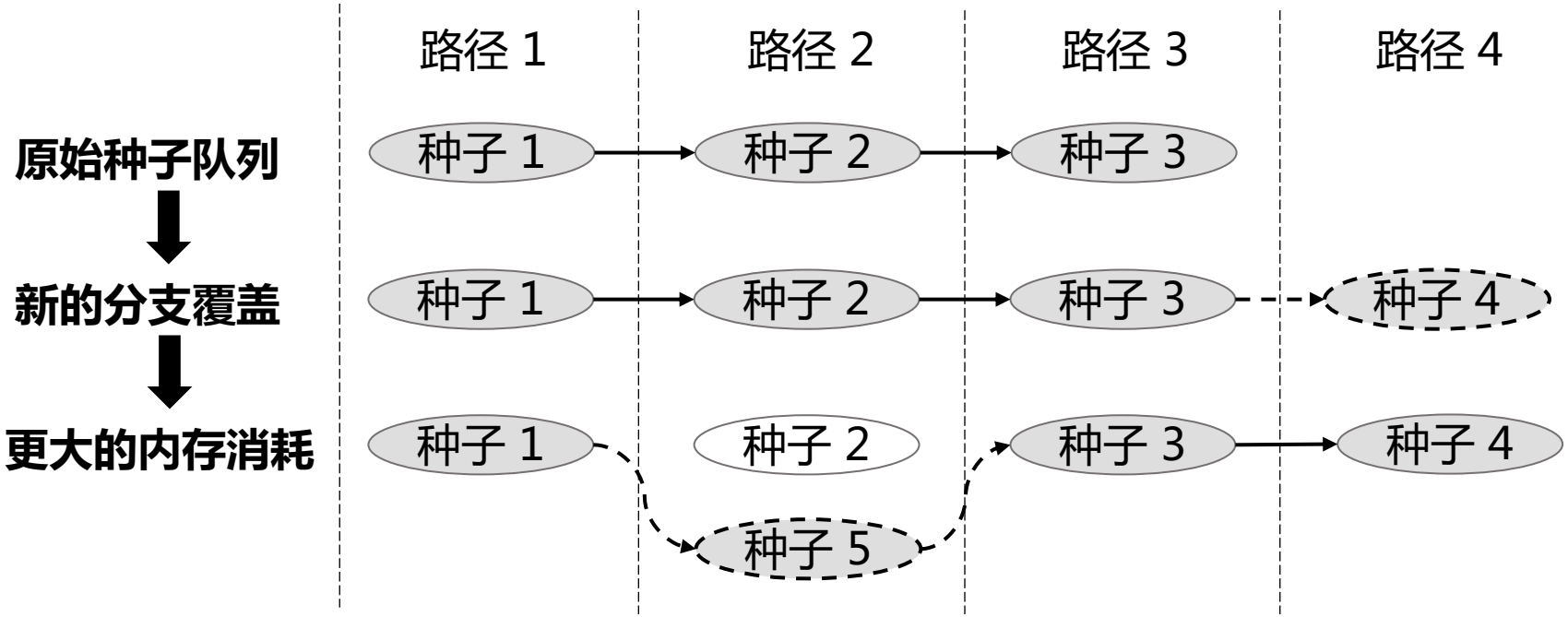


内存消耗引导的模糊测试

MemLock同时使用两个维度的信息来指导模糊测试

- 代码覆盖率信息：高效探索不同的程序路径
- 内存消耗信息：为每条程序路径触发更大的内存消耗

种子动态更新策略：有效利用两个维度的信息保留种子



运行示例： MemLock对CVE-2018-17985漏洞的检测过程

该程序中有一个不受控的递归调用漏洞

```
1  struct demangle_component *
2  cplus_demangle_type (struct d_info *di) {
3
4    // "peek" is a single character extracted from the input directly
5    char peek = d_peek_char (di);
6
7    switch (peek){
8      ...
9      case 'P':
10       ret = d_make_comp (di,
11         DEMANGLE_COMPONENT_POINTER,
12         cplus_demangle_type (di), NULL);
13       break;
14     case 'C':
15     ...
16   }
17   ...
18 }
```

递归函数

该代码片段来自GNU开源工具集 *Binutils v2.31* 中的 *c++filt* 程序

Memlock 生成的输入	递归调用深度	种子保留状态
p	1	Interesting
pp	2	Updating
pppp	4	Updating
pppp...ppp	128	Updating
pppp...ppppp	260	Updating
pppp...pppppp	…	Updating
pppp…ppppppp	**35000+**	**Trigger the bug**

运行示例： MemLock对CVE-2018-4866漏洞的检测过程

该程序中有一个不受控的内存分配漏洞

```
1  class EXIV2API DataBuf {
2  public:
3    // Constructor with an initial buffer size
4    explicit DataBuf(long size): pData(new byte[size]), size(size) {}
5    ...
6    byte* pData; // Pointer to the buffer
7    size_t size; // The current size of the buffer
8  };
9
10 void Jp2Image::readMetadata() {
11   while (io_->read((byte*)&subBox, sizeof(subBox)) ==
     ↪  sizeof(subBox) && subBox.length ) {
12     subBox.length = getLong((byte*)&subBox.length, bigEndian);
13     DataBuf data(subBox.length); // Allocation without checking
14     ...
15     io_->seek(position - sizeof(box) + box.length, BasicIo::beg);
16   }
17 }
```

内存分配操作

subBox来自用户的输入

在未进行安全检查的情况下直接分配内存空间

该代码片段来自开源工具 *Exiv2 v0.26* 的 *jp2image.cpp* 文件

MemLock 生成的输入(size)	种子保留状态
2048	Interesting
8192	Updating
61440	Updating
2097152	Updating
53687092	Updating
…	Updating
4294967296+	**Trigger the bug**

实验评估

MemLock的工具原型基于LLVM平台、AFL工具构建

- https://github.com/wcventure/MemLock-Fuzz

实验数据集：

- 14个常用的、功能各异的、不同规模的开源应用程序

对比的技术：

- AFL [AFL 2.52b]
- AFLfast [Böhme2017]
- PerfFuzz [Lemieux2018]
- FairFuzz [Lemieux2017]
- Angora [Chen2018]
- QSYM [Yun2018]

实验设计：

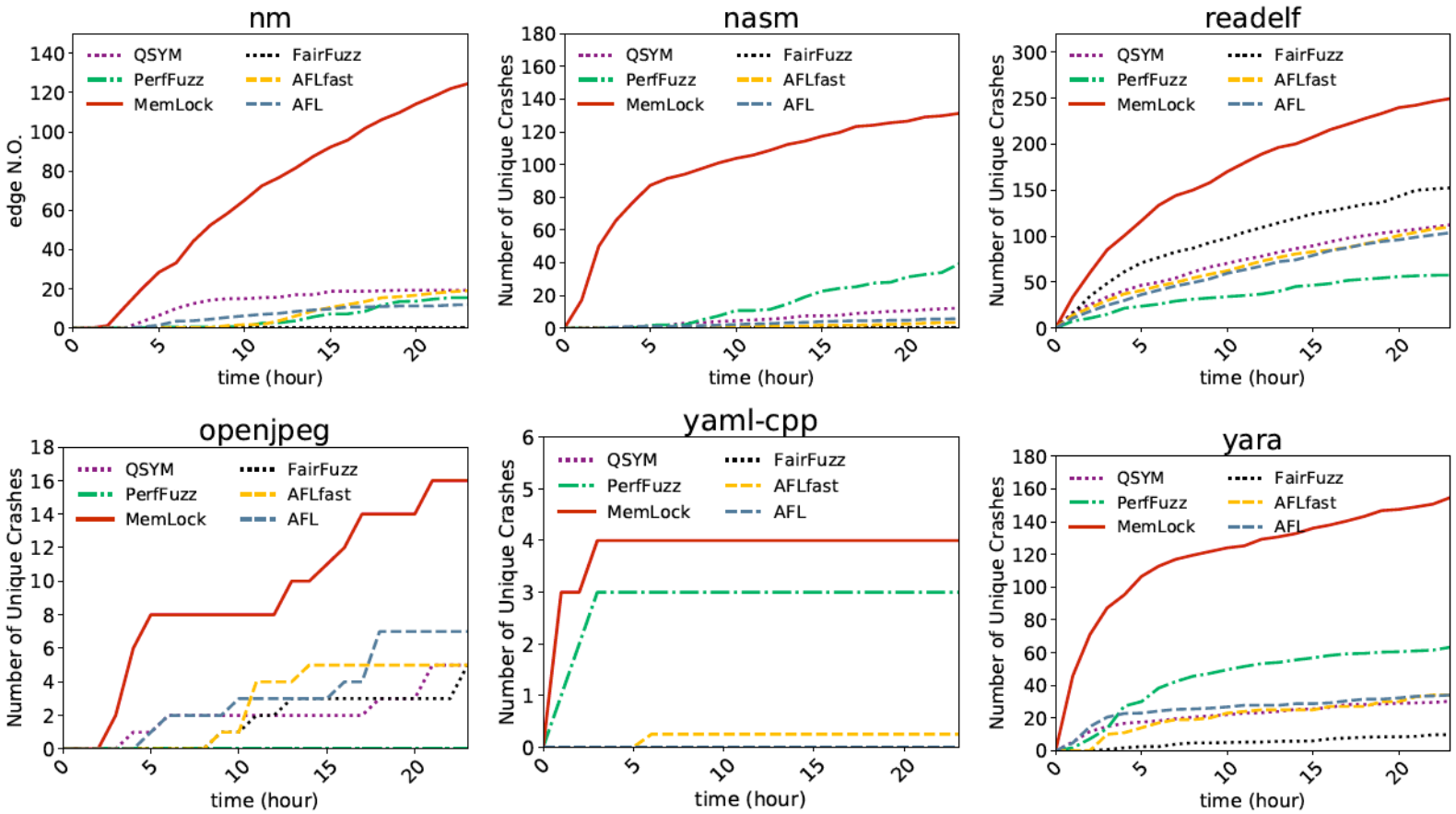
- 实验一：崩溃数量评估
- 实验二：漏洞检测能力评估
- 实验三：内存泄漏大小评估
- 实验四：种子造成的内存消耗评估

实验设计：

- 每个实验对象运行24小时，重复10次

实验一：崩溃数量评估

- MemLock与AFL、AFLfast、PerfFuzz、FairFuzz、Angora和QSYM相比，发现的与内存消耗漏洞相关的崩溃数量分别提高了59.2%、70.5%、76.9%、98.1%、40.5%和66.7%。
- MemLock发现内存消耗漏洞的崩溃的效率相比其它方法也更高



实验二：漏洞检测能力评估（一）

Program	Vulnerability	MemLock	AFL		AFLfast		PerfFuzz		FairFuzz		Angora		QSYM	
		Time(h)	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$
mjs	issue#58	0.5	0.3	0.25	0.4	0.25	0.2	0.13	0.4	0.25	T/O	1.00	0.3	0.22
	issue#106	13.7	T/O	1.00	T/O	1.00	T/O	1.00	T/O	1.00	T/O	1.00	T/O	1.00
cxxfilt	CVE-2018-9138	0.3	7.2	1.00	10.1	1.00	0.5	0.81	T/O	1.00	T/O	1.00	3.3	1.00
	CVE-2018-9996	T/O	16.5	0.00	T/O	0.50	T/O	0.50	T/O	0.50	T/O	0.50	T/O	0.50
	CVE-2018-17985	0.2	1.1	1.00	4.5	1.00	0.2	0.63	1.9	1.00	T/O	1.00	1.4	1.00
	CVE-2018-18484	0.2	1	1.00	4.5	1.00	0.2	0.63	8	1.00	T/O	1.00	1.4	1.00
	CVE-2018-18700	0.2	1.2	1.00	4.5	1.00	0.3	0.75	12.6	0.88	T/O	1.00	1.4	1.00
nm	CVE-2018-12641	2.6	19.1	1.00	12.6	1.00	12.2	0.88	T/O	1.00	T/O	1.00	12.8	0.88
	CVE-2018-17985	10.4	18.2	0.81	11.9	0.56	T/O	1.00	T/O	1.00	T/O	1.00	13.3	0.63
	CVE-2018-18484	9.9	16.4	0.84	17.1	0.84	T/O	1.00	T/O	1.00	T/O	1.00	14	0.75
	CVE-2018-18700	9.6	14.9	0.63	17.8	0.88	T/O	1.00	T/O	1.00	T/O	1.00	T/O	1.00
	CVE-2018-18701	13.9	T/O	1.00	T/O	1.00	T/O	1.00	T/O	1.00	T/O	1.00	T/O	1.00
	CVE-2019-9070	18.4	15.6	0.56	13.9	0.44	T/O	1.00	T/O	1.00	T/O	1.00	15.8	0.56
	CVE-2019-9071	12.4	T/O	0.88	14	0.69	T/O	0.88	T/O	0.88	T/O	1.00	T/O	0.88
nasm	CVE-2019-6290	0.9	T/O	1.00	19	1.00	9	1.00	T/O	1.00	T/O	1.00	17.6	0.00
	CVE-2019-6291	1.5	9	0.94	14	1.00	8.7	1.00	T/O	1.00	T/O	1.00	7.5	0.92
flex	CVE-2019-6293	5.4	T/O	1.00	T/O	1.00	T/O	1.00	T/O	1.00	T/O	1.00	T/O	1.00
yaml-cpp	CVE-2019-6292	0.4	T/O	1.00	18.4	1.00	0.9	0.81	T/O	1.00	T/O	1.00	T/O	1.00
	CVE-2018-20573	6.1	T/O	0.88	T/O	0.84	12.4	0.84	T/O	0.84	T/O	1.00	T/O	0.84

实验二：漏洞检测能力评估（二）

Program	Vulnerability	MemLock	AFL		AFLfast		PerfFuzz		FairFuzz		Angora		QSYM	
		Time(h)	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$
libsass	CVE-2018-19837	1.6	13.3	0.88	10.5	0.88	1.8	0.63	8.5	0.88	T/O	1.00	5	0.81
	CVE-2018-20821	0.1	5.7	1.00	6.5	1.00	0.1	0.50	9.5	1.00	T/O	1.00	7.4	1.00
	CVE-2018-20822	15.6	14.3	0.50	19.5	0.56	14.6	0.47	11.3	0.56	0.92	0.00	10.5	0.44
yara	CVE-2017-9438	0.2	0.9	1.00	4.3	1.00	0.6	0.91	5.3	1.00	T/O	1.00	0.8	1.00
readelf	CVE-2017-15996	0.2	0.3	0.86	0.2	0.68	0.5	0.92	0.3	0.68	T/O	1.00	0.3	0.96
exiv2	CVE-2018-4868	0.1	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50
bento4	CVE-2018-20186	0.4	0.4	0.50	0.4	0.50	0.4	0.50	0.4	0.50	0.1	0.00	0.4	0.50
	CVE-2019-7698	14.6	T/O	1.00	T/O	1.00	T/O	1.00	T/O	1.00	0.5	0.00	T/O	1.00
libming	CVE-2019-7581	0.6	0.8	0.68	1.4	0.80	2	0.88	0.4	0.36	T/O	1.00	1.6	0.80
	CVE-2019-7582	0.1	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50
	issue#155	1.4	1	0.30	1.3	0.36	1.4	0.40	1.2	0.42	T/O	1.00	1.6	0.64
openjpeg	CVE-2019-6988	7.8	15.1	0.86	11.1	0.84	T/O	1.00	T/O	1.00	T/O	1.00	15.3	0.81
	CVE-2017-12982	4.5	11.4	0.72	10	0.60	T/O	1.00	11.9	0.64	T/O	1.00	10	0.50
jasper	CVE-2016-8886	4.1	17	0.88	22.3	1.00	T/O	1.00	10.3	0.52	T/O	1.00	18.2	0.88
	issue#207	1.7	2.2	0.62	3.6	0.68	T/O	1.00	2.2	0.68	15.9	1.00	4	0.64
Average Time Usage(Improv.)		**5.4**	11.6 (2.15X)		11.6 (2.15X)		11.9 (2.20X)		14.5 (2.69X)		20.3 (3.76X)		11.2 (2.07X)	
Unique Vulnerabilities(Improv.)		**33**	26 (+26.9%)		28 (+17.9%)		20 (+65.0%)		17 (+94.1%)		6 (+450.0%)		25 (+37.0%)	

MemLock发现独特漏洞的数量最多，所用的时间也最短

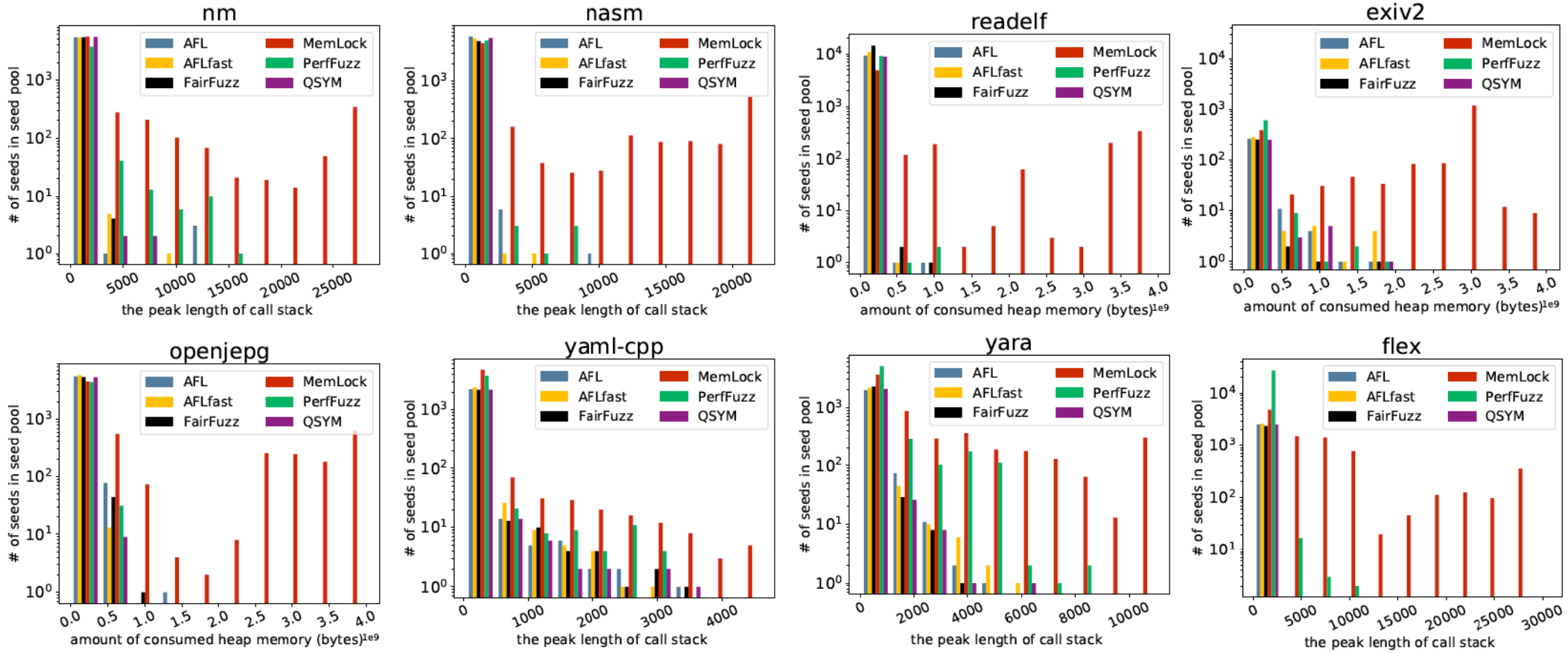
实验三：内存泄漏大小评估

- MemLock生成的种子能够造成更大的内存泄漏
- MemLock在24小时的执行中触发的内存泄漏的大小显著大于其它方法

程序	漏洞类型	工具	内存泄露大小（Bytes）	提升（百分比）	*p*-value	$\hat{A}_{12}$
bento4	CWE-401: 内存泄露	MemLock	**52,709,574**	-	-	-
		AFL	151,862	+34609%	0.0061	1.00
		AFLfast	1,233,255	+4174%	0.0061	1.00
		PerfFuzz	105,984	+49633%	0.0061	1.00
		FairFuzz	1,910,466	+2659%	0.0061	1.00
		Angora	141,512	+37147%	0.0060	1.00
		QSYM	15,784,847	+234%	0.0061	1.00
libming	CWE-401: 内存泄露	MemLock	**176,320,785**	-	-	-
		AFL	4,869,594	+3521%	0.0061	1.00
		AFLfast	2,535,212	+6855%	0.0061	1.00
		PerfFuzz	47,044,964	+257%	0.0061	1.00
		FairFuzz	828,742	+21176%	0.0061	1.00
		Angora	4,698	+3753163%	0.0060	1.00
		QSYM	1,219,093	+14363%	0.0061	1.00
jsaper	CWE-401: 内存泄露	MemLock	**2,372,844,732**	-	-	-
		AFL	56,018,839	+4136%	0.0061	1.00
		AFLfast	48,403,244	+4802%	0.0061	1.00
		PerfFuzz	6,229,898	+37988%	0.0061	1.00
		FairFuzz	56,788,235	+4096%	0.0061	1.00
		Angora	191,907,941	+1136%	0.0105	0.98
		QSYM	38,244,568	+6104%	0.0061	1.00

实验四：种子造成的内存消耗评估

- MemLock生成的种子能够造成更大的内存消耗
- 验证了内存消耗导向的模糊测试方法的确有助于生成能造成过量内存消耗的输入



阶段性研究成果（一）

- 撰写的论文《MemLock: Memory Usage Guided Fuzzing》已在软件工程领域的顶级国际会议ICSE '20上发表。
- MemLock的工具原型和相关实验数据集已通过ICSE’ 20的“Artifacts evaluated available and reusable”的认证。
- 在被广泛使用的真实应用程序中新发现了28个安全攸关的内存消耗漏洞（28 CVEs），在撰写本文时，其中25个漏洞已经被修复。

表 3-5 MemLock 新发现的内存消耗漏洞（28 CVEs）

CVE ID	程序	漏洞类型	CVE ID	程序	漏洞类型
CVE-2020-36375	*MJS v1.20.1*	CWE-674	CVE-2019-6291	*NASM v2.14.03*	CWE-674
CVE-2020-36374	*MJS v1.20.1*	CWE-674	CVE-2019-6290	*NASM v2.14.03*	CWE-674
CVE-2020-36373	*MJS v1.20.1*	CWE-674	CVE-2018-18701	*Binutils v2.31*	CWE-674
CVE-2020-36372	*MJS v1.20.1*	CWE-674	CVE-2018-18700	*Binutils v2.31*	CWE-674
CVE-2020-36371	*MJS v1.20.1*	CWE-674	CVE-2018-18484	*Binutils v2.31*	CWE-674
CVE-2020-36370	*MJS v1.20.1*	CWE-674	CVE-2018-17985	*Binutils v2.31*	CWE-674
CVE-2020-36369	*MJS v1.20.1*	CWE-674	CVE-2020-18899	*Exiv2 v0.27*	CWE-789
CVE-2020-36368	*MJS v1.20.1*	CWE-674	CVE-2019-7704	*Binaryen v1.38.22*	CWE-789
CVE-2020-36367	*MJS v1.20.1*	CWE-674	CVE-2019-7698	*Bento4 v1.5.1-624*	CWE-789
CVE-2020-36366	*MJS v1.20.1*	CWE-674	CVE-2019-7148	*Elfutils v0.175*	CWE-789
CVE-2020-18392	*MJS v1.20.1*	CWE-674	CVE-2018-20652	*Tinyexr v0.9.5*	CWE-789
CVE-2020-18898	*Exiv2 v0.27*	CWE-674	CVE-2018-18483	*Binutils v2.31*	CWE-789
CVE-2019-6293	*Flex v2.6.4*	CWE-674	CVE-2018-20657	*Binutils v2.31*	CWE-401
CVE-2019-6292	*Yaml-cpp v0.6.2*	CWE-674	CVE-2018-20002	*Binutils v2.31*	CWE-401

目　录

CONTENTS

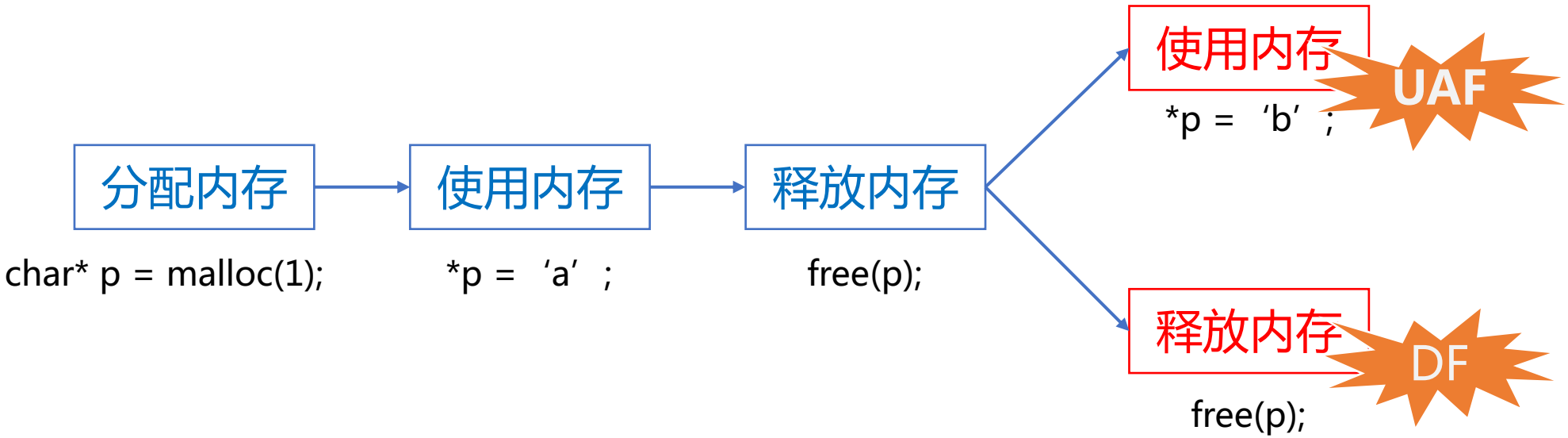
深圳大学
SHENZHEN UNIVERSITY

内存时序漏洞

内存时序漏洞：软件在进行内存操作时违反了内存使用的安全时序规则，在释放内存后继续使用内存或再次释放内存，可能导致未定义行为或程序崩溃。

典型的内存时序漏洞：

- 释放后使用（UaF）
- 双重释放（DF）



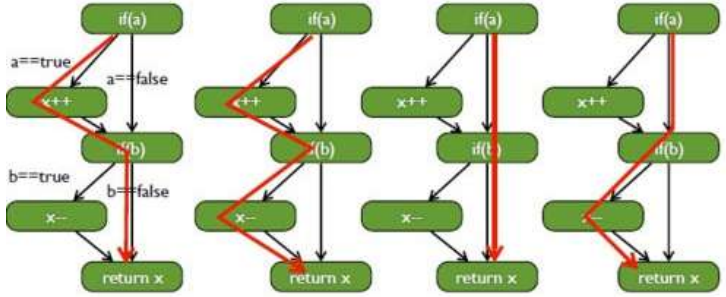
内存时序漏洞检测的难点以及现有方法的局限性

内存时序漏洞：软件在进行内存操作时违反了内存使用的安全时序规则，在释放内存后继续使用内存或再次释放内存，可能导致未定义行为或程序崩溃。

- 难点：这类漏洞的触发往往涉及一系列内存操作，这些内存操作可能并不都位于同一代码块中，仅当以某种特定的顺序执行这一系列内存操作时才触发错误
- 现有方法局限性：
  - 误报率和漏报率高；采用路径敏感的分析与约束求解，可扩展性差（静态）
  - 以控制流图的边覆盖率作为覆盖评估的标准，难以发现以特定时序执行一系列内存操作的输入（动态）

Infer和Cppcheck误报和漏报较高

CBMC采用路径敏感的分析和约束求解，可扩展性较差



动态测试工具（例如AFL）主要关注控制流程图的边覆盖率，而非边的覆盖时序

启发性示例：CVE-2018-20623释放后使用漏洞

按照以下顺序执行一系列特定的内存操作，将触发释放后使用漏洞

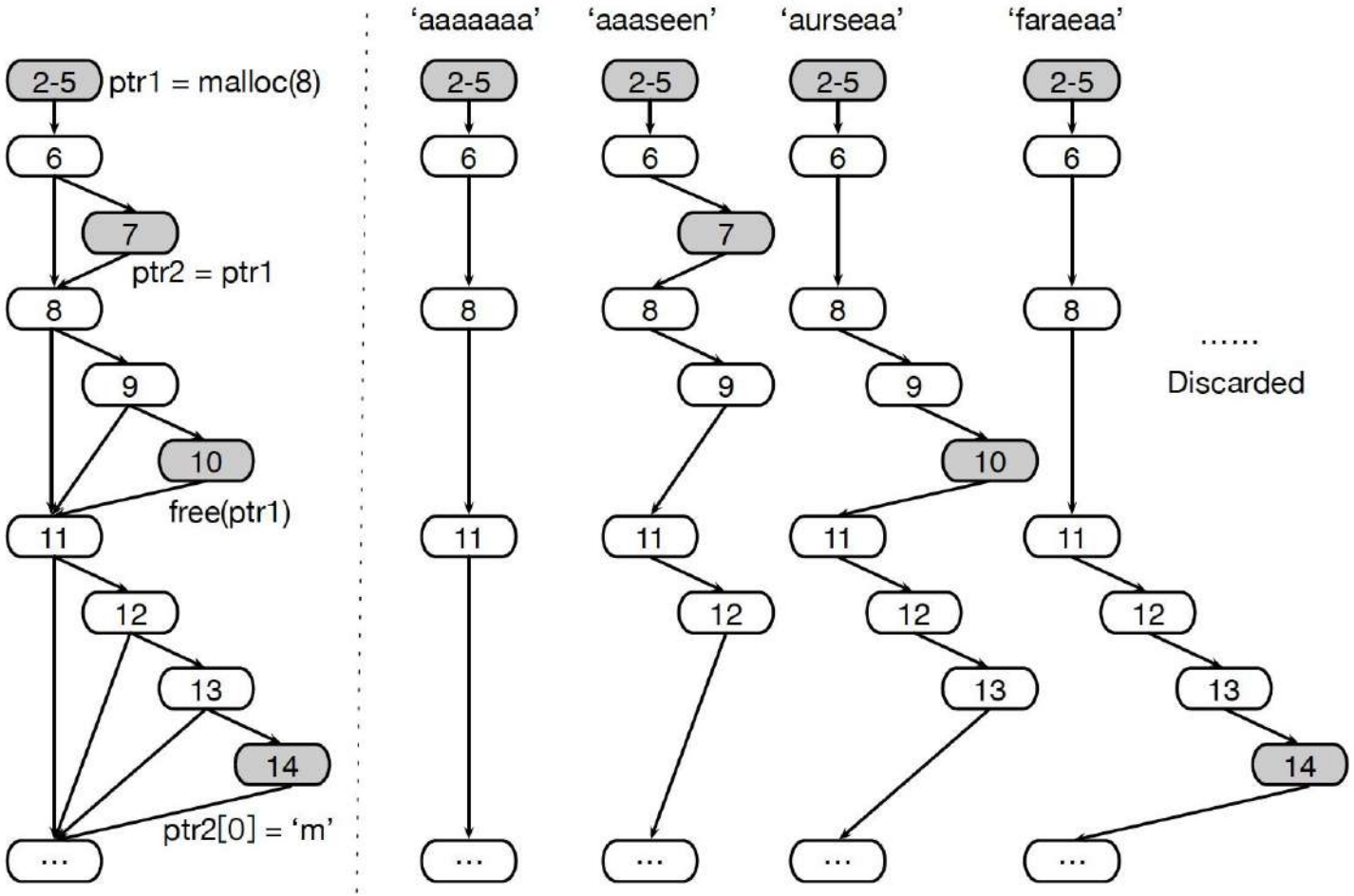
- ✓ 第4行： 分配内存空间
- ✓ 第7行： 指针ptr1和ptr2互为别名
- ✓ 第10行： 释放内存空间
- ✓ 第14行： 使用了已释放的内存空间

```
1  void main() {
2    char buf[7];
3    read(0, buf, 7)
4    char* ptr1 = malloc(8);
5    char* ptr2 = malloc(8);
6    if(buf[5] == 'e')
7      ptr2 = ptr1;
8    if(buf[3] == 's')
9      if(buf[1] == 'u')
10       free(ptr1);
11   if(buf[4] == 'e')
12     if(buf[2] == 'r')
13       if(buf[0] == 'f')
14         ptr2[0] = 'm';
15   ...
16 }
```

该代码片段来自GNU开源工具集 *Binutils v2.31* 中的 *readelf* 程序

代码覆盖导向的模糊测试技术的局限性

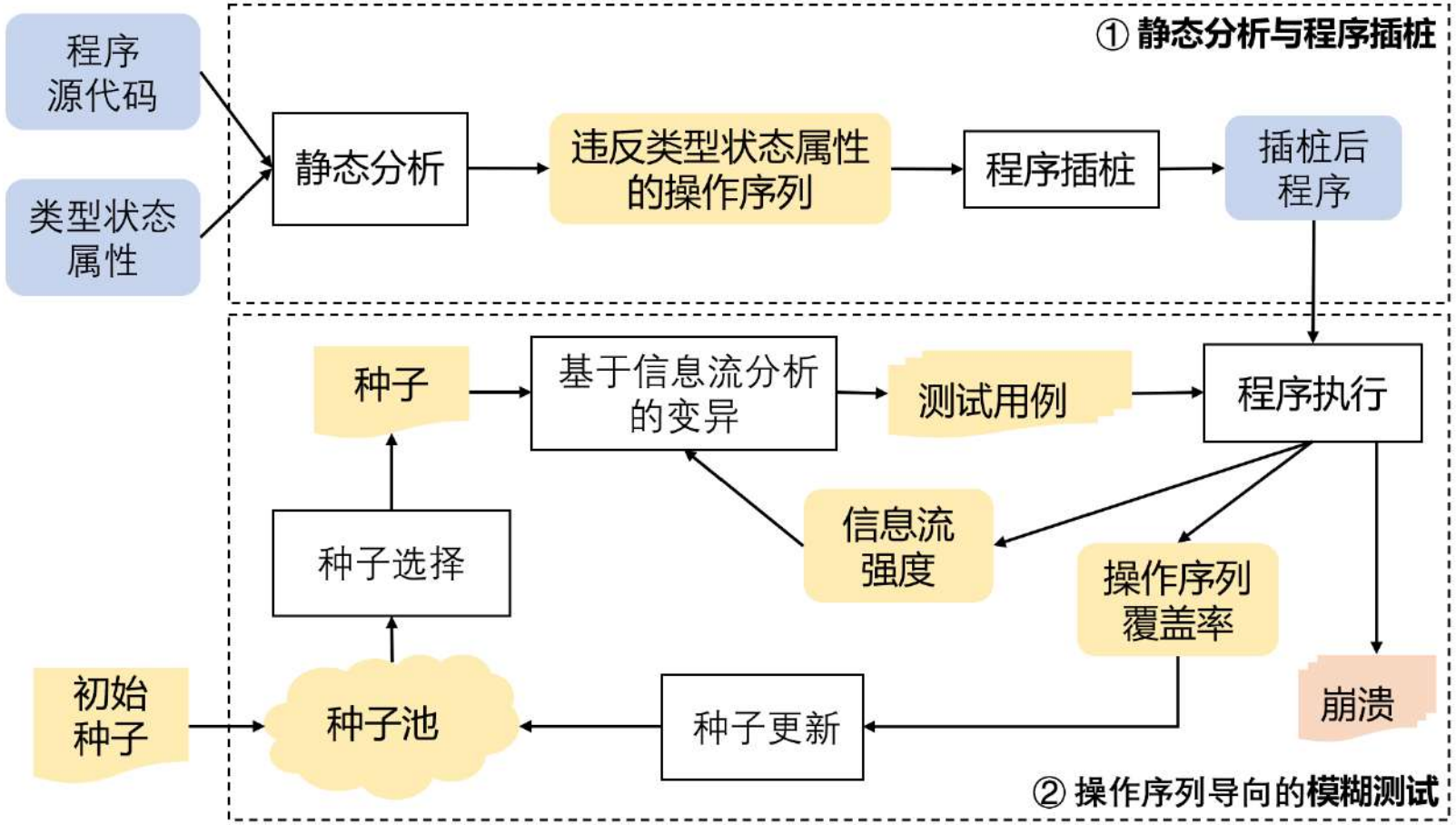
- 即便种子/测试用例覆盖了所有的分支，依然难以触发漏洞



(a) 控制流程图　　(b) AFL的运行过程

本文提出的内存时序漏洞检测方法：UAFL

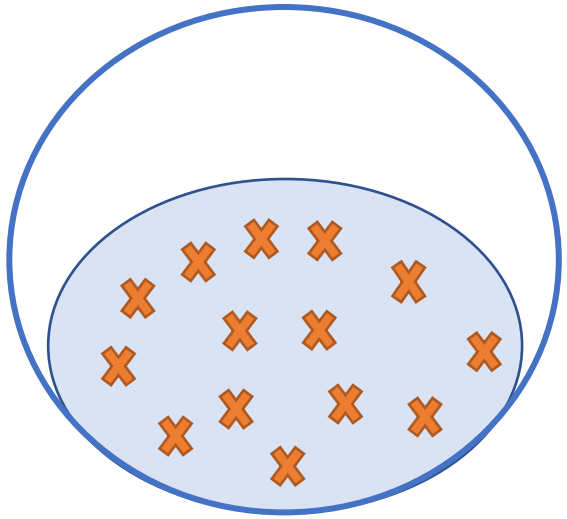
- 采用轻量级的程序静态分析技术识别违反内存时序的安全使用规则的操作序列
- 使用操作序列导向的模糊测试技术逐步生成能触发内存时序漏洞的输入



本文提出的内存时序漏洞检测方法：UAFL

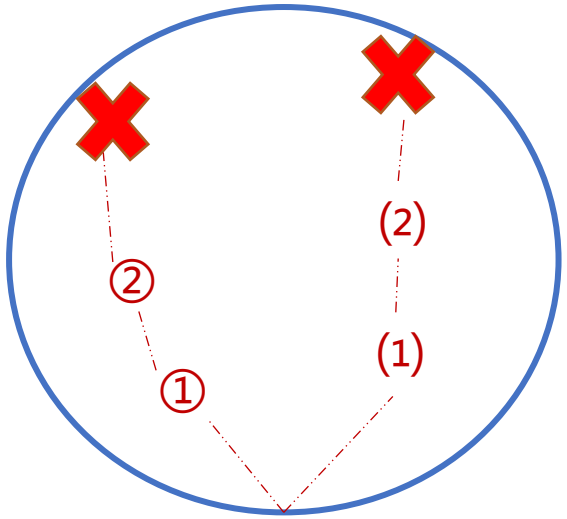
- 采用轻量级的程序静态分析技术识别违反内存时序的安全使用规则的操作序列
- 使用操作序列导向的模糊测试技术逐步生成能触发内存时序漏洞的输入

与代码覆盖导向的模糊测试技术区别：



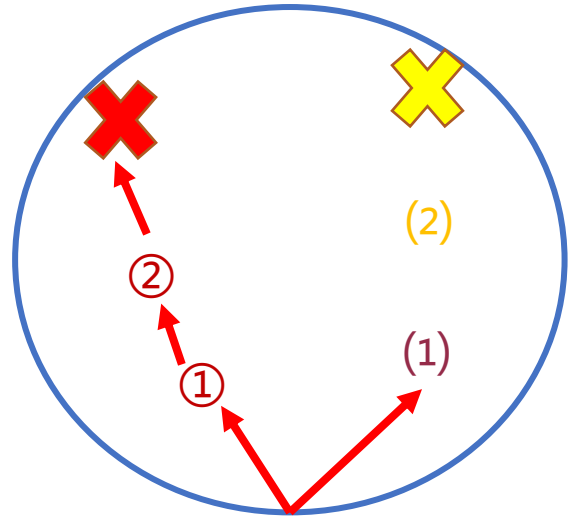
局部海域，遍地撒网

代码覆盖导向的模糊测试



精准定位，指明方向

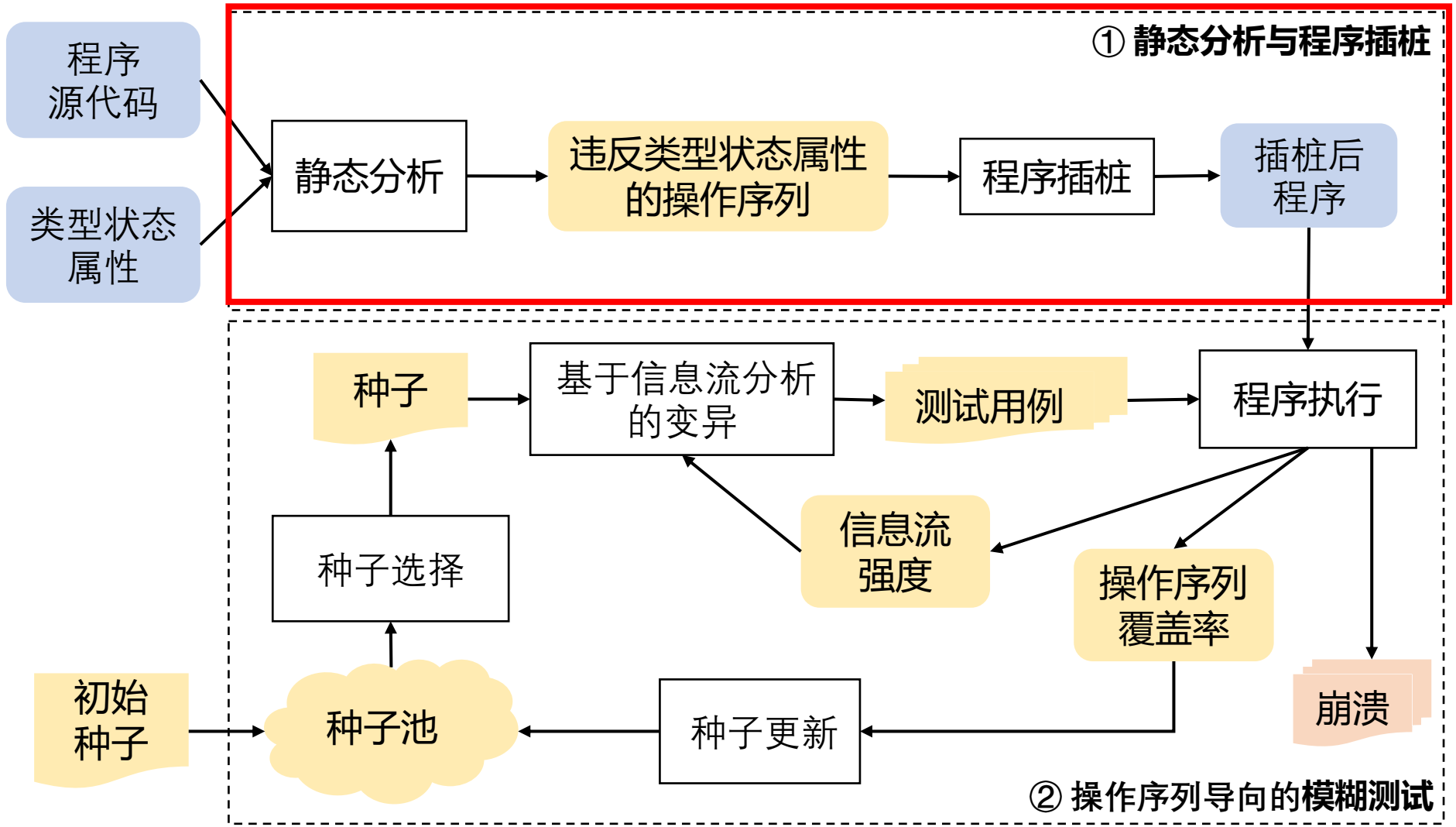
轻量级静态分析



自动导航，足下践行

操作序列导向的模糊测试

UAFL的整体流程

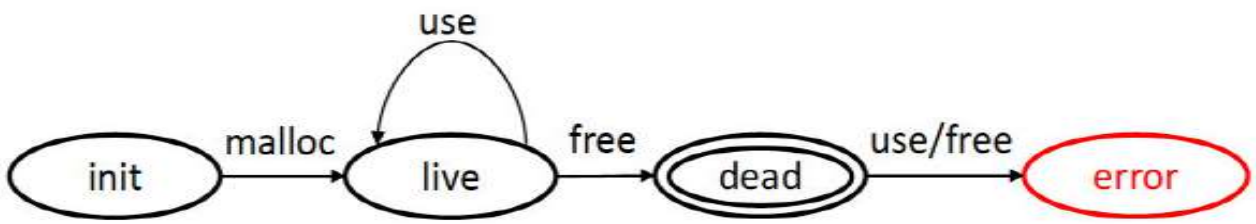

① 静态分析与程序插桩
程序
源代码
类型状态
属性
静态分析
违反类型状态属性
的操作序列
程序插桩
插桩后
程序
种子
基于信息流分析
的变异
测试用例
程序执行
信息流
强度
操作序列
覆盖率
种子选择
初始
种子
种子池
种子更新
崩溃
② 操作序列导向的模糊测试

静态分析与程序插桩（一）

1. 内存使用的安全时序规则建模为具有一定类型状态属性的状态机模型

   内存时序属性：可用自动机$P_m = (Q, \sum, \delta, q_0, Q\backslash\{q_{err}\})$表示，其中

   - $Q = \{\text{init}, \text{live}, \text{dead}, \text{error}\}$
   - $\sum = \{\text{malloc}, \text{use}, \text{free}\}$
   - $Q = \{\text{init}, \text{live}, \text{dead}, \text{error}\}$
   - $q_0 = \{\text{init} \xrightarrow{\text{malloc}} \text{live},\ \text{live} \xrightarrow{\text{use}} \text{live},\ \text{live} \xrightarrow{malloc} \text{dead},\ \text{dead} \xrightarrow{\text{use/free}} \text{error},\ \text{error} \xrightarrow{*} \text{error}\}$
   - $q_{err} = \text{error}$



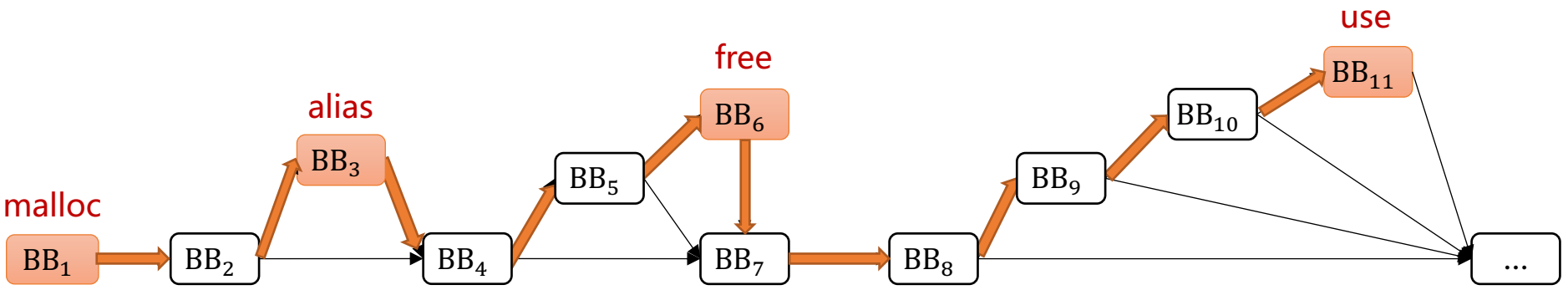
2. 通过安德森指针分析识别操作同一内存对象的指针别名
3. 使用路径不敏感的静态分析技术识别违反内存时序属性的操作序列
4. 通过程序插桩将运行时操作序列的覆盖情况反馈给模糊测试

静态分析与程序插桩（二）

1. 内存使用的安全时序规则建模为具有一定类型状态属性的状态机模型
2. 通过安德森指针分析识别操作同一内存对象的指针别名

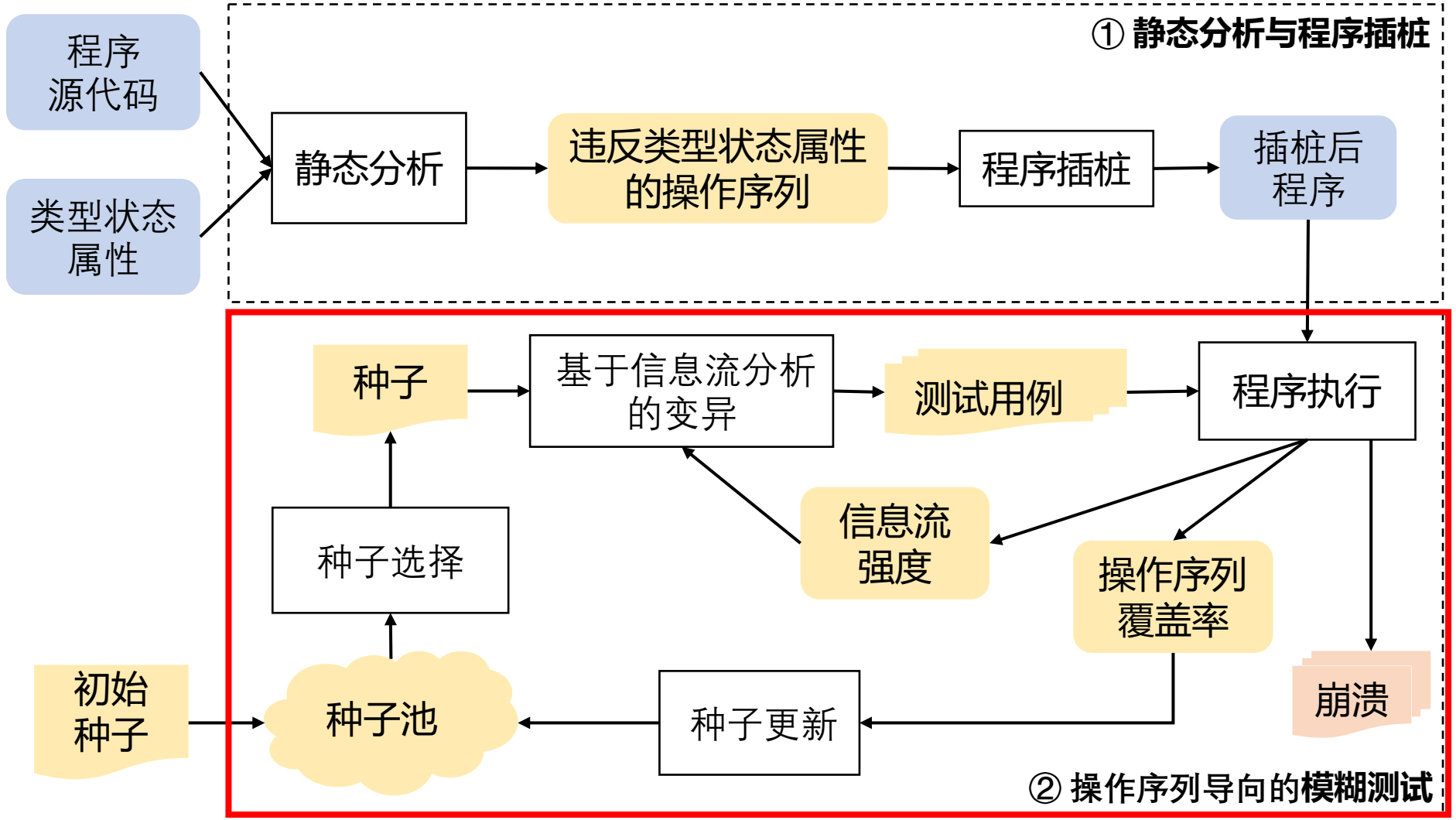
程序语句	指向关系图
$r = *m;$	$m \rightarrow \{\} \rightarrow \{\} \leftarrow r$
$m = \&p;$	$m \rightarrow \{p\} \rightarrow \{\} \leftarrow r$
$P = \&b;$	$m \rightarrow \{p\} \rightarrow \{b\} \leftarrow r$

3. 使用路径不敏感的可达性分析来识别违反内存时序属性的操作序列



4. 通过程序插桩将运行时操作序列的覆盖情况反馈给模糊测试

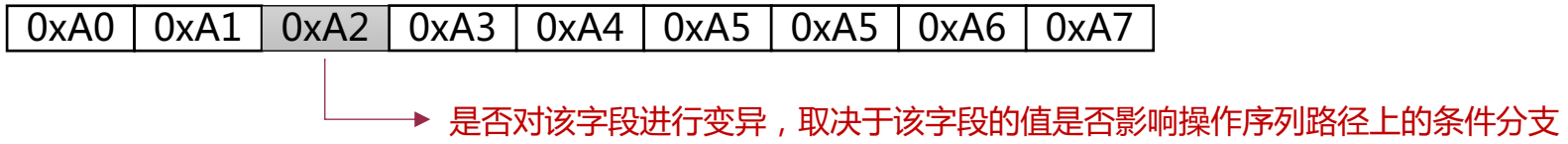
UAFL的整体流程


① 静态分析与程序插桩
程序
源代码
类型状态
属性
静态分析
违反类型状态属性
的操作序列
程序插桩
插桩后
程序
种子
基于信息流分析
的变异
测试用例
程序执行
信息流
强度
操作序列
覆盖率
种子选择
初始
种子
种子池
种子更新
崩溃
② 操作序列导向的模糊测试

操作序列导向的模糊测试

引导测试用例的自动生成不断地朝着更高的操作序列覆盖率迈进

- 操作序列引导的反馈机制
  - 逐步覆盖更多、更完整的操作序列
- 基于信息流分析的变异优化策略
  - 关注输入的哪些字段对覆盖操作序列有影响，对这些字段加大变异力度

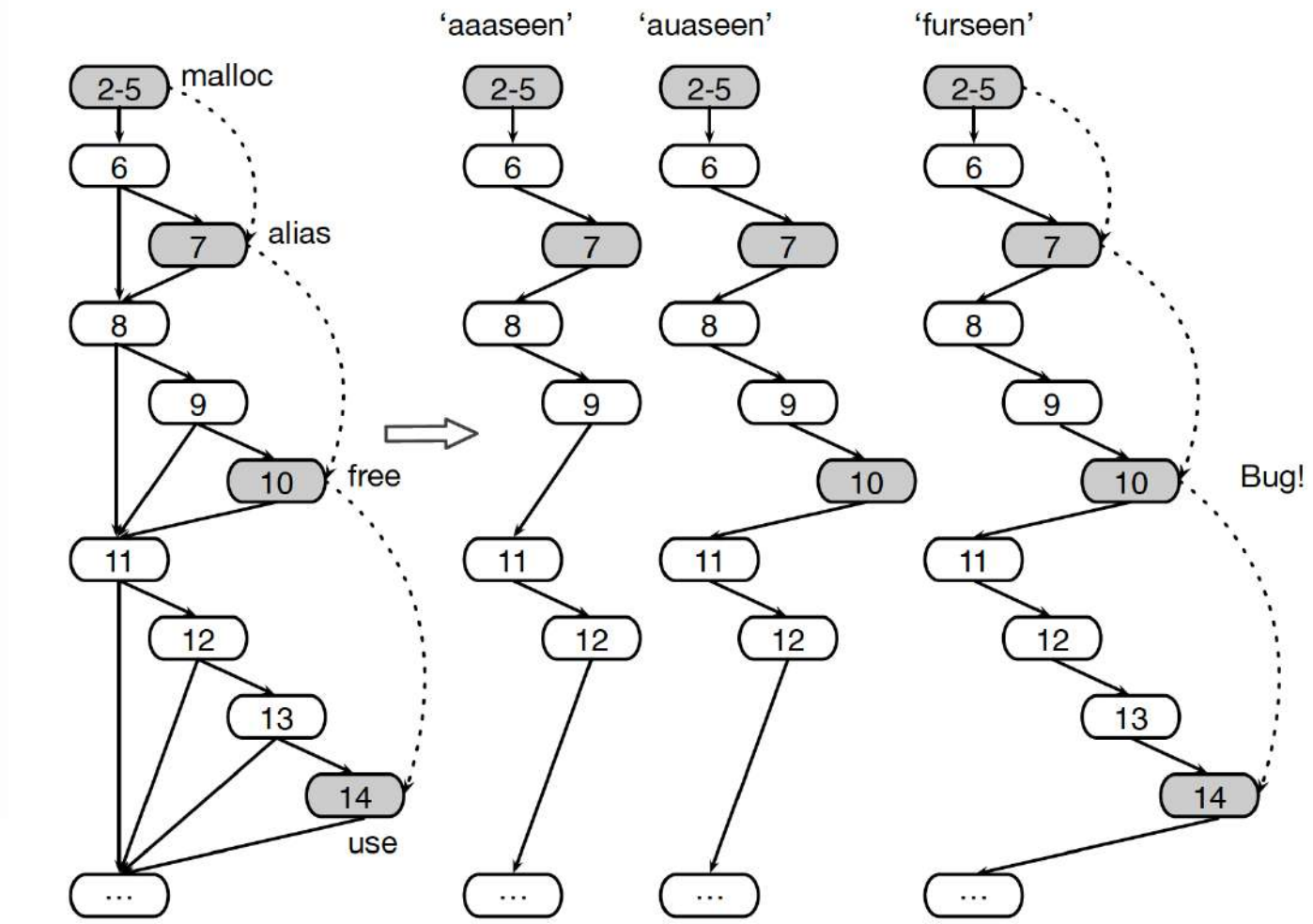


- 种子选择
  - 给予操作序列覆盖率更高的种子更多次数的变异机会

$$assignEnergyOSe(t) = \underbrace{assignEnergy(t)}_{\text{评估种子 } t \text{ 后得到的种子质量值}} * \left(1 + \underbrace{\frac{\#c\_OSe}{\#t\_OSe}}_{\text{操作序列覆盖率}}\right)$$

运行示例：UAFL对CVE-2018-20623漏洞的检测过程

- 逐步覆盖操作序列，直到最终覆盖了完整的操作序列



阶段一：静态分析与程序插桩　　阶段2：操作序列导向的模糊测试

实验评估

UAFL的工具原型基于LLVM平台、SVF工具和AFL工具构建

实验数据集：

- 14个常用的、功能各异的、不同规模的开源应用程序

对比的技术：

- AFL [AFL 2.52b]
- AFLfast [Böhme2017]
- MOpt [Lyu2019]
- FairFuzz [Lemieux2017]
- Angora [Chen2018]
- QSYM [Yun2018]

实验设计：

- 实验一：静态分析性能评估
- 实验二：漏洞检测能力评估
- 实验三：策略有效性评估
- 实验四：覆盖率评估

实验设计：

- 每个实验对象运行24小时，重复8次

实验一：静态分析性能评估

- 操作序列导向的反馈机制和信息流分析策略分别需要对19.2% 与13.3% 的基本块进行插桩
- UAFL 执行的静态分析所引入的时间开销是可接受的，其平均耗时为1,148 秒（0.32 小时）

程序	BB	BB_{UAF}	BB_{IF}	BB_{Free}	操作序列	耗时（秒）
readelf 2.28	16,967	2,681 (15.8%)	1,103 (6.5%)	91	41,605	262
readelf 2.31	19,973	3,647 (18.2%)	1,555 (7.8%)	98	130,102	508
jpegoptim	634	36 (5.7%)	28 (4.4%)	5	44	1
liblouis	2,957	486 (16.4%)	190 (6.4%)	8	422	18
lrzip	9,356	1,051 (11.2%)	467 (5.0%)	6	313	150
Mini XML	4,237	890 (21.0%)	788 (18.6%)	10	486	44
boringssl	22,547	3,701 (16.4%)	3,265 (14.4%)	32	84,069	2,005
GNU cflow	5,095	1,402 (27.5%)	751 (14.7%)	33	4330	30
Boolector	26,866	11,511 (42.8%)	9,031 (33.6%)	4	28,586	2,387
openh264	12,735	2,090 (16.4%)	927 (7.3%)	1	1,219	1,127
libpff	18,569	6,371 (34.3%)	6,041 (32.5%)	60	20,865	122
mjs	4,937	546 (11.0%)	343 (6.9%)	16	1,143	24
ImageMagick	31,190	1,573 (5.0%)	1,336 (4.3%)	3	55,877	2,185
nasm	13,965	3,812 (27.2%)	3,390 (24.2%)	2	3,357	2,210
Avg.	13,573	2,842 (19.2%)	2,087 (13.3%)	26	26,601	1,148

实验二：漏洞检测能力评估

- UAFL在发现内存时序漏洞的数量上要比其它方法多于25%
- UAFL在发现内存时序漏洞速度上相比其它方法实现了至少2.63倍的提速

程序	漏洞标识符	漏洞类型	发现漏洞所需要的时间消耗（单位：小时）							
			UAFL	$UAFL_{NIF}$	AFL	AFLFast	FairFuzz	MOpt	Angora	QSYM
readelf 2.28	CVE-2017-6966	UaF	**0.59**	1.32	6.09	1.43	0.68	3.61	T/O	6.20
readelf 2.31	CVE-2018-20623	UaF	0.10	0.10	0.10	0.10	T/O	0.10	**0.02**	0.10
jpegoptim	CVE-2018-11416	DF	**0.09**	0.10	0.59	0.88	1.08	1.49	T/O	1.95
liblouis	CVE-2017-13741	UaF	**1.11**	1.81	15.81	T/O	6.96	17.38	T/O	13.42
lrzip	CVE-2018-11496	UaF	**0.01**	0.01	0.01	0.01	0.01	0.01	0.01	0.01
Mini XML	CVE-2018-20592	UaF	**0.38**	0.93	1.28	2.59	0.54	16.7	T/O	18.99
boringssl	Google Test-suit	UaF	**0.33**	1.06	T/O	T/O	4.67	7.62	–	T/O
GNU cflow	CVE-2019-16165	UaF	**1.80**	12.21	23.29	T/O	20.02	T/O	T/O	T/O
Boolector	uaf-issue-1	UaF	0.83	0.97	1.09	0.82	**0.39**	1.66	–	1.16
openh264	uaf-issue-2	UaF	**8.17**	13.00	15.80	11.15	8.17	15.39	T/O	18.45
libpff	CVE-2020-18897	UaF	**1.39**	1.39	4.21	4.11	3.98	4.35	T/O	4.98
mjs	uaf-issue-3	UaF	**1.21**	1.23	3.10	3.02	1.45	4.6	T/O	6.71
ImageMagick	CVE-2019-15140	UaF	**6.29**	13.92	T/O	T/O	T/O	T/O	T/O	T/O
nasm	CVE-2018-19216	UaF	**2.59**	4.69	8.32	3.45	2.86	11.46	2.75	9.64
	CVE-2018-20535	UaF	**17.03**	T/O	T/O	T/O	T/O	T/O	T/O	T/O
漏报的漏洞数量			0	1	3	5	3	3	10	4
平均时间消耗			2.79 + 0.32	5.12 + 0.32	10.11	9.84	8.18	10.42	18.67	11.84
UAFL 在发现漏洞速度上的提升			—	1.75×	3.25×	3.16×	2.63×	3.35×	6.00×	3.80×
$UAFL_{NIF}$ 在发现漏洞速度上的提升			—	—	1.86×	1.81×	1.50×	1.92×	3.43×	2.18×

* UaF 和 DF 分别代表 use-after-free 和 double-free 漏洞的缩写；T/O 代表超时

实验三：策略有效性评估

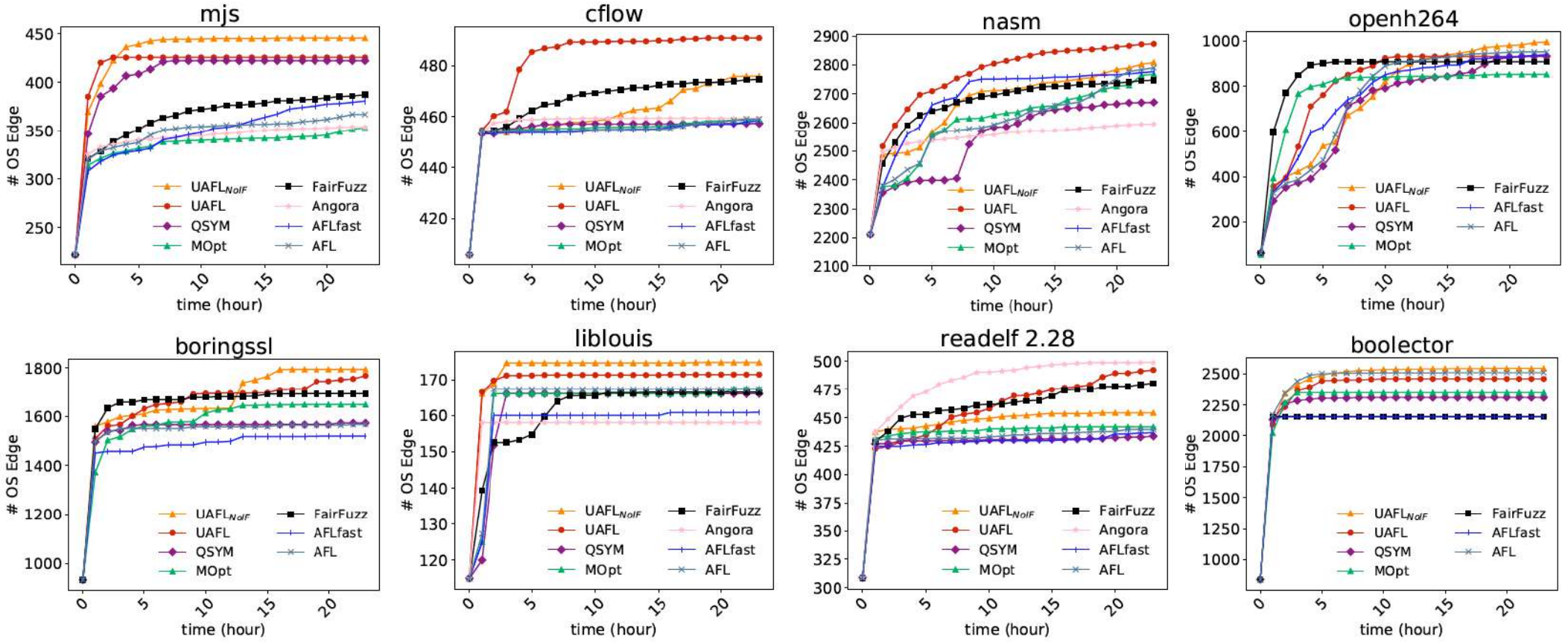
- 操作序列导向的反馈机制使得UAFL发现内存时序漏洞的能力更强
- 基于信息流分析的变异优化策略提升了UAFL发现内存时序漏洞的效率

程序	漏洞标识符	漏洞类型	发现漏洞所需要的时间消耗（单位：小时）							
			UAFL	$UAFL_{NIF}$	AFL	AFLFast	FairFuzz	MOpt	Angora	QSYM
readelf 2.28	CVE-2017-6966	UaF	**0.59**	1.32	6.09	1.43	0.68	3.61	T/O	6.20
readelf 2.31	CVE-2018-20623	UaF	0.10	0.10	0.10	0.10	T/O	0.10	**0.02**	0.10
jpegoptim	CVE-2018-11416	DF	**0.09**	0.10	0.59	0.88	1.08	1.49	T/O	1.95
liblouis	CVE-2017-13741	UaF	**1.11**	1.81	15.81	T/O	6.96	17.38	T/O	13.42
lrzip	CVE-2018-11496	UaF	**0.01**	0.01	0.01	0.01	0.01	0.01	0.01	0.01
Mini XML	CVE-2018-20592	UaF	**0.38**	0.93	1.28	2.59	0.54	16.7	T/O	18.99
boringssl	Google Test-suit	UaF	**0.33**	1.06	T/O	T/O	4.67	7.62	–	T/O
GNU cflow	CVE-2019-16165	UaF	**1.80**	12.21	23.29	T/O	20.02	T/O	T/O	T/O
Boolector	uaf-issue-1	UaF	0.83	0.97	1.09	0.82	**0.39**	1.66	–	1.16
openh264	uaf-issue-2	UaF	**8.17**	13.00	15.80	11.15	8.17	15.39	T/O	18.45
libpff	CVE-2020-18897	UaF	**1.39**	1.39	4.21	4.11	3.98	4.35	T/O	4.98
mjs	uaf-issue-3	UaF	**1.21**	1.23	3.10	3.02	1.45	4.6	T/O	6.71
ImageMagick	CVE-2019-15140	UaF	**6.29**	13.92	T/O	T/O	T/O	T/O	T/O	T/O
nasm	CVE-2018-19216	UaF	**2.59**	4.69	8.32	3.45	2.86	11.46	2.75	9.64
	CVE-2018-20535	UaF	**17.03**	T/O	T/O	T/O	T/O	T/O	T/O	T/O
漏报的漏洞数量			0	1	3	5	3	3	10	4
平均时间消耗			2.79 + 0.32	5.12 + 0.32	10.11	9.84	8.18	10.42	18.67	11.84
UAFL 在发现漏洞速度上的提升			—	1.75×	3.25×	3.16×	2.63×	3.35×	6.00×	3.80×
$UAFL_{NIF}$ 在发现漏洞速度上的提升			—	—	1.86×	1.81×	1.50×	1.92×	3.43×	2.18×

* UaF 和 DF 分别代表 use-after-free 和 double-free 漏洞的缩写；T/O 代表超时

实验四：覆盖率评估

- UAFL达成的操作序列覆盖率更高，这也是其在检测内存时序漏洞方面有效的主要原因



阶段性研究成果（二）

- 撰写的论文《Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities》已在软件工程领域的顶级国际会议ICSE ‘20上发表。（与NTU合作）
- 在被广泛使用的真实应用程序中新发现了7个安全攸关的内存时序漏洞（7 CVEs），包括6个释放后使用漏洞和1个双重释放漏洞。这7个内存时序漏洞现已经被全部修复。

表 4-5 UAFL 新发现的内存时序漏洞（7 CVEs）

CVE ID	程序	漏洞类型
CVE-2020-18897	*libpff 894a4*	CWE-416: Use After Free
CVE-2019-16165	*GNU cflow v1.6*	CWE-416: Use After Free
CVE-2019-15140	*ImageMagick v7.0.8*	CWE-416: Use After Free
CVE-2019-11471	*libheif v1.4.0*	CWE-416: Use After Free
CVE-2019-7703	*Binaryen v1.38.22*	CWE-416: Use After Free
CVE-2018-20592	*Mini-XML v2.12*	CWE-416: Use After Free
CVE-2018-16402	*Elfutils v0.173*	CWE-415: Double Free

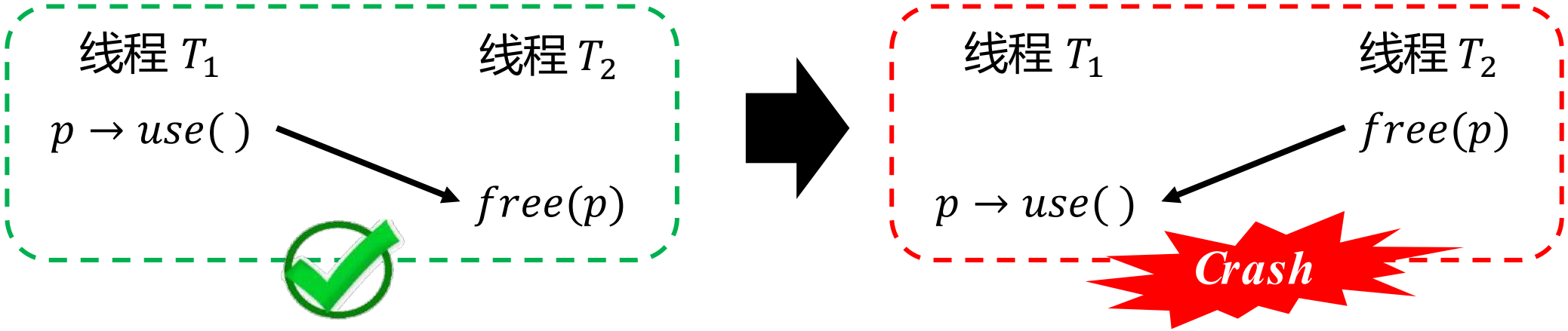
目　录

CONTENTS

深圳大学
SHENZHEN UNIVERSITY

内存并发漏洞

内存并发漏洞：两个或两个以上的线程因线程交错，交互地作用于一个或多个共享内存，引起程序崩溃或挂起，或产生与串行执行不同的结果。



未触发内存错误的线程交错

触发内存错误的线程交错

典型的内存并发漏洞：

- 并发释放后使用、并发双重释放、并发指针空解引用、死锁、竞态条件

内存并发漏洞检测的难点

内存并发漏洞：两个或两个以上的线程因线程交错，交互地作用于一个或多个共享内存，引起程序崩溃或挂起，或产生与串行执行不同的结果。

- 并发程序相比串行程序的复杂度更高，并发漏洞难理解、难检测、难复现。
- 难点：
  - 执行路径具有不确定性：一些潜在漏洞的触发是随机发生的
  - 执行交错的空间十分庞大：并发漏洞通常仅在罕见的、特定的执行交错下才暴露出来

如果两个线程分别具有m和n条指令，那么所有可能的线程交错数量是：

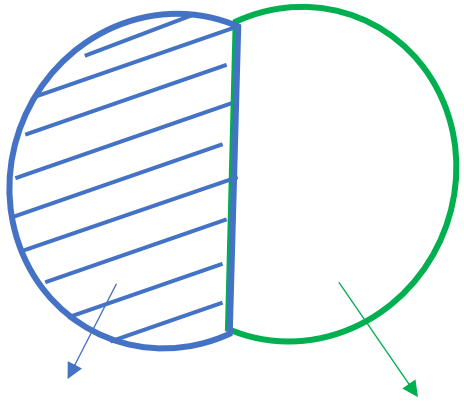
$$\frac{(\boldsymbol{m}+\boldsymbol{n})!}{\boldsymbol{m}!\times\boldsymbol{n}!}$$

$m = n = 4$	$m = n = 8$	$m = n = 16$
70	13K	601M

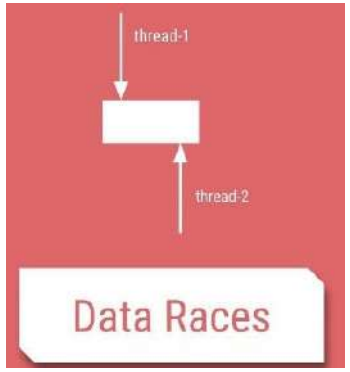
内存并发漏洞检测技术研究现状

现有的程序分析与测试技术对于并发漏洞检测有着很大的局限性：

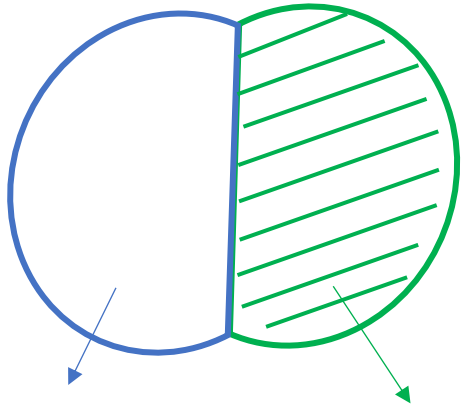
- 面向串行程序的程序分析技术不适用于并发程序，缺乏对多线程并发执行语义的建模
- 传统的压力测试和模糊测试可能运行数天甚至数月也难以发现某些并发漏洞
- 面向并发程序的漏洞检测研究工作大多关注数据竞争的检测，对线程交错空间的探索不足

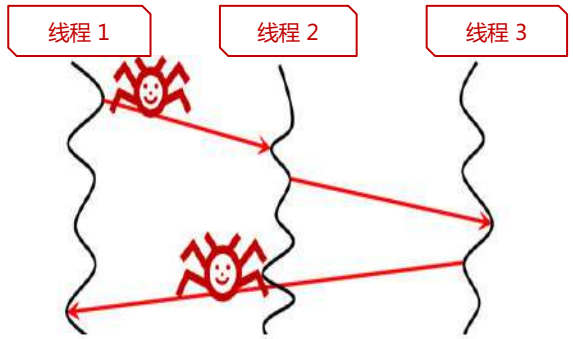


面向串行程序的分析与测试方法
（MemLock、UAFL）



并发程序分析方法
（TSAN、FastTrack）

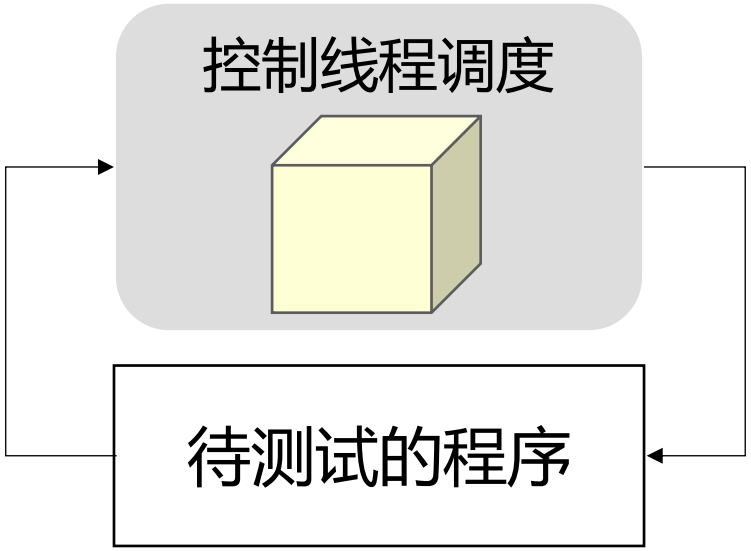


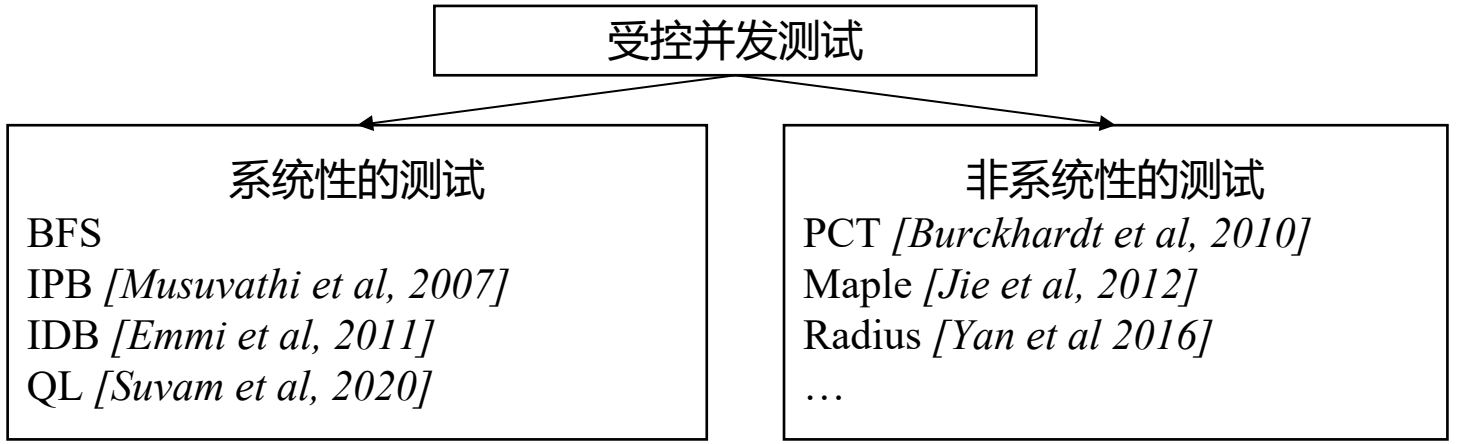


本章的系统性并发测试方法
（PERIOD）

受控并发测试（Controlled Concurrency Testing）

受控并发测试技术通过控制线程调度，探索并发程序的调度空间，每次测试执行一个不同的线程交错，并监视其执行是否发生异常。





- 在某种约束和规则下（e.g., DFS）遍历所有可能的线程交错
- 能在某个范围内保证程序的线程交错情况的正确性

- 通过随机调度的方案对程序进行重复性测试
- 概率性地发现程序中的错误

工业界近期案例：

- Amazon使用受控并发测试技术检查S3存储桶的并发安全性（SOSP 2021）
- 微软使用基于受控并发测试的单元测试工具Coyote对Azure服务的并发执行交错进行测试

启发性示例： CVE-2016-1972内存并发漏洞

```
1  static pthread_mutex_t* lock;
2  static long waiters = 0;
3  static int done = 0;
4
5  void *once(void *) {
6      if(done)
7          return 0;
8      ++waiters;
9      pthread_mutex_lock(lock);
10     // do some thing ...
11     if(!done)
12         done = 1;
13     pthread_mutex_unlock(lock);
14     if(!--waiters) {
15         free(lock);
16         lock = NULL;
17     }
18 }
19 void main() {
20     lock = malloc(sizeof(pthread_mutex_t));
21     pthread_t T0, T1;
22     pthread_create(&T0, NULL, once, NULL);
23     pthread_create(&T1, NULL, once, NULL);
24     pthread_join(T1, NULL);
25     pthread_join(T0, NULL);
26 }
```

该代码片段简化自 *Firefox* 浏览器中的库 *libvpx*

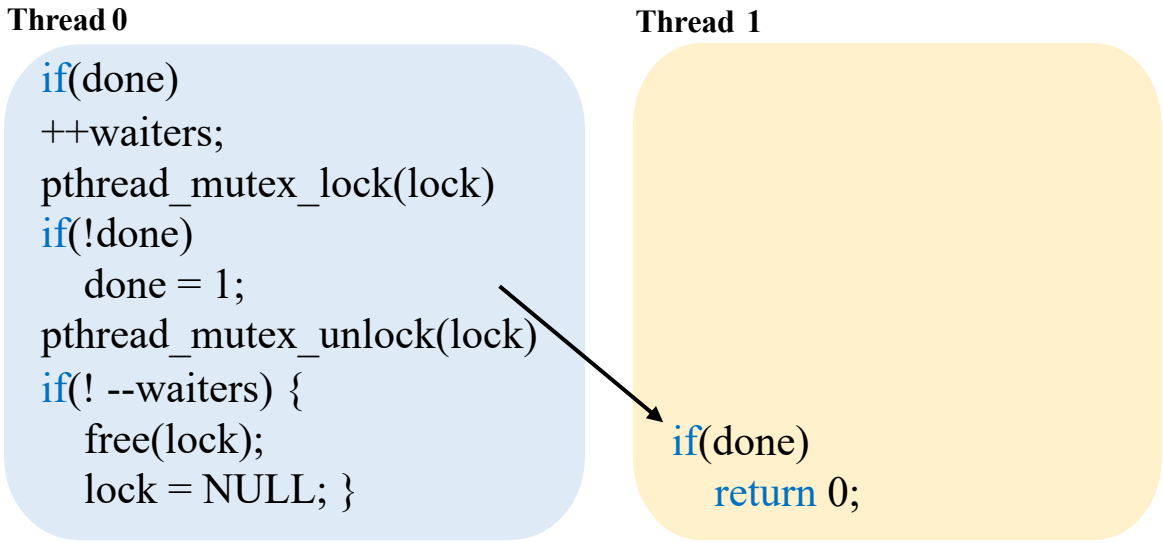
通过三个共享变量进行线程间的同步操作:

- *lock , waiters , done*

传统测试技术难以发现该程序中的问题：

- 传统的压力测试难以覆盖特定的并发交错（覆盖率低）
- ASAN、TSAN等消毒剂仅在触发错误时才报告错误

测试过程中常见的执行交错（仅一次线程切换）：



三种不同类型的内存并发漏洞的触发条件

```
if(done);
```

```
            if(done)
            ++waiters;
            mutex_lock(lock);
            if(!done)
              done=1;
            mutex_unlock(lock);
            if(!--waiters) {
              free(lock);
              lock = NULL;}
```

```
++waiters;
mutex_lock(lock);
if(!done)
mutex_unlock(lock);
if(!--waiters) {
  free(lock);
  lock = NULL;
}
```

NPD

```
if(done);
++waiters;
mutex_lock(lock);
if(!done)
```

```
            if(done)
```

```
  done=1;
mutex_unlock(lock);
if(!--waiters) {
  free(lock);
```

```
            ++waiters;
            mutex_lock(lock);
            if(!done)
            mutex_unlock(lock);
            if(!--waiters) {
              free(lock);
              lock = NULL;
            }
```

```
  lock = NULL;
}
```

UAF

```
if(done);
++waiters;
mutex_lock(lock);
if(!done)
```

```
            if(done)
```

```
  done=1;
mutex_unlock(lock);
if(!--waiters) }
```

```
            ++waiters;
            mutex_lock(lock);
            if(!done)
            mutex_unlock(lock);
            if(!--waiters) {
              free(lock);
```

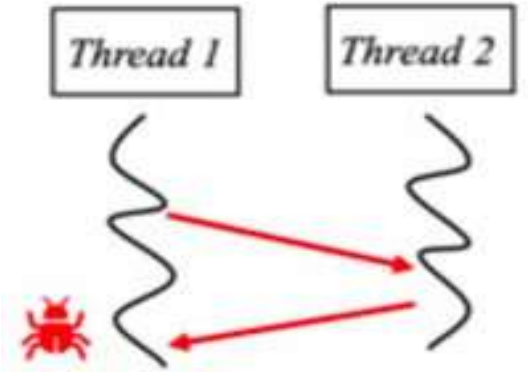
```
  free(lock);
  lock = NULL;
}
```

```
              lock = NULL;
            }
```

DF

这三种并发错误分别要求不同的线程切换次数才能触发:

- 空指针解引用（NPD）
  - *至少需要 2 次线程切换*
- 释放后使用（UAF）
  - *至少需要 4 次线程切换*
- *双重释放*（DF）
  - *至少需要 5 次线程切换*



传统的受控并发测试方法

通过控制程序的线程调度，（系统性/随机）遍历程序中所有语句的线程交错

```
if(done);
++waiters;
mutex_lock(lock);
if(!done)
  done=1;
mutex_unlock(lock);
if(!--waiters) {
  free(lock);
  lock = NULL; }
```

```
            if(done)
            ++waiters;
            mutex_lock(lock);
            if(!done)
              done=1;
            mutex_unlock(lock);
            if(!--waiters) {
              free(lock);
              lock = NULL;}
if(done);
++waiters;
mutex_lock(lock);
if(!done)
  done=1;
mutex_unlock(lock);
if(!--waiters) {
  free(lock);
  lock = NULL; }
```

```
if(done);
++waiters;
mutex_lock(lock);
if(!done)
  done=1;
mutex_unlock(lock);
if(!--waiters) {
  free(lock);
 if(done)
            ++waiters;
            mutex_lock(lock);
            if(!done)
              done=1;
            mutex_unlock(lock);
            if(!--waiters) {
              free(lock);
              lock = NULL;}
  lock = NULL;
}
```

```
if(done);
++waiters;
mutex_lock(lock);
if(!done)
  done=1;
mutex_unlock(lock);
if(!--waiters) {
 if(done)
            ++waiters;
            mutex_lock(lock);
            if(!done)
              done=1;
            mutex_unlock(lock);
            if(!--waiters) {
              free(lock);
              lock = NULL;}
  free(lock);
  lock = NULL;
}
```

```
if(done);
++waiters;
mutex_lock(lock);
if(!done)
  done=1;
mutex_unlock(lock);
 if(done)
            ++waiters;
            mutex_lock(lock);
            if(!done)
              done=1;
            mutex_unlock(lock);
            if(!--waiters) {
              free(lock);
              lock = NULL;}
if(!--waiters) {
  free(lock);
  lock = NULL;
```

所有可能的线程交错数量是：

$$\frac{(9+9)!}{9!\times 9!} = 48,620$$

本文方法约减后的线程交错数量：

≈ 600

挑战一：如何主动控制线程调度？

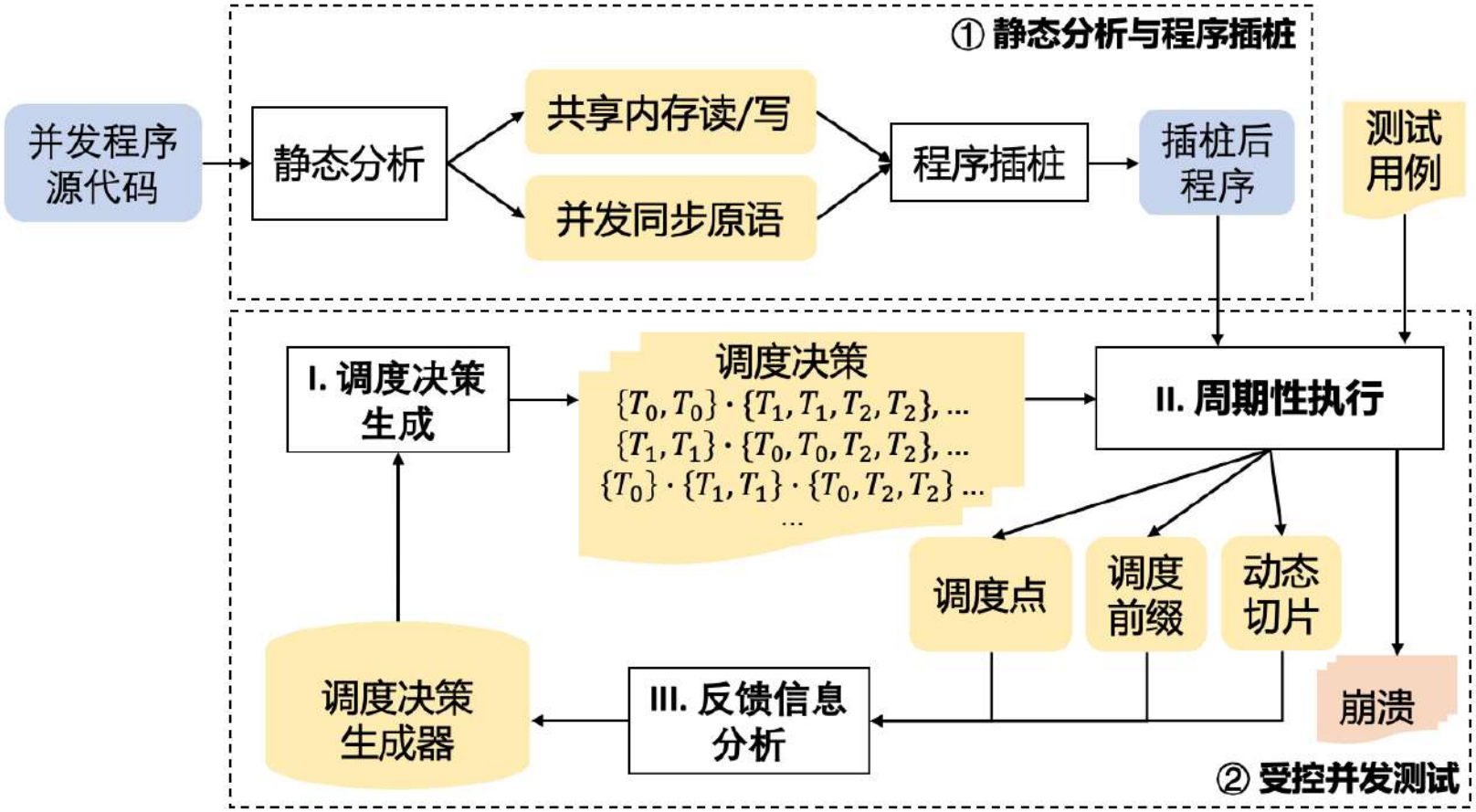
序列化执行；信号量/锁；优先级调度？

挑战二：如何有效探索线程交错？

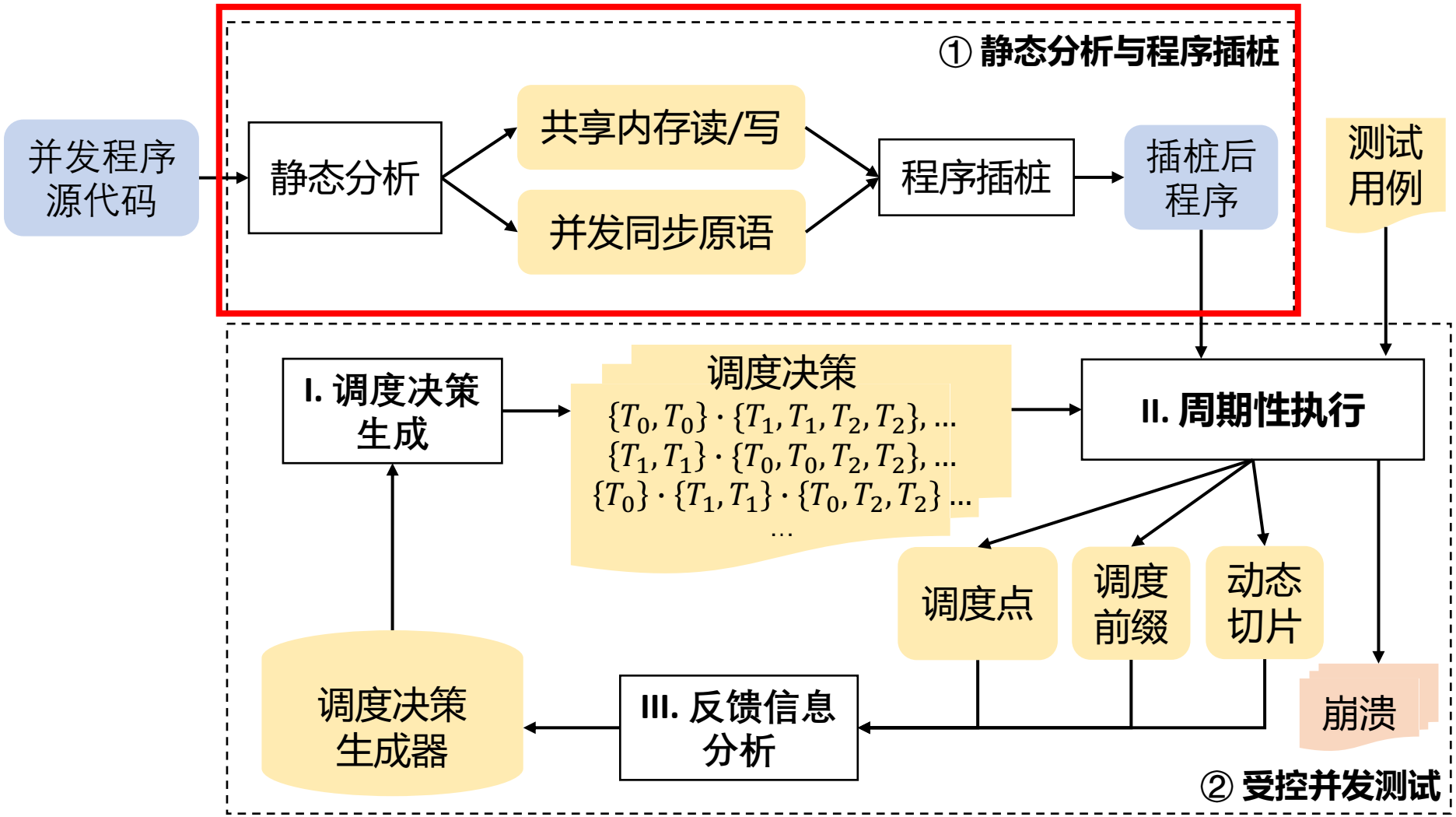
调度点分析；随机探索；系统性探索？

本文提出的内存并发漏洞检测方法：PERIOD

- 采用轻量级的程序静态分析技术识别可能导致并发漏洞的的调度点
- 使用受控并发测试技术系统性地探索调度点之间的执行交错



PERIOD的整体工作流程

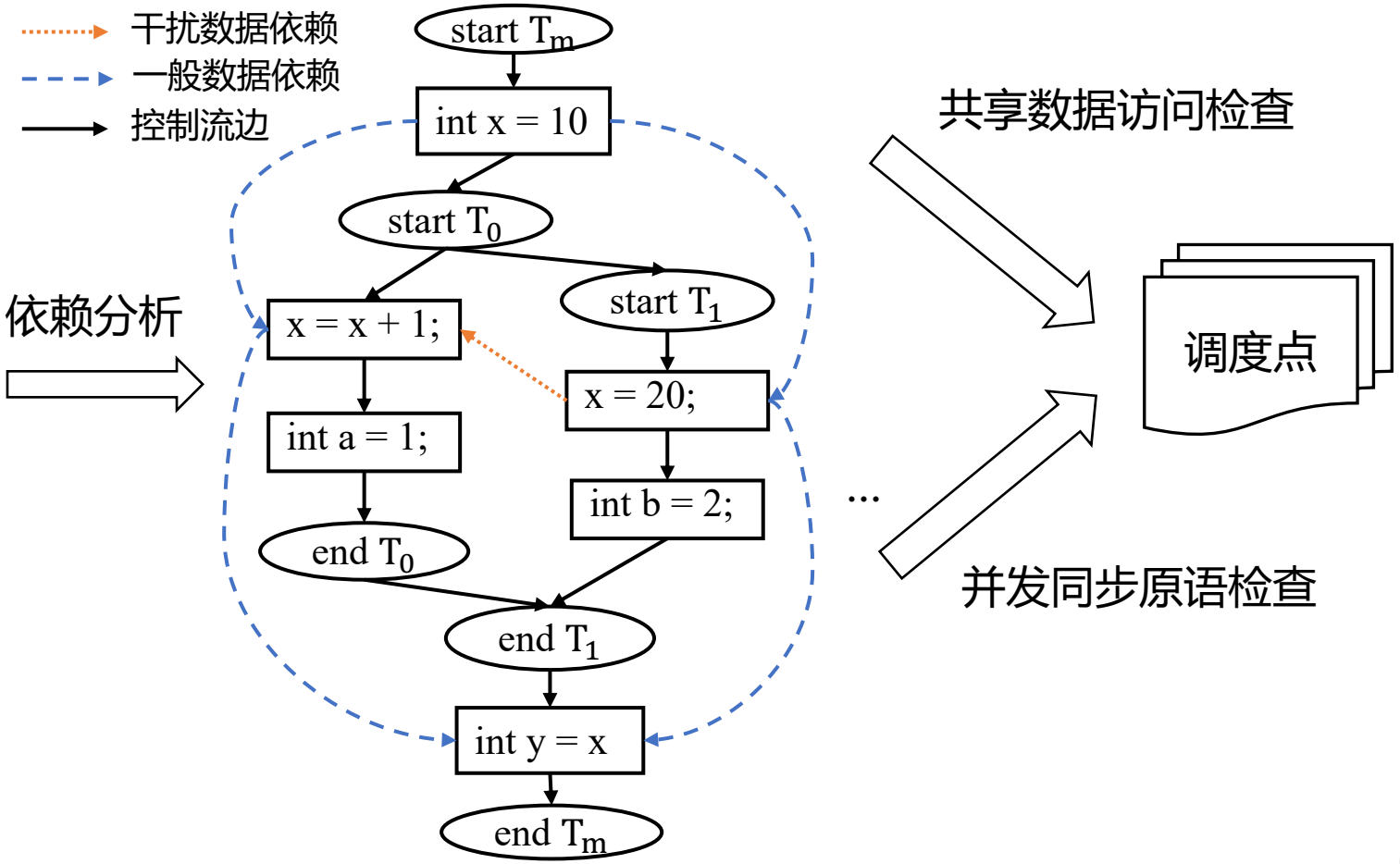


静态分析识别调度点

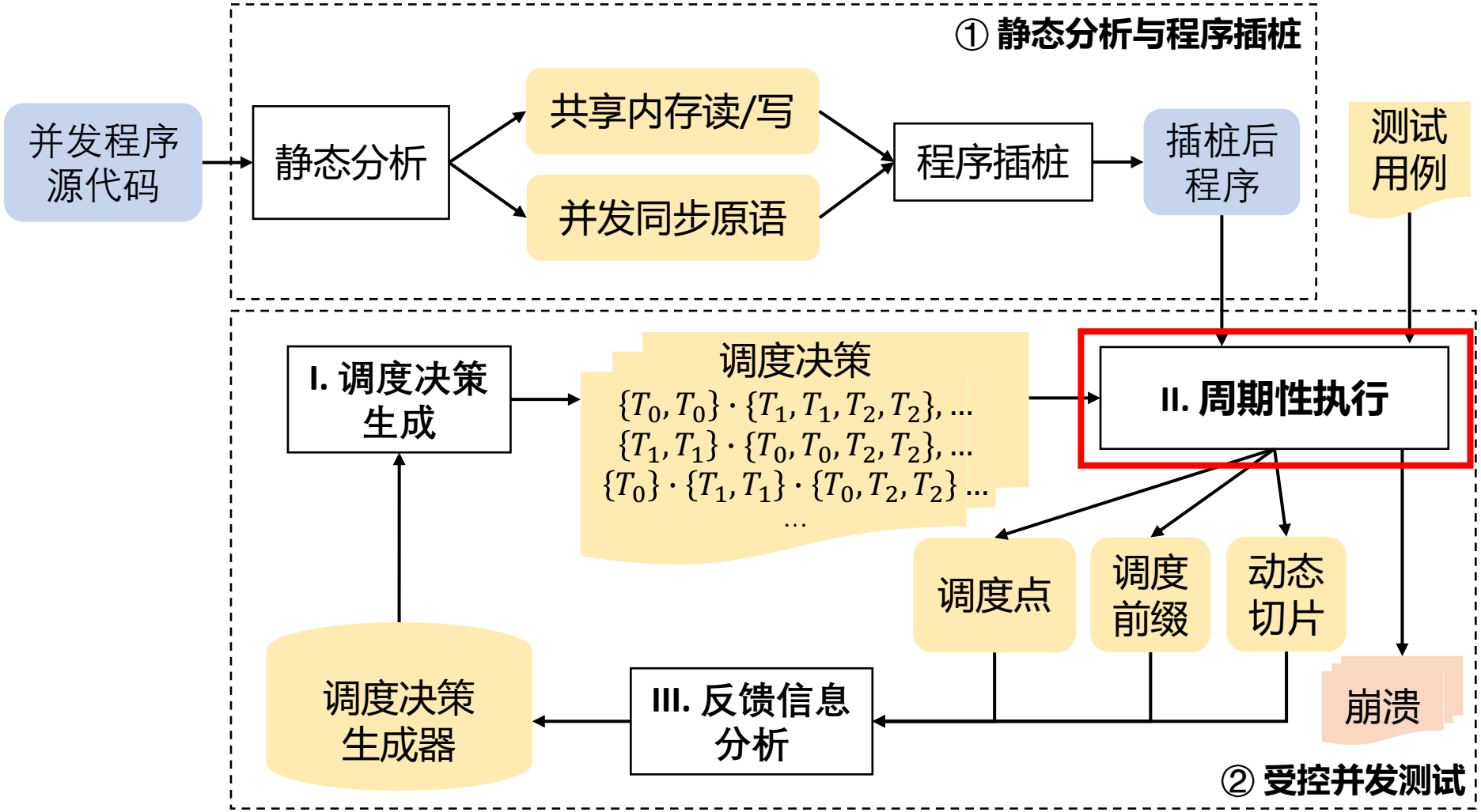
- 调度点是指线程中对共享内存进行读/写访问或使用了并发同步原语的程序语句。
- 将线程调度的范围限制在调度点上，可以避免大量不必要的探索。

```
int x =10;                    //共享数据

void f() {
  x = x +1;                   //调度点
  int a = 1; }                //非调度点

void g() {
  x =  20;                    //调度点
  int b = 2; }                //非调度点

void main() {
  pthread_create(&T0, null, f, null);
  pthread_create(&T1, null, g, null);
  pthread_join(T0);
  pthread_join(T1);
  int y = x;
}
```

并发程序



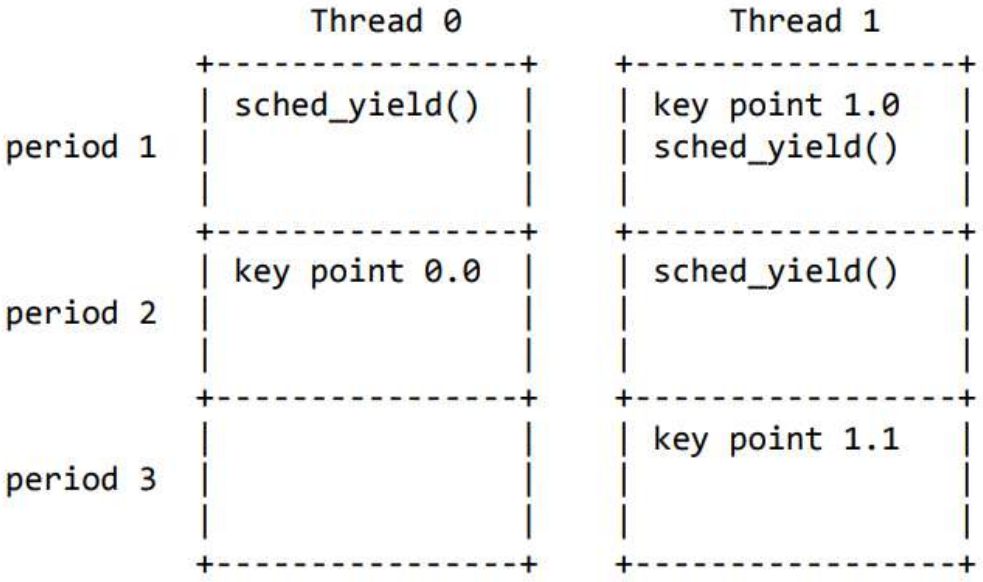
PERIOD的整体工作流程



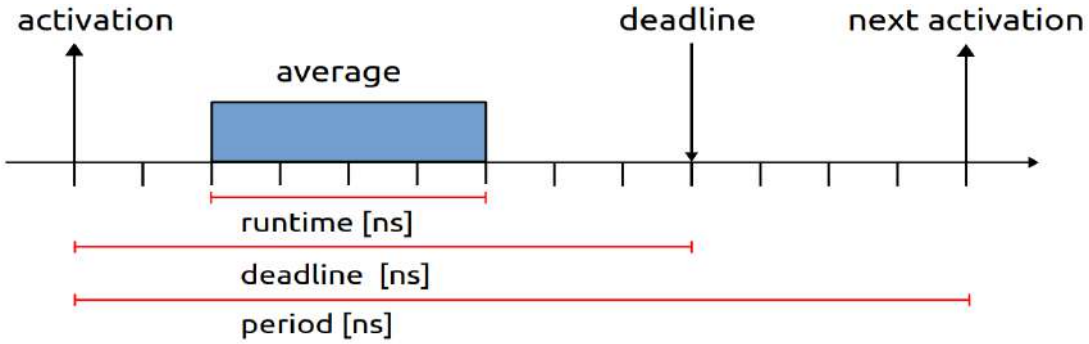
周期性执行

本文将并发程序的执行建模成一个多周期的执行过程

- 仅当一个执行周期结束后，下一个执行周期才开始
- 通过将调度点分配到不同的执行周期来控制它们的相对执行顺序

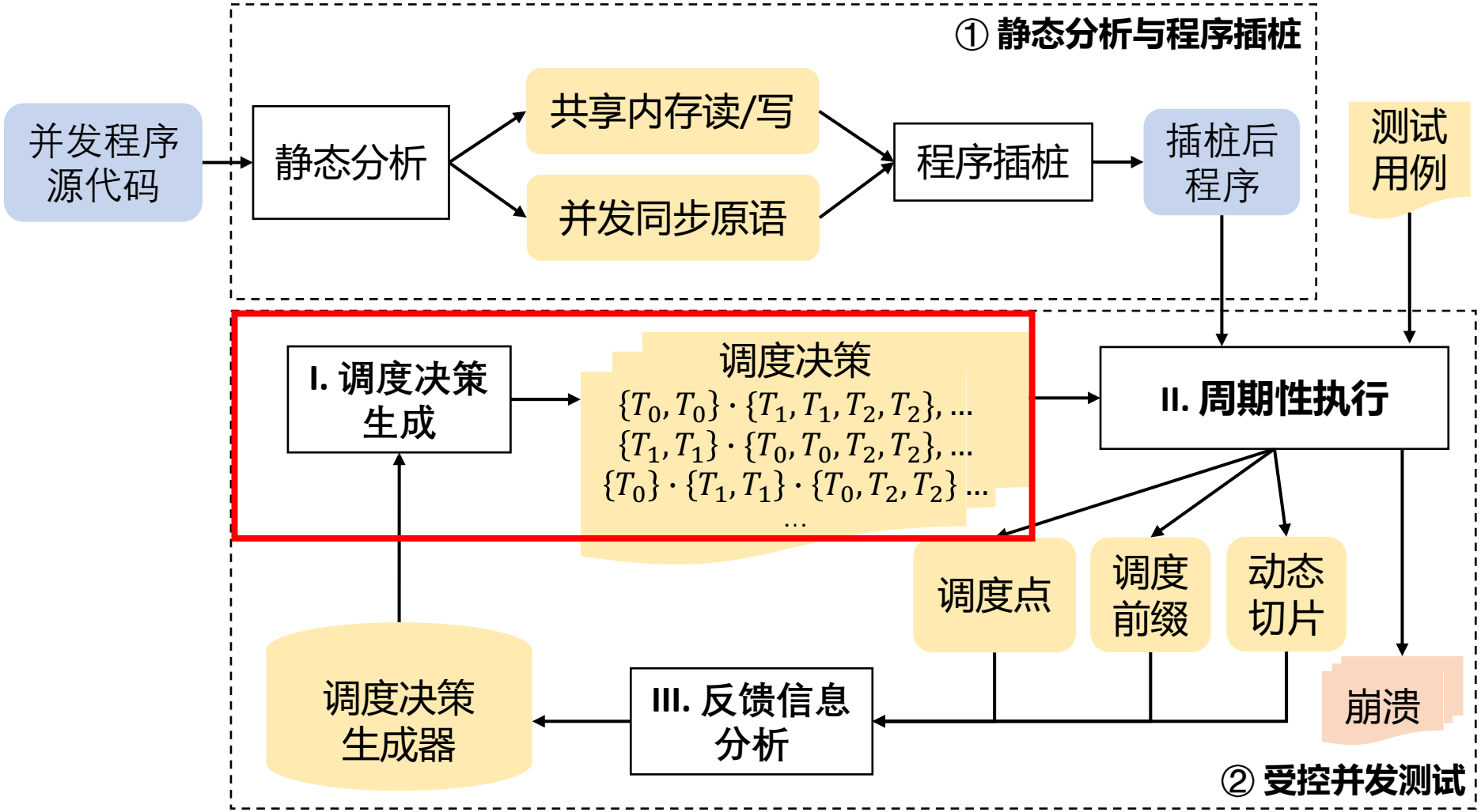


调度决策 $\{T_1\} \cdot \{T_0\} \cdot \{T_1\}$ 的执行过程



利用Linux操作系统提供的deadline任务调度实现主动调度，需要保证参数 runtime < deadline = period

PERIOD的整体工作流程



调度决策生成

调度决策：以一系列线程标识符描述不同执行阶段哪个线程的调度点可执行（例如 $\{T_1\}\cdot\{T_0\}$）

生成调度决策的方式： 设定（并逐步增加）生成的调度决策中所允许周期数

调度决策生成

并发空指针解引用示例：

```
Thread T0:          Thread T1:
lock(p);            /* missing lock(p);   */
p = NULL;  <------  if(p != NULL)
unlock(p);  ------>   x = p->content;
                    /* missing unlock(p); */
```

$\{T_0\times 3\}\cdot\{T_1\times 2\}$

$\{T_1\times 2\}\cdot\{T_0\times 3\}$

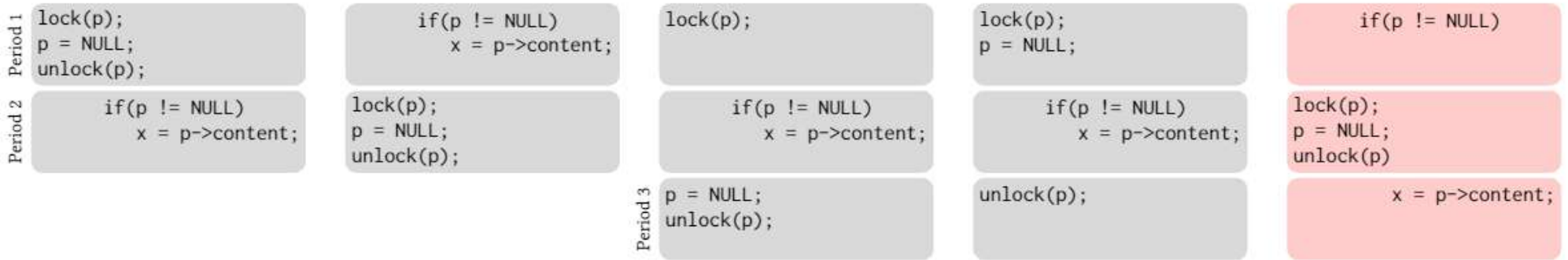
$\{T_0\}\cdot\{T_1\times 2\}\cdot\{T_0\times 2\}$

$\{T_0\times 2\}\cdot\{T_1\times 2\}\cdot\{T_0\}$

$\{T_1\}\cdot\{T_0\times 3\}\cdot\{T_1\}$

……

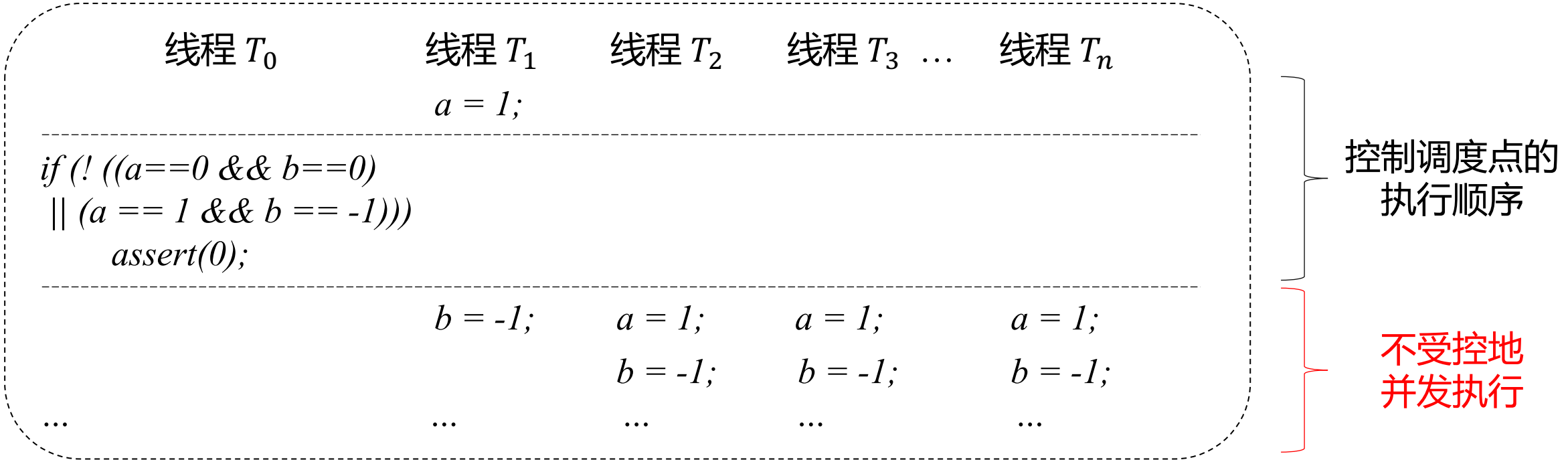
周期性执行



并发调度决策生成策略（从序列化执行到部分并发执行）

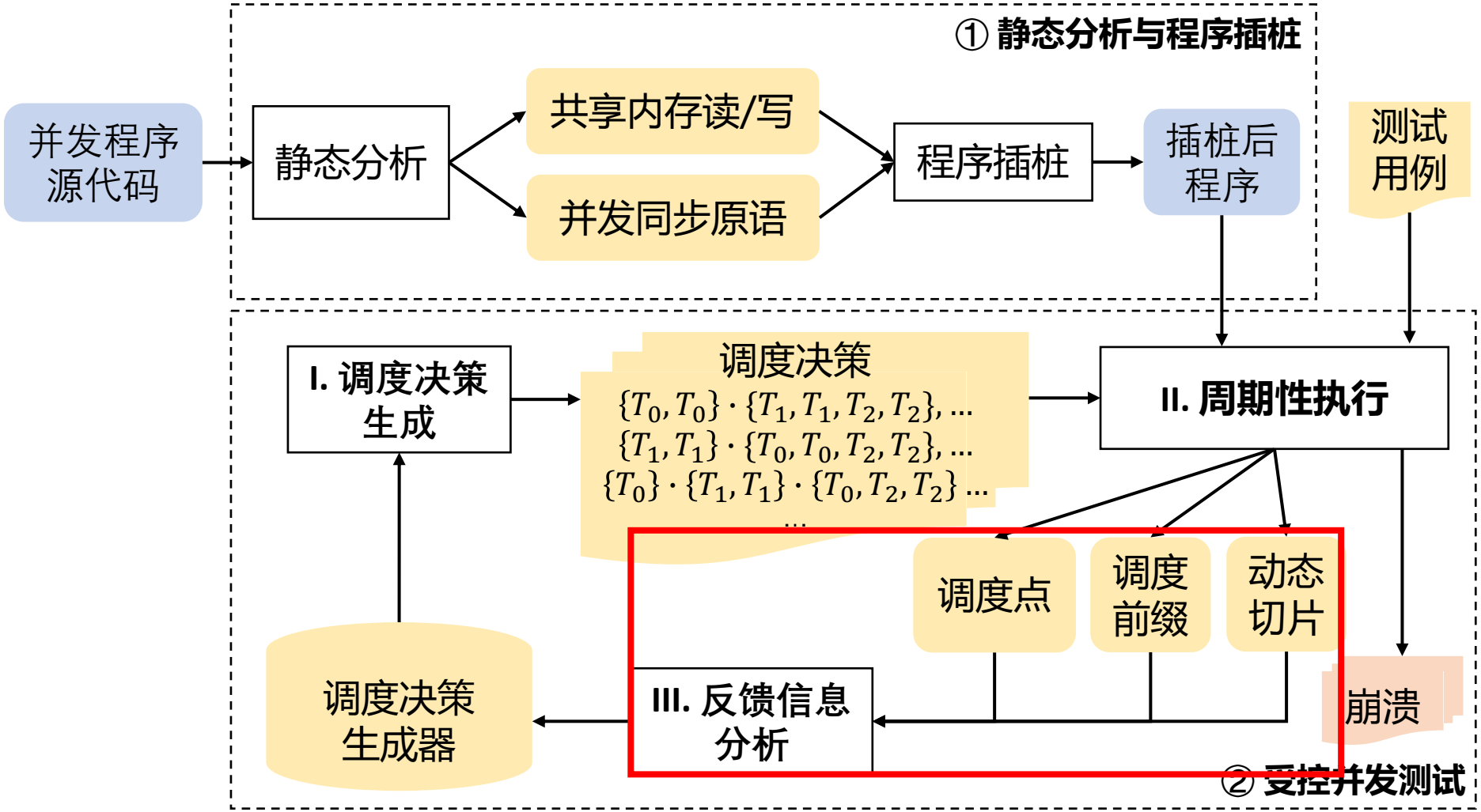
针对线程数量较多的情况进行优化。

核心思想： 在控制一部分调度点的执行交错的情况下，允许剩余的调度点不受控地并发执行



效果：不仅提升了测试执行的速度，还极大程度地减小了需要探索的调度空间

PERIOD的整体工作流程



反馈信息分析

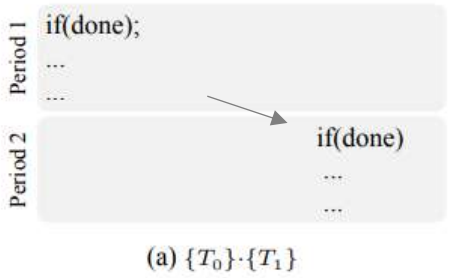
动态调度点切片：某次动态执行所覆盖的各线程的调度点的切片（简称切片）

- 反映的是每个线程执行到的一条具体路径
- 基于切片生成的调度决策更精确，避免不必要的搜索

调度前缀： 调度决策的一段起始部分

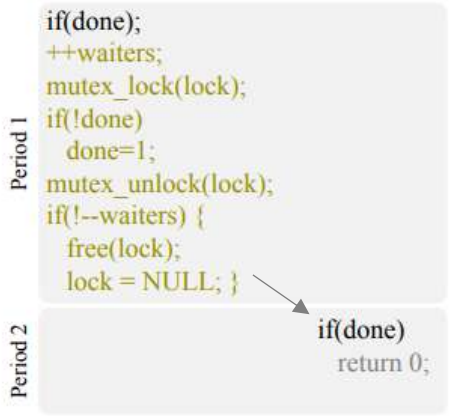
- 反映的是了到达某个切片的必要条件
- 基于切片生成满足特定调度前缀的调度决策，避免不必要的搜索

调度决策



(a) $\{T_0\}\cdot\{T_1\}$

实际执行结果



动态切片（调度前缀为$\{T_0\}$）

Thread T0	Thread T1

if(done)
++waiters;
mutex_lock(lock)
if(!done)
done = 1;
mutex_unlock(lock)
if(! --waiters) {
free(lock);
lock = NULL; }
	if(done)

运行示例： PERIOD对CVE-2016-1972的检测流程（一）

通过轻量级的静态分析技术识别程序中的共享内存读/写访问和并发同步原语

- Once函数中识别出9个调度点（业务流代码已被忽略）

```
1  static pthread_mutex_t* lock;
2  static long waiters = 0;
3  static int done = 0;
4
5  void *once(void *) {
6      if(done)
7          return 0;
8      ++waiters;
9      pthread_mutex_lock(lock);
10     // do some thing ...
11     if(!done)
12         done = 1;
13     pthread_mutex_unlock(lock);
14     if(!--waiters) {
15         free(lock);
16         lock = NULL;
17     }
18 }
19 void main() {
20     lock = malloc(sizeof(pthread_mutex_t));
21     pthread_t T0, T1;
22     pthread_create(&T0, NULL, once, NULL);
23     pthread_create(&T1, NULL, once, NULL);
24     pthread_join(T1, NULL);
25     pthread_join(T0, NULL);
26 }
```

轻量级静态分析

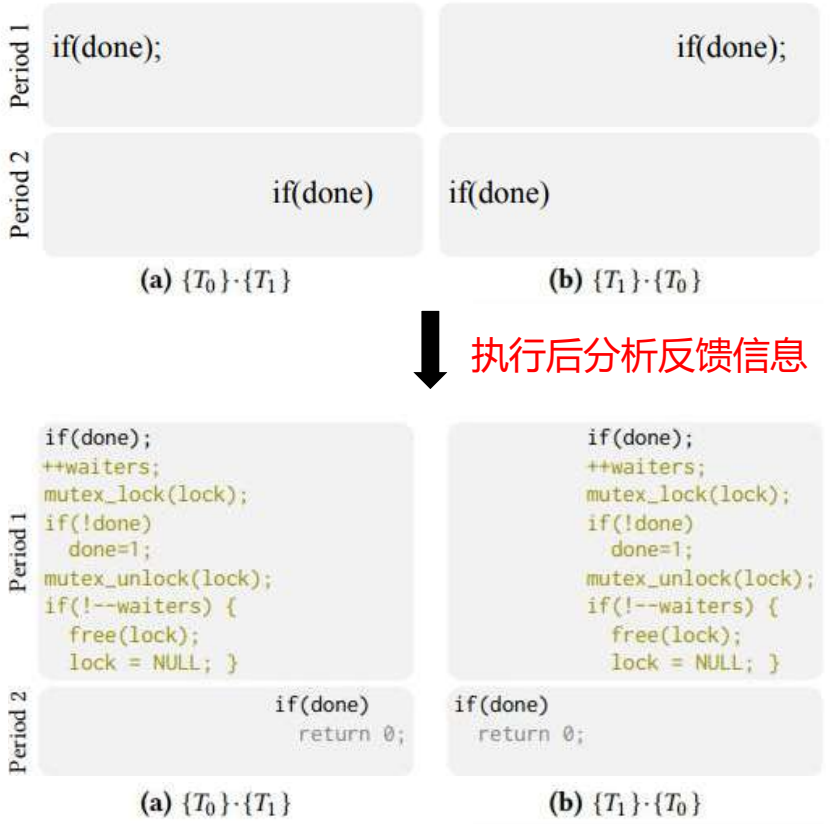
① if(done)
② ++waiters;
③ pthread_mutex_lock(lock)
④ if(!done)
⑤ done = 1;
⑥ thread_mutex_unlock(lock)
⑦ if(! --waiters) {
⑧ free(lock);
⑨ lock = NULL; }

运行示例： PERIOD对CVE-2016-1972的检测流程（二）

仅控制第一个调度点，生成两个周期内的调度决策

动态执行程序后分析反馈信息

```
1  static pthread_mutex_t* lock;
2  static long waiters = 0;
3  static int done = 0;
4
5  void *once(void *) {
6      if(done)
7          return 0;
8      ++waiters;
9      pthread_mutex_lock(lock);
10     // do some thing ...
11     if(!done)
12         done = 1;
13     pthread_mutex_unlock(lock);
14     if(!--waiters) {
15         free(lock);
16         lock = NULL;
17     }
18 }
19 void main() {
20     lock = malloc(sizeof(pthread_mutex_t));
21     pthread_t T0, T1;
22     pthread_create(&T0, NULL, once, NULL);
23     pthread_create(&T1, NULL, once, NULL);
24     pthread_join(T1, NULL);
25     pthread_join(T0, NULL);
26 }
```



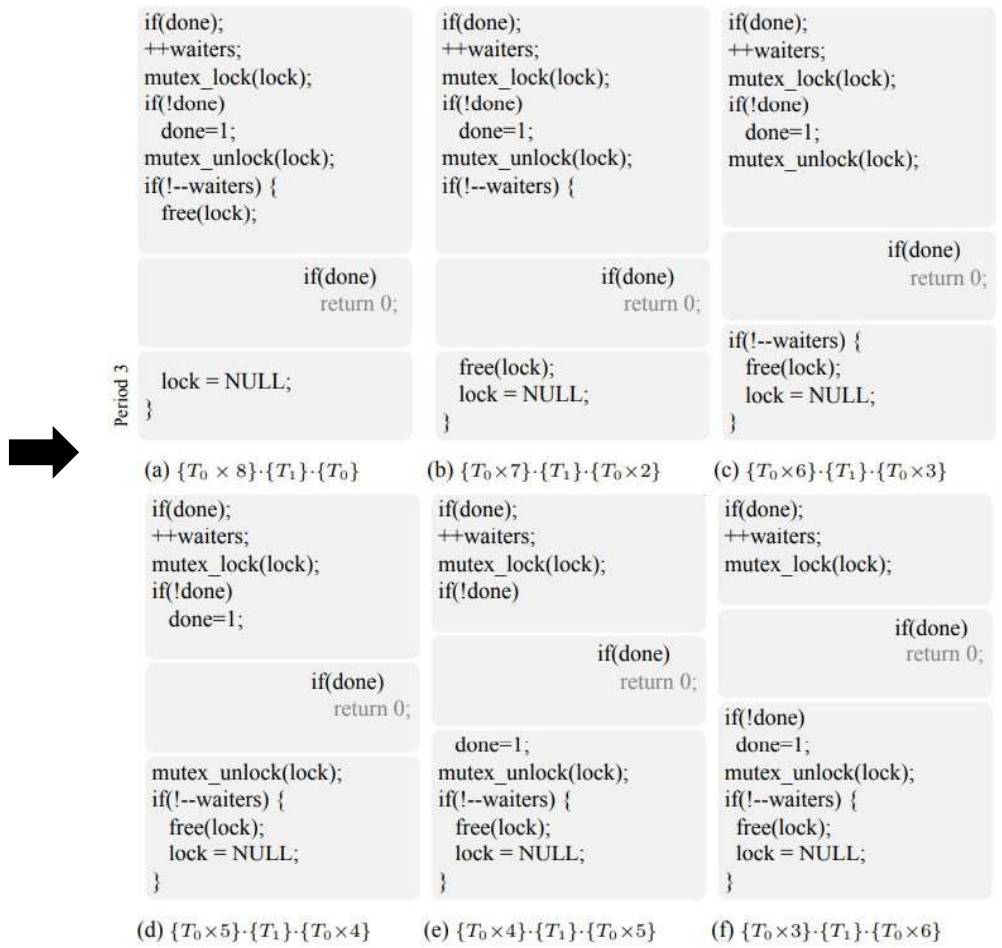
运行示例： PERIOD对CVE-2016-1972的检测流程（三）

基于新发现的调度点切片，继续生成可能的调度决策（最大允许三个周期）

动态执行程序后分析反馈信息

```
1  static pthread_mutex_t* lock;
2  static long waiters = 0;
3  static int done = 0;
4
5  void *once(void *) {
6      if(done)
7          return 0;
8      ++waiters;
9      pthread_mutex_lock(lock);
10     // do some thing ...
11     if(!done)
12         done = 1;
13     pthread_mutex_unlock(lock);
14     if(!--waiters) {
15         free(lock);
16         lock = NULL;
17     }
18 }
19 void main() {
20     lock = malloc(sizeof(pthread_mutex_t));
21     pthread_t T0, T1;
22     pthread_create(&T0, NULL, once, NULL);
23     pthread_create(&T1, NULL, once, NULL);
24     pthread_join(T1, NULL);
25     pthread_join(T0, NULL);
26 }
```

```
if(done);
++waiters;
mutex_lock(lock);
if(!done)
  done=1;
mutex_unlock(lock);
if(!--waiters) {
  free(lock);
  lock = NULL; }
                    if(done)
                      return 0;
```



运行示例： PERIOD对CVE-2016-1972的检测流程（四）

动态执行5个调度决策后，发现了新的动态调度点切片

分析调度前缀，并在后续的探索中使用调度前缀指导调度决策的生成

```
1  static pthread_mutex_t* lock;
2  static long waiters = 0;
3  static int done = 0;
4
5  void *once(void *) {
6      if(done)
7          return 0;
8      ++waiters;
9      pthread_mutex_lock(lock);
10     // do some thing ...
11     if(!done)
12         done = 1;
13     pthread_mutex_unlock(lock);
14     if(!--waiters) {
15         free(lock);
16         lock = NULL;
17     }
18 }
19 void main() {
20     lock = malloc(sizeof(pthread_mutex_t));
21     pthread_t T0, T1;
22     pthread_create(&T0, NULL, once, NULL);
23     pthread_create(&T1, NULL, once, NULL);
24     pthread_join(T1, NULL);
25     pthread_join(T0, NULL);
26 }
```

```
if(done);
++waiters;
mutex_lock(lock);
if(!done)
```

```
                    if(done)
                      return 0;
```

```
  done=1;
mutex_unlock(lock);
if(!--waiters) {
  free(lock);
  lock = NULL;
}
```

执行后分析反馈信息

```
if(done);
++waiters;
mutex_lock(lock);
if(!done)
```

```
            if(done)
            ++waiters;
```

```
  done=1;
mutex_unlock(lock);
if(!--waiters)
            mutex_lock(lock);
            if(!done)
            mutex_unlock(lock);
            if(!--waiters) {
              free(lock);
              lock = NULL;
            }
```

调度前缀: $\{T_1 \times 4\} \cdot \{T_0\} \cdot \ldots$

```
if(done)
++waiters;
pthread_mutex_lock(lock)
if(!done)

  done = 1;
…
…
…
```

```
if(done)
…
…
…
```

运行示例： PERIOD对CVE-2016-1972的检测流程（五）

通过调度前缀生成能到达特定动态调度点切片的调度，避免不必要的探索。

当迭代的周期数增加到6时，共发现NPD、UAF和DF三种内存并发漏洞。

实验评估

PERIOD的工具原型基于LLVM平台、SVF工具和SCHED_DEADLINE构建

- https://github.com/wcventure/PERIOD

实验数据集：

- CVE benchmark (10 CVEs): 释放后使用、双重释放、空指针解引用漏洞
- SCTBench (36 programs): 竞态漏洞、死锁

对比的技术：

- PERIOD的序列化版本（无并发调度决策生成方案）： SERIAL
- 系统性的受控并发测试技术： IPB, IDB, DFS
- 非系统性的受控并发测试技术： PCT, Random, Maple, Native
- 并发漏洞动态检测工具： ConVul, UFO
- 数据竞争检测工具： FastTrack, Helgrind, ThreadSanitizer (Tsan)

CVE Benchmark上的实验结果

				Systematic Testing												Non-systematic Testing								Con. Bug Detector		Data Race Detector		
				PERIOD / SERIAL			IPB			IDB			DFS			Native		PCT		Random		Maple							
Bug ID	Programs	Bug Type	Bug Depth	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	runs to 1st bug	buggy runs	schedules to bug (1st)	buggy schedules	schedules to bug (1st)	buggy schedules	founds?	schedules	ConVul	UFO / UFO$_{NPD}$	FastTrack	Helgrind	TSAN
CVE-2009-3547	Linux-2.6.32	NPD	1	2	6	3	3	36	5	5	33	4	4	10	1	249	3	5	3333	8	2506	✓	30	✓	✓	✓	✓	✓
CVE-2011-2183	Linux-2.6.39	NPD	2	3	906	130	8	98	11	6	85	9	5	31	3	681	10	5	394	8	3745	✓	60	✗	✗	✗	✗	✗
CVE-2013-1792	Linux-2.8.3	NPD	2	13	179	6	15	321	18	22	260	14	15	88	5	✗	0	61	124	8	741	✓	165	✓	✗	✗	✗	✗
CVE-2015-7550	Linux-4.3.4	NPD	2	3	14	6	8	73	11	6	64	9	5	22	3	✗	0	3	394	1	3745	✓	160	✓	✓	✗	✗	✗
CVE-2016-1972	Firefox-45.0	NPD	2	3	573	20	16	L	881	11	6176	472	228	L	90	✗	0	3	731	55	430	✓	144	✗	✗	✗	✗	✗
		UAF	4	159		11	91		918	66		663	229		337	✗	0	134	15	1	1539	✗		✓	✗	✗	✗	✗
		DF	5	447		1	✗		0	✗		0	✗		0	✗	0	✗	0	✗	0	✗		✗	✗	✗	✗	✗
CVE-2016-1973	Firefox-45.0	NPD	2	5	31	2	✗	L	0	✗	L	0	✗	L	0	1415	3	✗	0	✗	0	✓	157	✗	✗	✗	✗	✗
		UAF	3	17		5	✗		0	✗		0	✗		0	✗	0	✗	0	✗	0	✗		✓	✗	✗	✗	✗
CVE-2016-7911	Linux-4.6.6	NPD	2	3	19	8	8	204	66	6	170	54	5	58	21	799	15	5	511	5	3733	✓	143	✓	✗	✗	✗	✗
CVE-2016-9806	Linux-4.6.3	DF	2	6	42	4	9	226	84	7	193	65	6	71	28	✗	0	3	1135	1	2353	✓	36	✓	-	✗	✗	✗
CVE-2017-15265	Linux-4.9.13	UAF	2	11	96	1	✗	88	0	✗	83	0	✗	31	0	✗	0	✗	0	✗	0	✓	73	✓	✓	✗	✗	✓
CVE-2017-6346	Linux-4.13.8	NPD	2	5	182	60	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✓	118	✗	✗	✗	✗	✗
		UAF	3	47		6	✗		0	✗		0	✗		0	✗	0	✗	0	✗	0	✗		✓	✗	✗	✗	✗	✗
		DF	2	46		14	✗		0	✗		0	✗		0	✗	0	✗	0	20	1625	✓		✓	-	✗	✗	✗
Total bugs found (Buggy Programs)				15 (10)			8 (7)			8 (7)			8 (7)			4 (3)		8 (7)		9 (7)		11 (7)		10 (9)		3 (3)	1 (1)	1 (1)	2 (2)

我们的工具

系统性的受控并发测试技术

基于随机调度的受控并发测试

启发式方法

[FSE' 20] [ICSE'18]

数据竞争检测工具

SCTBench上的实验结果（第一部分）

Programs	Sub Threads	Bug Type	Bug Depth	Systematic Testing															Non-systematic Testing							
				PERIOD			SERIAL			IPB			IDB			DFS			Native		PCT		Random		Maple	
				schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	runs to 1st bug	buggy runs	schedules to bug (1st)	buggy schedules	schedules to bug (1st)	buggy schedules	founds?	schedules
CS.account_bad	3	AF	1	2	65	26	4	136	56	3	70	13	4	43	6	3	28	4	2903	1	5	2396	8	1177	✓	80
CS.bluetooth_driver_bad	2	AF	2	9	205	9	9	205	9	7	92	9	7	92	9	36	177	10	✗	0	700	85	8	648	✗	57
CS.carter01_bad	2	DL	2	5	42	11	5	42	11	11	396	38	9	250	18	8	1708	49	555	1	3	608	1	4750	✓	6
CS.circular_buffer_bad	2	AF	2	17	871	207	17	871	207	26	806	363	42	623	219	20	3991	2043	✗	0	11	842	1	9110	✗	58
CS.deadlock01_bad	2	DL	2	3	14	6	3	14	6	11	81	8	8	65	6	10	46	3	4353	2	38	174	1	3745	✗	89
CS.lazy01_bad	3	AF	1	3	39	12	4	60	20	1	208	13	1	87	62	1	118	81	2	6631	1	5128	2	6092	✓	1
CS.queue_bad	2	AF	2	25	998	119	25	998	119	101	L	7275	106	8310	3768	43	L	6405	7993	1	6	984	1	L	✓	64
CS.reorder_10_bad	10	AF	2	27	2350	89	✗	L	0	✗	L	0	✗	7406	0	✗	L	0	✗	0	85	9	✗	0	✗	56
CS.reorder_20 bad	20	AF	2	39	L	2870	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	891	18	✗	0	✗	56
CS.reorder_3_bad	3	AF	2	6	27	6	10	30	10	50	1192	25	33	205	6	126	2494	23	✗	0	192	54	39	237	✗	56
CS.reorder_4_bad	4	AF	2	9	100	12	37	384	90	393	L	31	262	518	7	6409	L	4	✗	0	164	40	68	86	✗	56
CS.reorder_5_bad	5	AF	2	12	225	20	283	5040	816	3587	L	3	✗	996	0	✗	L	0	✗	0	355	28	68	23	✗	56
CS.stack_bad	2	AF	2	3	918	144	3	918	144	25	2429	318	23	1595	273	22	L	512	✗	0	15	6	3	6189	✓	2
CS.token_ring_bad	4	AF	1	2	232	58	7	2424	401	8	503	114	15	113	13	8	280	57	✗	0	11	6	9	1293	✓	45
CS.twostage_100_bad	100	AF	2	690	L	141	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	13453	11	✗	0	✗	56
CS.twostage_bad	2	AF	2	4	33	3	4	33	3	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✓	8
CS.wronglock_3_bad	4	AF	2	6	172	42	26	1320	265	277	L	1227	16	1568	197	3233	L	94	7212	1	7	313	1	3197	✓	19
CS.wronglock_bad	8	AF	2	10	1464	210	✗	L	0	✗	L	0	32	L	710	✗	L	0	✗	0	44	307	1	3286	✓	19
CB.aget-bug	3	AF	2	9	3279	679	12	L	1900	1	4359	2903	1	292	194	1	2847	1814	2	3341	7	3202	4	4853	✓	1
CB.stringbuffer	2	AF	2	12	163	16	12	163	16	13	38	2	13	38	2	8	30	2	✗	0	3555	5	23	673	✓	40

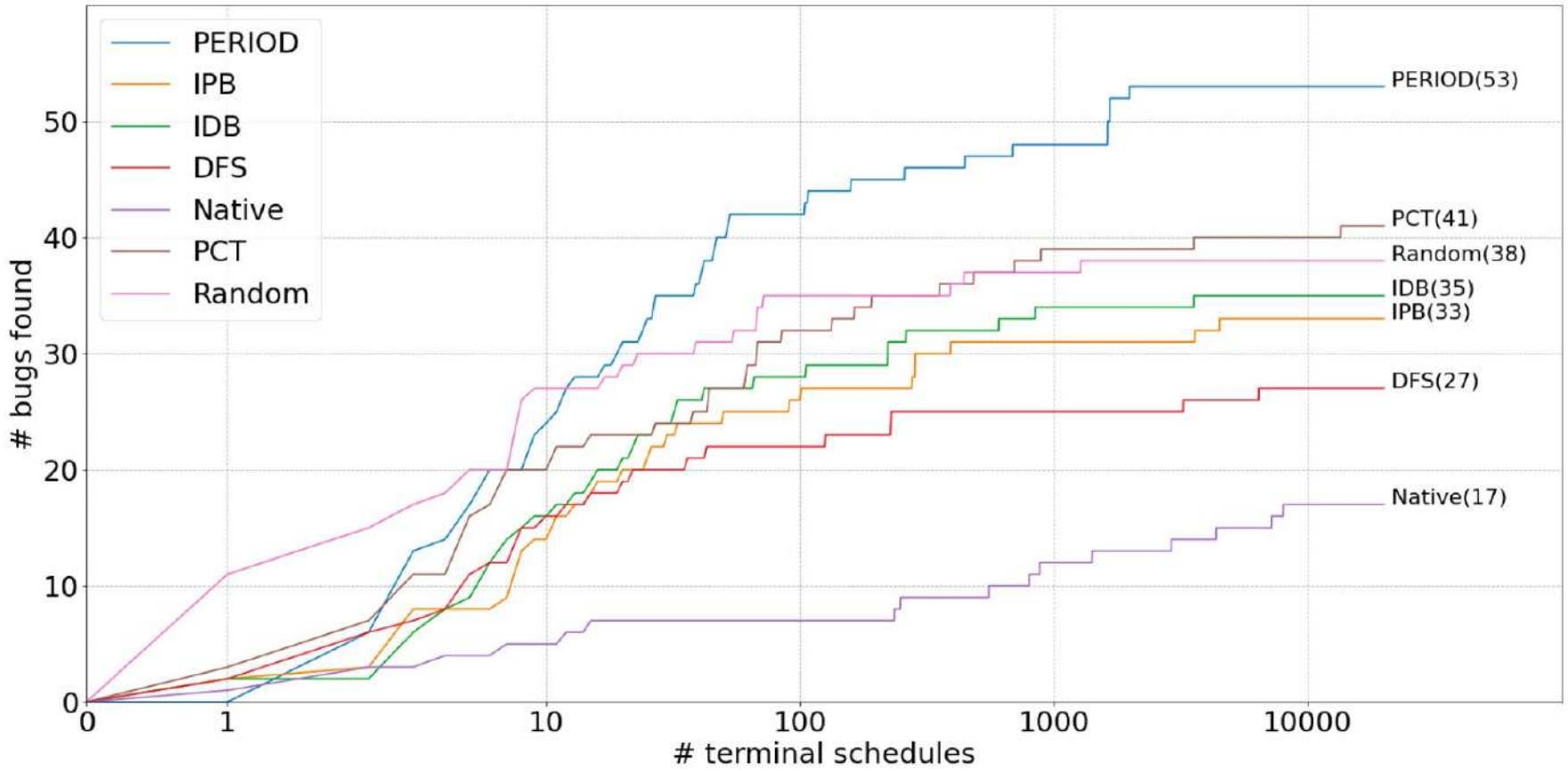
PERIOD相比其序列化版本（SERIAL）用更少的调度决策发现同一个漏洞；在有限的时间内发现更多的漏洞

SCTBench上的实验结果（第二部分）

Programs	Sub Threads	Bug Type	Bug Depth	Systematic Testing: PERIOD schedules to bug (1st)	PERIOD schedules	PERIOD buggy schedules	SERIAL schedules to bug (1st)	SERIAL schedules	SERIAL buggy schedules	IPB schedules to bug (1st)	IPB schedules	IPB buggy schedules	IDB schedules to bug (1st)	IDB schedules	IDB buggy schedules	DFS schedules to bug (1st)	DFS schedules	DFS buggy schedules	Non-systematic Testing: Native runs to 1st bug	Native buggy runs	PCT schedules to bug (1st)	PCT buggy schedules	Random schedules to bug (1st)	Random buggy schedules	Maple founds?	Maple schedules
CB.pbzip2	4	BOF	2	41	1626	14	578	L	27	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	42
		NPD	2	52		41	82		427	16		36	4		1733	12		573	✗	0	62	136	1	2545	✓	
		UAF	1	53		24	83		498	2		4902	3		3053	2		1267	✗	0	7	2193	5	1594	✗	
Chess.WSQ	3	AF	3	105	434	15	574	5175	117	4502	L	306	845	L	270	✗	L	0	✗	0	2	1118	392	10	✓	49
Chess.IWSQ	3	AF	3	108	711	17	836	L	138	✗	L	0	3554	L	192	✗	1503	0	✗	0	5	1173	443	24	✗	10
Chess.SWSQ	3	BOF	3	1628	L	11	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	60
		AF	3	1630		13	✗		0	284		1	222		1	✗		0	✗	0	68	257	17	1078	✗	
Chess.IWSQWS	3	BOF	3	1661	L	11	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	70
		AF	3	1663		13	✗		0	284		1	222		1	✗		0	✗	0	68	261	3	1918	✗	
Inspect.qsort_mt	3	AF	2	27	8643	135	30	L	96	33	L	365	20	3882	158	✗	L	0	✗	0	44	194	72	100	✗	115
Inspect.boundedbuffer	4	AF	2	20	L	1514	39	L	1382	✗	L	0	608	L	103	✗	L	0	15	294	27	109	8	2808	✗	158
Misc.safestack	3	AF	-	✗	6519	0	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	59
Splash2.barnes	2	AF	1	2	L	1186	2	L	1186	3	L	1006	3	L	741	2	L	2202	7	1251	2	5013	2	4893	✓	1
Splash2.fft	2	AF	1	2	L	3963	2	L	3963	3	L	9221	3	L	9221	2	L	7210	1	6862	2	5047	2	6241	✓	2
Splash2.lu	2	AF	1	2	6129	2848	2	6129	2848	3	L	6900	3	L	6900	2	L	5560	12	2177	2	5724	2	9714	✓	4
RADBench.bug2	2	AF	3	1985	L	9	1985	L	9	✗	L	0	✗	L	0	✗	L	0	✗	0	1813	10	✗	0	✗	264
RADBench.bug3	2	AF	2	42	L	3478	42	L	3478	✗	L	0	✗	L	0	✗	L	0	✗	0	1	489	✗	0	✗	227
RADBench.bug4	2	AF	3	259	L	6	259	L	6	✗	L	0	✗	L	0	✗	L	0	4	2013	✗	0	1275	13	✓	1
RADBench.bug5	2	DL	2	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	224
RADBench.bug6	2	DL	2	24	2950	340	24	2950	340	30	8327	112	27	4039	70	✗	L	0	236	7	484	34	3	855	✓	14
Total bugs found (Buggy Programs)				38 (34)			30 (28)			25 (24)			27 (26)			19 (18)			13 (13)		33 (32)		29 (28)		18 (18)	

PERIOD和其序列化版本（SERIAL）都比其它方法发现了更多的并发漏洞

发现的漏洞数量随调度决策数量的增长趋势



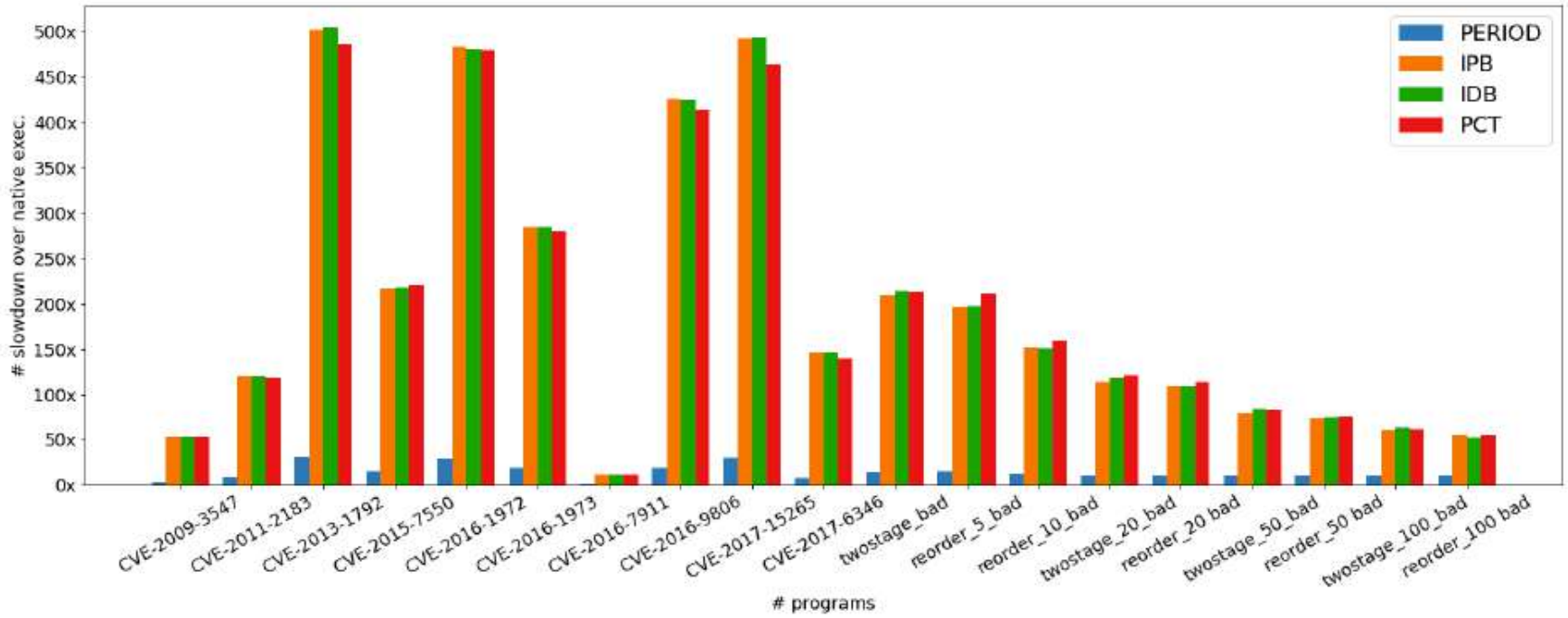
相比其它受控并发测试技术

- PERIOD可以用同样数量的调度决策发现更多的并发漏洞
- PERIOD发现相同数量的并发漏洞所需要的调度数量更少

运行时开销评估

PERIOD引入运行时间开销相比IPB、IDB、PCT更低，其原因主要在两个方面：

- PERIOD的基于周期性执行的调度方案没有引入额外的延时或抢占式的中断
- PERIOD的并发调度决策生成策略在控制一部分调度点的线程交错的情况下，运行剩余的调度点不受控地并发执行，而IPB、IDB、PCT均要求序列化执行程序。



执行速度（单位时间内执行的调度决策数）对比

阶段性研究成果（三）

- 撰写的论文《Controlled Concurrency Testing via Periodical Scheduling》已在软件工程领域的顶会ICSE '22上发表。
- PERIOD的工具原型和相关实验数据集已通过ICSE’22的“Artifacts evaluated available and reusable”的认证。

- 新发现并报告了5个并发漏洞（1 CVE）
- 在华为公司产品线落地应用。成功复现一个现网问题，发现该并发错误的效率相比压力测试提升60x以上；2022年6月收到产品线的感谢信；现已成功应用到两个模块的单元测试和集成测试阶段。

表 5-5　PERIOD 新发现的内存并发漏洞

漏洞 ID	程序	漏洞类型
CVE-2022-26291	*Lrzip v0.641*	（并发）释放后使用
#issue-1	*Pbzip2 v1.0.5*	（并发）缓冲区溢出
#issue-2	*CTrace v1.2*	（并发）非法内存解引用
#issue-3	*CTrace v1.2*	竞态漏洞
#issue-4	*Axel v2.17.10*	竞态漏洞

图：2022年6月收到华为公司产品线的感谢信

目　录

CONTENTS

深圳大学
SHENZHEN UNIVERSITY

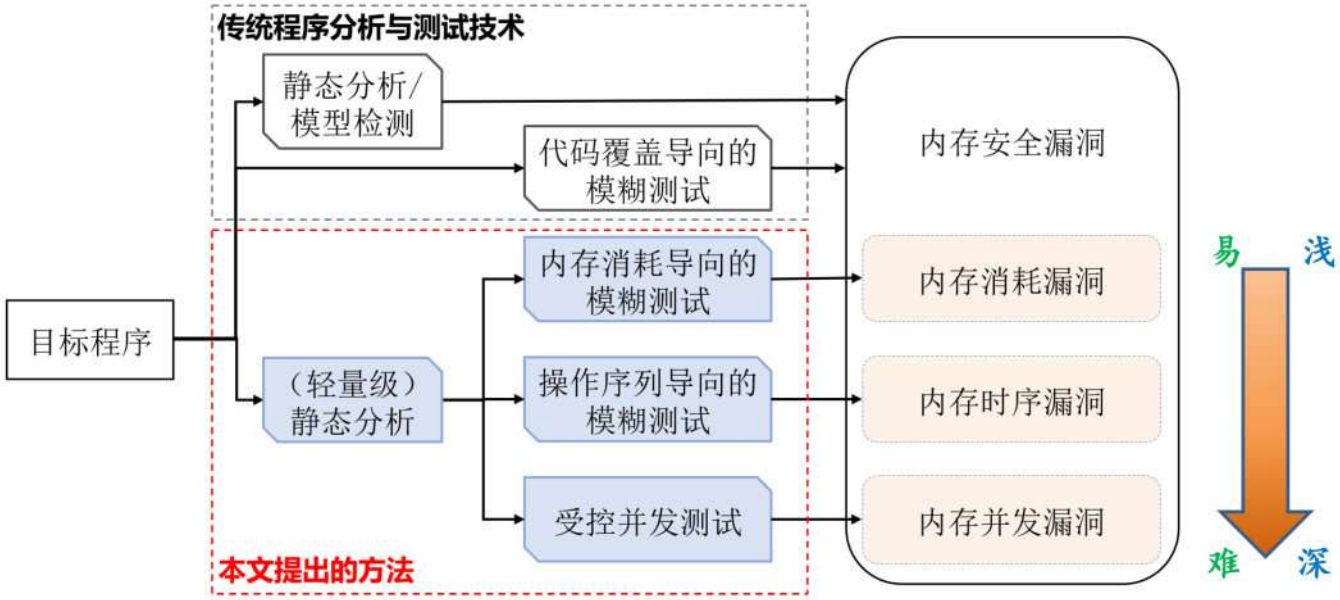
总结与展望

本文工作总结

- 一个通用可扩展的内存安全漏洞检测框架：
  - 内存消耗漏洞检测：首次提出增加内存消耗这个新维度来指导模糊测试，使得测试过程中能自动化地逐步生成能造成过量内存消耗的测试用例，弥补了当前模糊测试技术在检测内存消耗漏洞方面的盲区。
  - 内存时序漏洞检测：首次提出将违反内存使用的安全时序规则的操作序列用于引导模糊测试，提升了模糊测试在发现内存时序漏洞方面的有效性和效率。
  - 内存并发漏洞检测：创新性地提出了一种系统性的并发程序测试方法，使用一种基于周期性执行的调度方案来主动控制线程调度，能根据历史执行信息循序渐进地高效探索调度空间，极大程度地提升了发现并发漏洞的效率。
- 工具集： MemLock、UAFL、PERIOD
- 36个安全攸关的软件漏洞（36 CVEs）

未来工作展望

- 更加通用准确的内存安全漏洞检测
- 弱内存模型下的内存安全漏洞检测
- 面向分布式系统的并发漏洞检测



已发表的学术论文（博士期间）

[1] **Cheng Wen**, Mengda He, Bohao Wu, et al.. Controlled Concurrency Testing via Periodical Scheduling. IEEE/ACM 44th International Conference on Software Engineering (ICSE). 2022. (第一作者, CCF A)

[2] **Cheng Wen**, Haijun Wang, Yuekang Li, Shengchao Qin, et al. MemLock: Memory Usage Guided Fuzzing. IEEE/ACM 42nd International Conference on Software Engineering (ICSE). 2020. (第一作者, CCF A)

[3] Haijun Wang, Xiaofei Xie, Yi Li, **Cheng Wen**, et al. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. IEEE/ACM 42nd International Conference on Software Engineering (ICSE). 2020. (CCF A)

[4] Zhiwu Xu, **Cheng Wen**, Shengchao Qin and Mengda He. Extracting automata from neural networks using active learning. PeerJ Computer Science, April 2021. (JCR二区)

[5] Zhiwu Xu, **Cheng Wen**, Shengchao Qin. Type Learning for Binaries and its Applications. Accepted by The IEEE Transactions on Reliability, 2019. (中科院二区)

已发表的学术论文（硕士期间）

[6] Zhiwu Xu, **Cheng Wen**, Shengchao Qin. State-taint analysis for detecting resource bugs. Science of Computer Programming, 93-109, 2018. (SCI/ESI, CCF B)

[7] Zhiwu Xu, Xiongya Hu, **Cheng Wen** and Shengchao Qin. Extracting Automata from Neural Networks Using Active Learning. 3rd National Conference on Formal Methods and Applications, 2018. (Best Paper Award)

[8] Zhiwu Xu, **Cheng Wen**, Shengchao Qin and Zhong Ming. Effective Malware Detection based on Behaviour and Data Features. SmartCom, 53-66, 2017. (Best Student Paper Award)

[9] Zhiwu Xu, **Cheng Wen**, Shengchao Qin. Learning types for binaries. 19th International Conference on Formal Engineering Methods, 430-446, 2017. (CCF C)

谢谢各位老师！