软件安全缺陷分析初探

报告人：文 成

时间：2023年02月07日

目　录

CONTENTS

数字经济下软件无处不在

- 安全攸关系统 ：航空航天、轨道交通、汽车电子、医疗设备等
- 关键基础设施：通讯网络、电力水利系统、智慧城市、云服务…
- 其他ICT常见系统：IoT设备软件、手机等移动设备、智能家居家电、机器人、无人机等
- 其他系统：政府/企业（包括银行）软件系统等

系统越来越复杂多样，安全性可靠性要求越来越高

软件缺陷/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

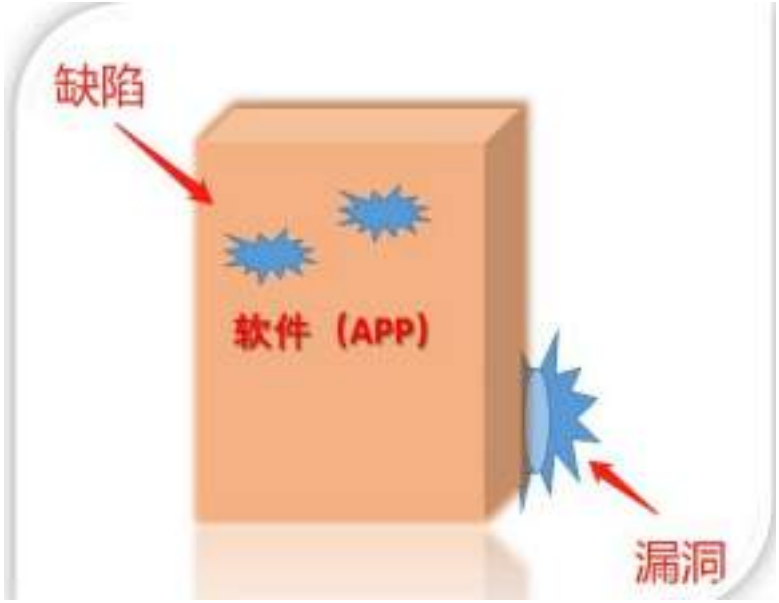
软件数据竞争问题导致美国东北部大面积停电（2003）

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

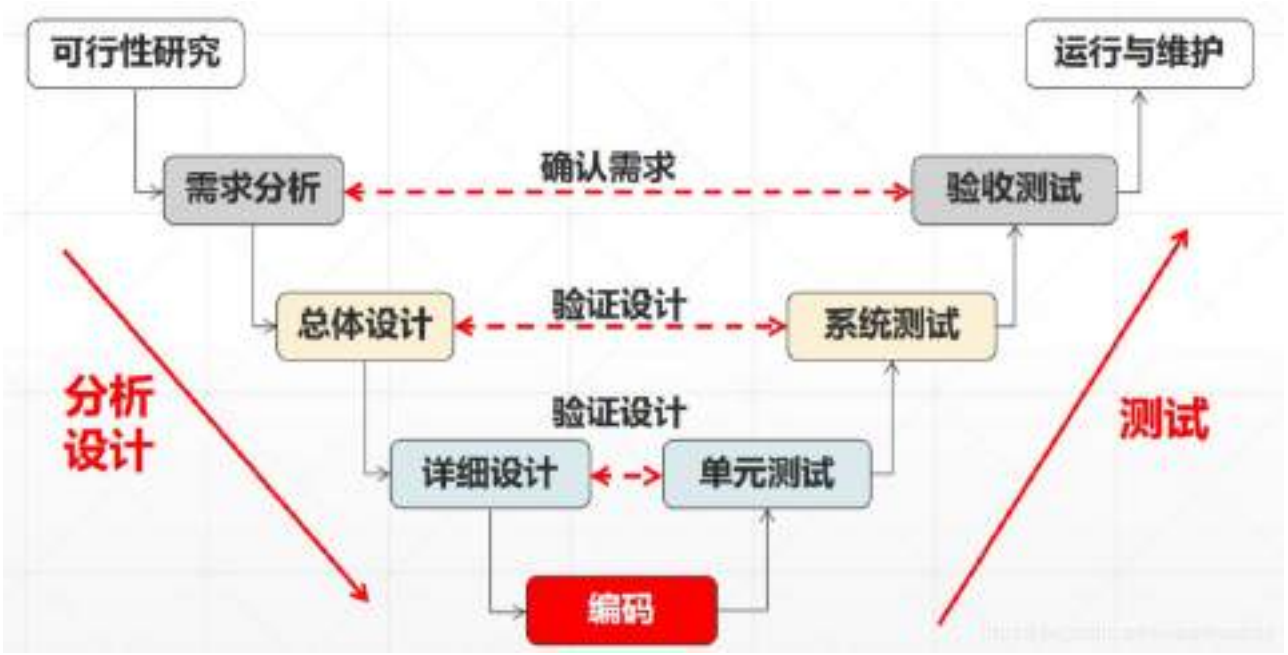
什么是软件缺陷/漏洞？

- 所谓软件缺陷（常常又被叫做Bug），即为计算机软件或程序中存在的某种破坏正常运行能力的问题、错误，或者隐藏的功能缺陷。 缺陷的存在会导致软件产品在某种程度上不能满足用户的需要。
- 软件漏洞是指软件在设计、实现、配置策略及使用过程中出现的缺陷，它可能导致攻击者在未授权的情况下访问或破坏系统。

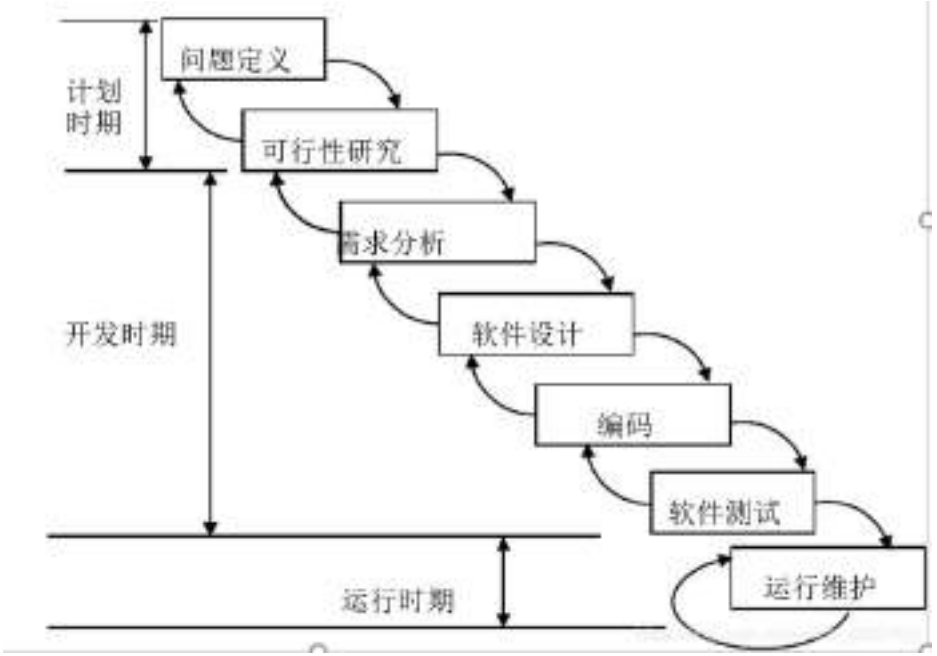


为什么会存在软件缺陷/漏洞？

- “你不可能写出完美的软件”——《程序员修炼之道》
- 软件项目开发是智力密集、劳动密集型的工作，受人力的影响较大
- 软件缺陷/漏洞存在于软件生命周期的各个阶段
  - 需求不明确、软件结构复杂、编码错误、项目期限短等



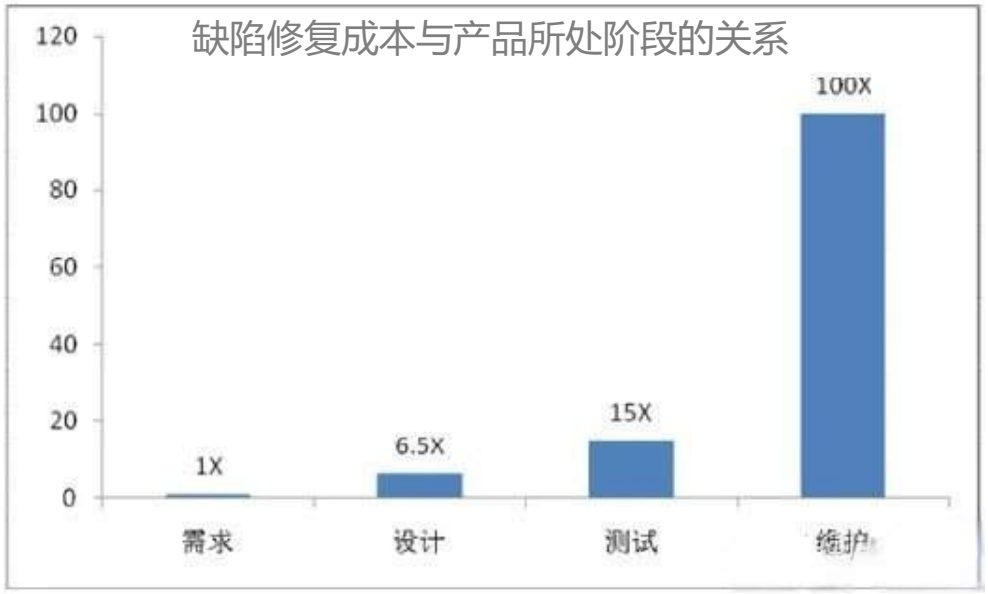
软件生命周期的V模型



软件生命周期的瀑布模型

软件缺陷分析

- 软件缺陷分析就是对计算机软件中的缺陷进行量化表示和定性解释，评估各种类型缺陷发生的概率，明晰缺陷产生的根因，以帮助我们评估和改进软件质量的一种活动及其工程方法
- **目的和愿景**：通过缺陷分析最大限度地发现软件中的缺陷，建立软件质量的量化评估系统，不断改进软件的正确性、安全性和可靠性，促进软件质量的不断提高。
- 软件缺陷越早被发现，解决软件缺陷的成本投入就越少！



软件缺陷的分类

软件缺陷有很多种，从不同的角度可以将软件缺陷分为不同的种类

划分标准					
测试种类	界面类	功能类	性能类	安全性类	兼容性类
严重程度	严重	一般	次要	建议	
优先级	立即解决	高优先级	正常排队	低优先级	
发生阶段	需求阶段	构架阶段	设计阶段	编码阶段	测试阶段

表：按照不同标准划分缺陷类型

安全性类缺陷

- 内存安全缺陷：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
- 线程安全缺陷：多线程环境下因线程交互地作用于相同的共享资源，引起程序崩溃或挂起，或产生与串行执行不同的结果。
- SQL注入缺陷：web应用软件在接收相关数据参数时未做好过滤，将其直接带入到数据库中查询，导致攻击者可以拼接执行构造的SQL语句
- 信息泄露缺陷：软件无意间向用户泄露敏感信息

内存安全缺陷/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

（空间内存问题）

1.数组越界读写

2.非法指针读写

3.栈地址逃逸

4.不受控的内存消耗

（时间内存问题）

5.内存泄漏

6.空指针解引用

7.释放后使用（UAF）

8.双重释放（DF）

...

```
A *g;
void Test() {
    A* l = malloc(...);
    if ( l == NULL) {
        return;
    }

    g = l;

    if (error) {
      free(l);
      l = NULL;
      //g 未置空，导致其他逻辑再次使用
    }
}

void Test2() {
    if (g != NULL) {
        // 对g解引用
    }
}
```

内存释放后使用

```
void Test() {
    // 采用内存管理模块申请空间
    void *p = MemoryManagerMalloc(xxx);
    // 采用free进行释放，还给系统，而非还给内存管理模块
    free(p);
    p = NULL;
}
```

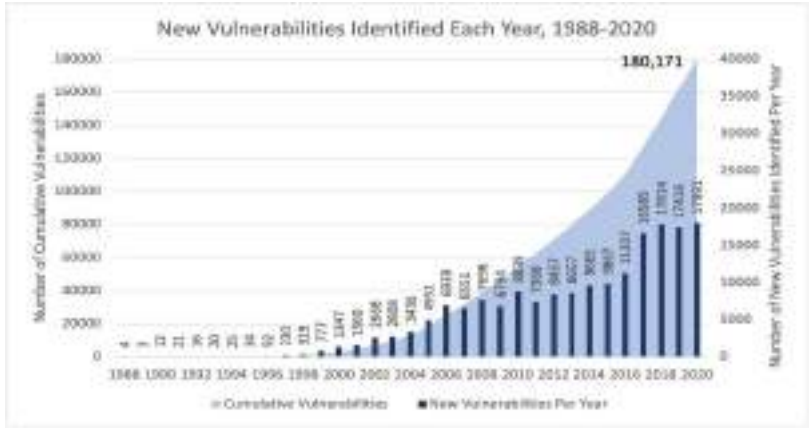
内存申请释放不匹配

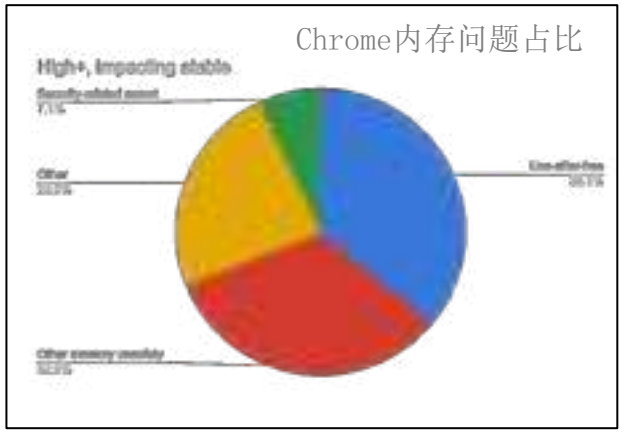
内存泄漏

```
//不合理，误以为这里置空能反应到caller
void CustomFree2(int *p) {
    if (p != NULL) {
        free(P);
        P = NULL;
    }
}
```

只改变型参，置空操作无效

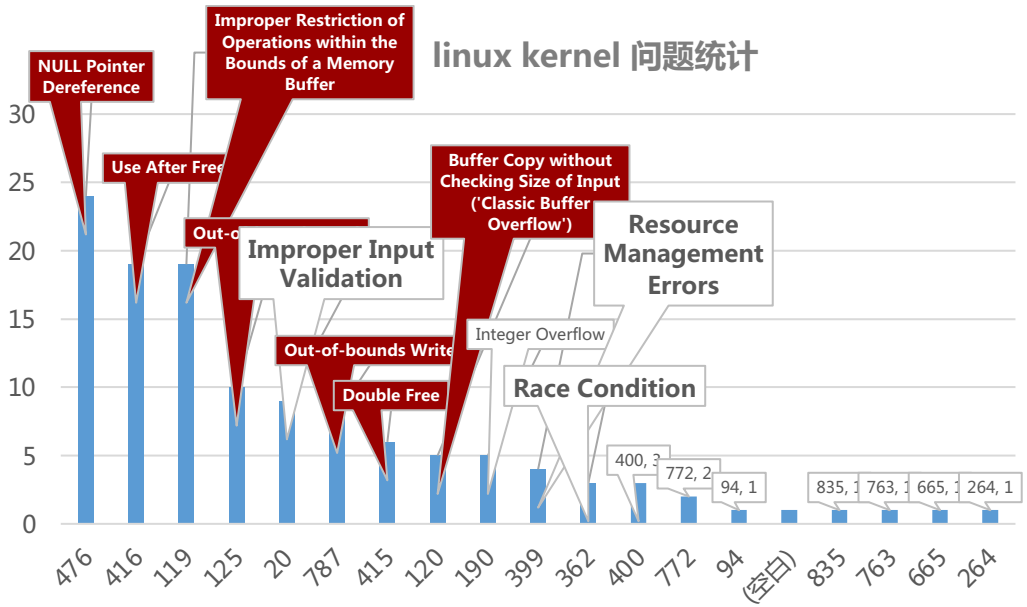
内存安全问题当前依然是软件缺陷的最主要来源





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%的Bug**均由内存安全问题导致
- Chrome浏览器项目中大约**70%的Bug**是内存安全问题
- MacOS/iOS中**60-80%的Bug**由内存安全问题导致

内存安全问题一直是软件缺陷的主因，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，一般至少是系统崩溃

目　录

CONTENTS

软件缺陷分析与检测相关技术图谱

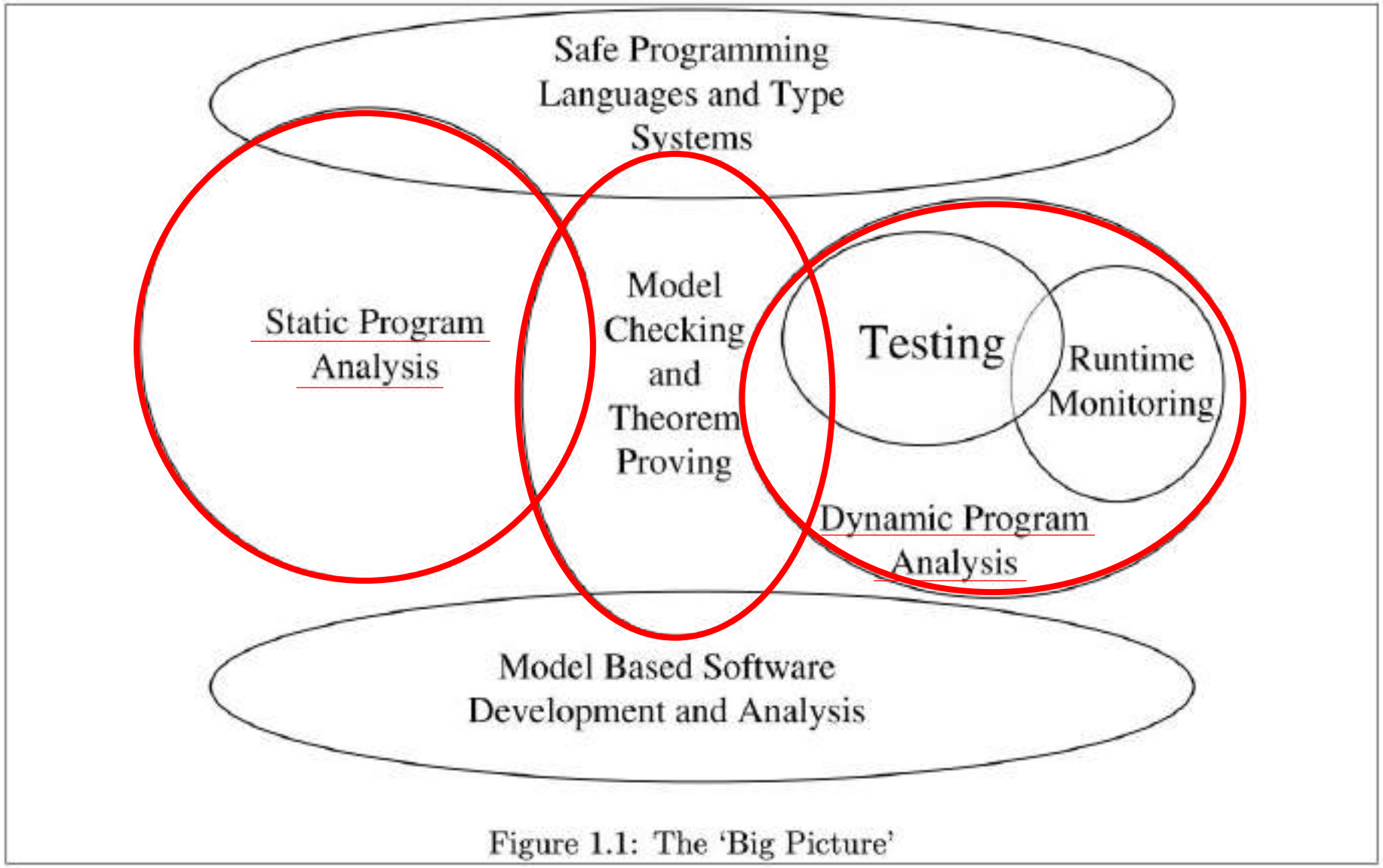
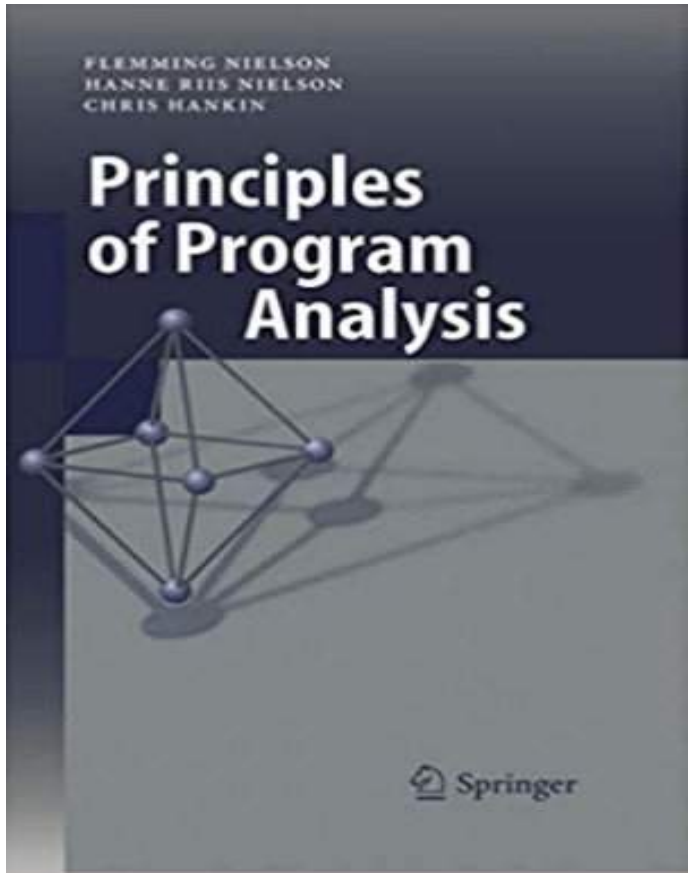


Figure 1.1: The 'Big Picture'

---- Sen K. Scalable automated methods for dynamic program analysis[J]. 2006.

基础理论：程序分析

- 程序分析（Program Analysis）是一种通过程序来自动化分析程序行为的技术



- 数据流分析 （Data Flow Analysis）
- 约束分析 （Constraint-Based Analysis）
- 抽象表达 （Abstract Interpretation）
- 类型效果系统 （Type and Effect System）

学术界使用了大量高级数学进行程序分析理论的研究，例如伽罗华代数，晶格（**Lattice**），集合论，逻辑等

程序员眼中的程序分析

```
void TestLocalRemalloc(int cond) {
  int *p = (int*)malloc(sizeof(int));
  //…
  if (cond == 2) {
    free(p);
    p = (int*)malloc(sizeof(int));
  } else if (cond = 3) {
    p = (int *)malloc(sizeof(int));
  }
  //…
  memset(p, 1 , sizeof(int));
  printf("%d", *p);
  free(p);
}
```

通过代码审查提高代码质量

是否能够自动识别出内存泄漏？

- 字符串扫描，缺陷模式匹配？

跨过程的内存泄漏如何分析？

```
void* CreateData(size_t num) {
  if (num > 0){
    return malloc(num * sizeof(char));
  }
  return NULL;
}

void TestLocalForgetFree(int data) {
  void *p = malloc(10 * sizeof(char));
  void *q = CreateData(20);
   if (data > 0) {
    free(p);
    return;
   }
   free(p);
   free(q);
}
```

是否能够自动识别出内存泄漏？

全局变量的影响

```
int* p_mem;
int cond_code;

void UAFTestCase2IncompleteFree() {
  int* q = (int*) malloc(MEM_SIZE * sizeof(int));
  if (q == NULL) {
    return;
  }
  p_mem = q;  //   1

  if (cond_code < 0) {
    free(q);   //   2
    q = NULL;
    // p_mem is not set empty
  }
}
```

```
void UAFTestCase2Free() {
  if (p_mem != NULL) {
    free(p_mem);
  }
}

int UAFTestCase2() {
  UAFTestCase2IncompleteFree();
  UAFTestCase2Free();   // 3
  return OK;
}
```


是否能够自动识别出双重释放？

结构体字段如何建模？

```
typedef struct {
    int type;
    int data1;
    int data2;
    char buf[256];
    int data3;
    char data4;
    int data5;
    int data7;
} TEST_MSG;

typedef struct {
    int item1, item2, item3;
    struct {
        int x, y;
        struct {
            int d1, d2, d3;
            TEST_MSG* pMsg;
        } inner_data;
    } data1;
} OUTTER_MSG;

OUTTER_MSG oMsg;
```

```
static void process_4(TEST_MSG* pMsg, int x, int y) {
    pMsg->data4 -= 100;
}

static void process_3(TEST_MSG* pMsg, int x, int y) {
    pMsg->data1 = x + y;
    if (pMsg->type == 3)
        pMsg->data2 = pMsg->data1 * x + y;
    if (pMsg->data5 + pMsg->data4 > 100)
        process_4(pMsg, x, y);
}

static void process_2(TEST_MSG* pMsg, int x, int y) {
    int temp = 2;
    temp = x + y * 10;
    if (temp < 10)
        temp = 10;
    else
        temp *= 100;

    pMsg->data7 = temp + 100;
    process_3(pMsg, x, y);
}
```

```
static void process_1(int x, int y) {
    TEST_MSG* pMsg = oMsg.data1.inner_data.pMsg;

    int temp = 2;
    temp = x + y * 10;
    if (temp < 10)
        temp = 10;
    else
        temp *= 100;

    for (int i = 0; i < 10; i++)
        process_2(pMsg, x, y);
}

void process_global(int x, int y) {
    int temp = 5;
    temp = x + y;
    if (temp > 10)
        temp = 10;
    else
        temp *= 10;

    oMsg.data1.inner_data.pMsg = (TEST_MSG*)malloc(sizeof(TEST_MSG));

    if (oMsg.data1.inner_data.pMsg == nullptr)
        return;

    TEST_MSG* pMsg = oMsg.data1.inner_data.pMsg;

    if (pMsg->type == 1)
        pMsg->data1 = temp;
    else if (pMsg->type == 2)
        pMsg->data1 = x;
    else
        process_1(x, y);
}
```

在对消息指针操作前没有进行判空，是否安全？

程序分析的基础技术

- 数据流
- 控制流
- 域敏感
- 流敏感
- 约束求解
- ……

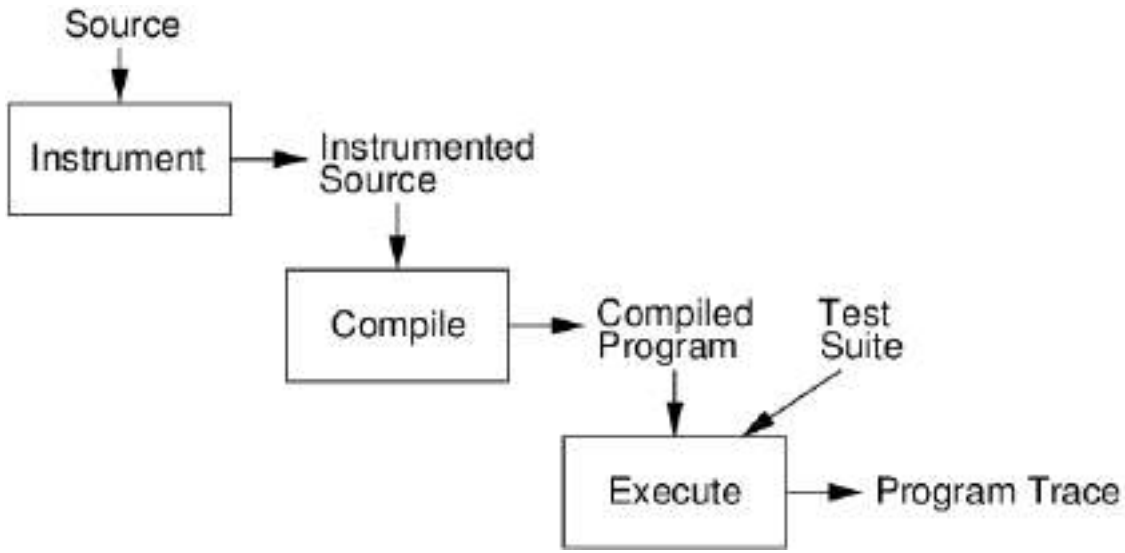
程序分析与程序员的日常工作关系密切。静态检查工具和编译器中使用了大量程序分析技术。

学术界的重心是理论研究，工业界更关心如何利用程序分析技术解决实际问题，例如安全、性能、代码质量等。

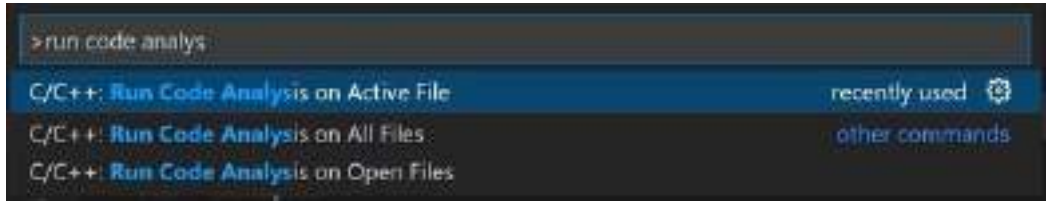
程序静态分析 VS 程序动态分析

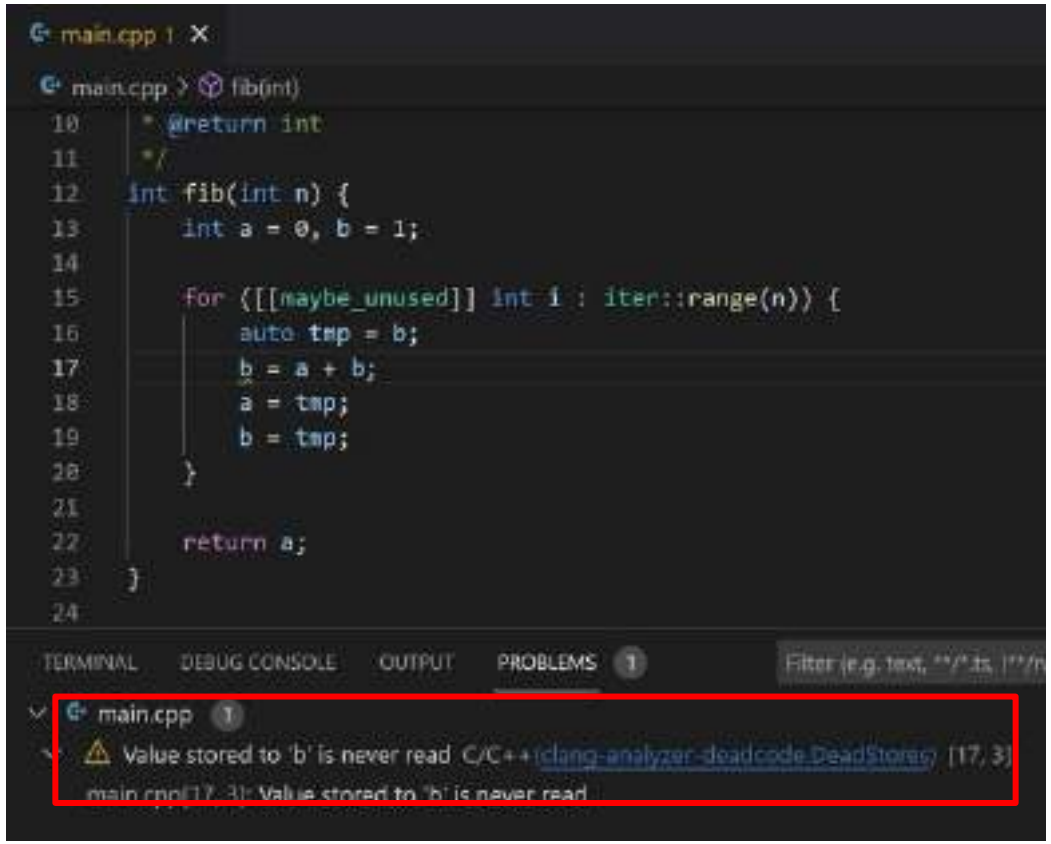
- 程序静态分析是在不执行程序的前提下对代码进行扫描，估计代码是否满足规范性、安全性、可靠性、可维护性等指标

- 程序动态分析是在一个真实的或是模拟的运行环境上运行给定的程序，确认或发现程序运行时的性质

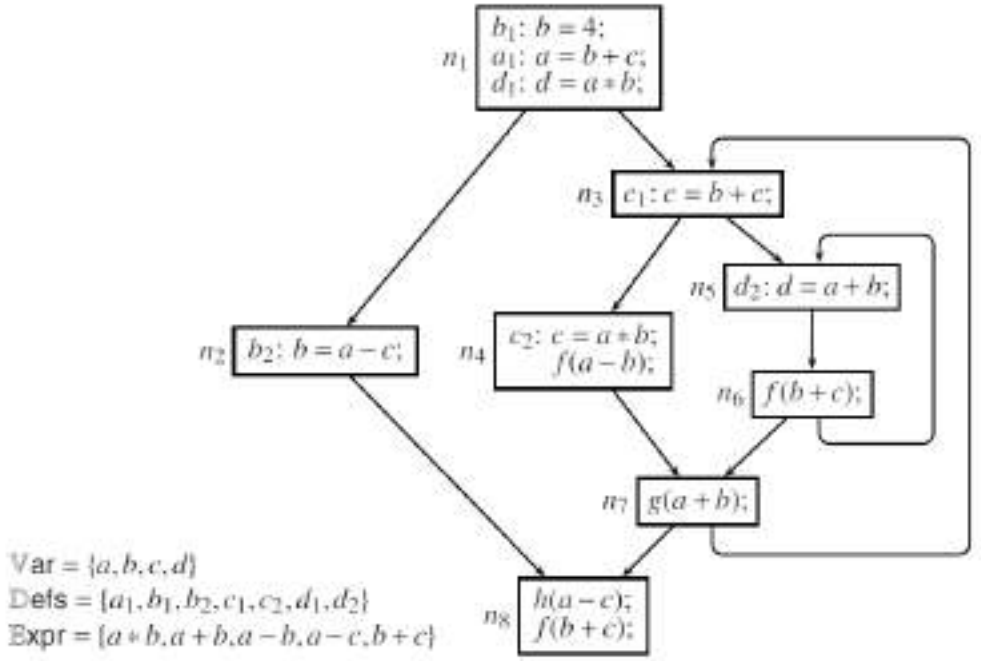


程序静态分析



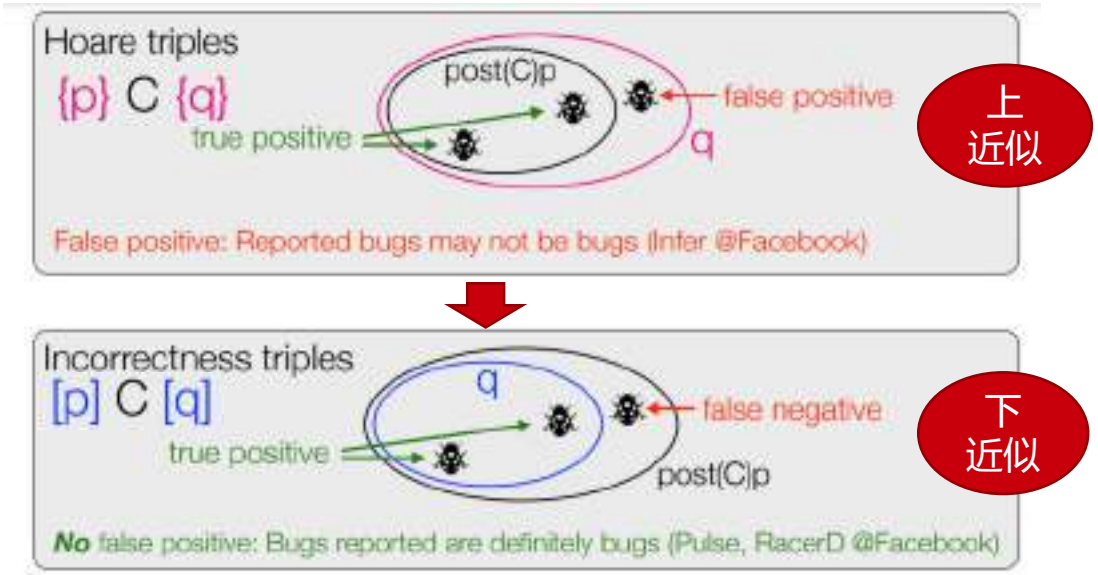


程序静态分析通过词法分析、语法分析、控制流、数据流分析等技术对程序代码进行分析。静态分析技术向模拟执行的技术发展以能够发现更多传统意义上动态测试才能发现的缺陷，例如符号执行、抽象解释、值依赖分析等等并采用数学约束求解工具进行路径约减或者可达性分析以减少误报增加效率。



静态分析技术成功案例 – Meta/Facebook的Infer工具

- FB Infer前世今生
  - 来源于英国多团队近10年的合作研究（O'Hearn, Distefano, Calcagno等人）& UKRI大量科研经费资助
  - 学术原型：Smallfoot（符号执行; 2004） -> SpaceInvader（可组合分析与验证; 2009）
  - 2009成立Monoidics公司后推Infer工具
  - 2013Facebook收购后改为FB Infer(开源)：https://fbinfer.com
  - 2013年至今Facebook持续投入
  - 被很多公司采用：Facebook, AWS, Uber, Spotify, M&S, Oculus, WhatsApp, Microsoft…
- Infer使用的理论与技术
  - 基于bi-abduction技术来支持组合推理与验证：
  - 基于抽象解释的静态分析
- Infer功能特点：面向部分内存安全属性的静态分析
  - 检查Android/Java代码空指针异常，资源泄漏，race conditions等
  - 检查C/C++/Obj C代码的空指针问题、内存泄漏、不可用API等问题
  - 具有较好的可扩展性（可分析>百万行代码）
  - 支持增量分析，已集成到Facebook的CI/CD
  - 每个月发现数百~数千代码缺陷/漏洞
- Infer研究的最新发展趋势
  - **从"无漏报;高误报"(证明正确) 到"无误报;允许漏报"(发现错误)**
  - 从Over-Approximation(Separation Logic)到Under-Approximation(Incorrectness Separation Logic)



静态分析技术成功案例 – Meta/Facebook和亚马逊的增量分析技术

传统的全量分析

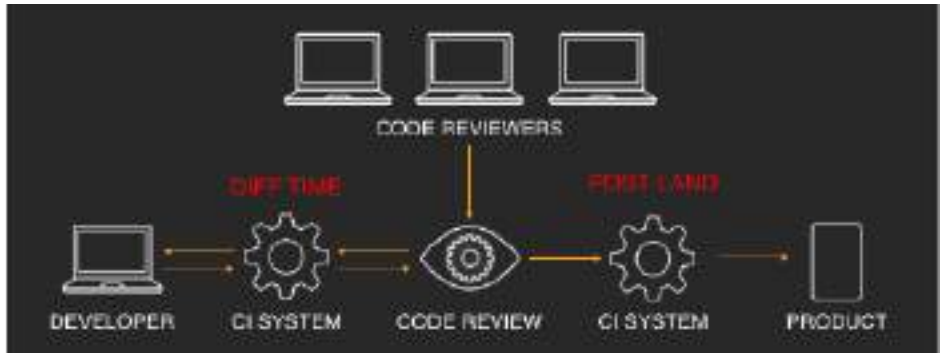
- 要么面向固定的软件版本，一旦软件改变，就需要大量的重复建模、编码、证明工作；
- 要么围绕从规范中生成正确代码而设计；

以上两种方法都不适用于持续开发中的软件，新版本的代码无法自动从先前版本的证明中继承正确性。

增量分析 / 持续分析

- 当代码变更时，只分析、验证差异部分，以降低分析验证的开销；
- 当代码变更时，无需或只需少量的手动工作，就能进行自动化的验证；

Meta/Facebook使用 Infer 对代码差异部分进行增量检查



大代码 小定理

对C、Java等语言代码的通用性质（内存安全、资源泄露等问题）的验证，具有较好的伸缩性、通用性以及自动化验证能力，可以匹配持续迭代的开发流程。

亚马逊介绍了一套面向开源TLS实现s2n中DRBG、HMAC、TLS handshake关键性质的持续验证方案

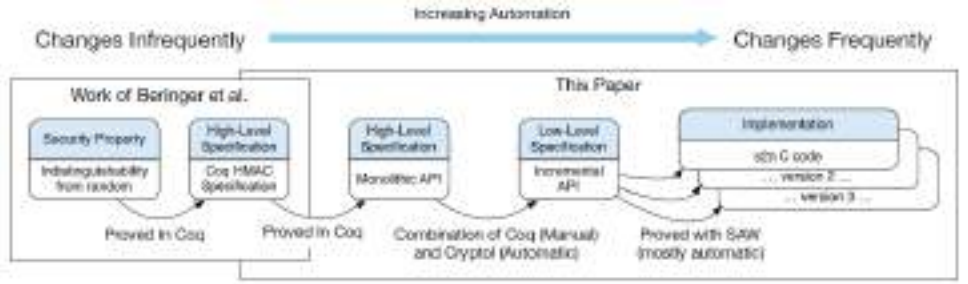


Fig. 1. An overview of the structure of our HMAC proof.

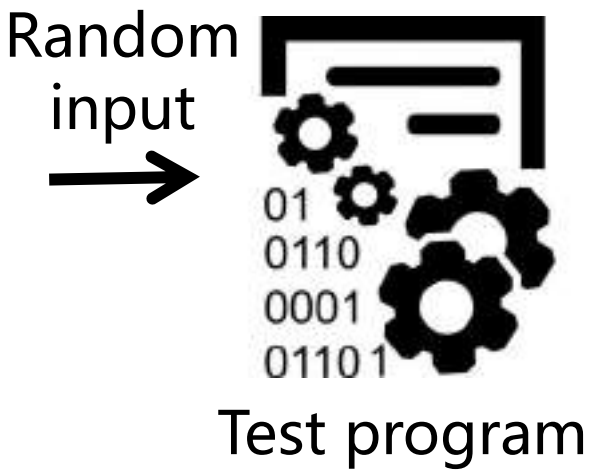
小代码 大定理

对于特定业务场景，其高层需求是相对稳定的，对于给定的高层形式化性质，随着实现代码调整，能够实现持续的自动验证。

程序动态分析

生成测试输入/用例 + 动态执行程序

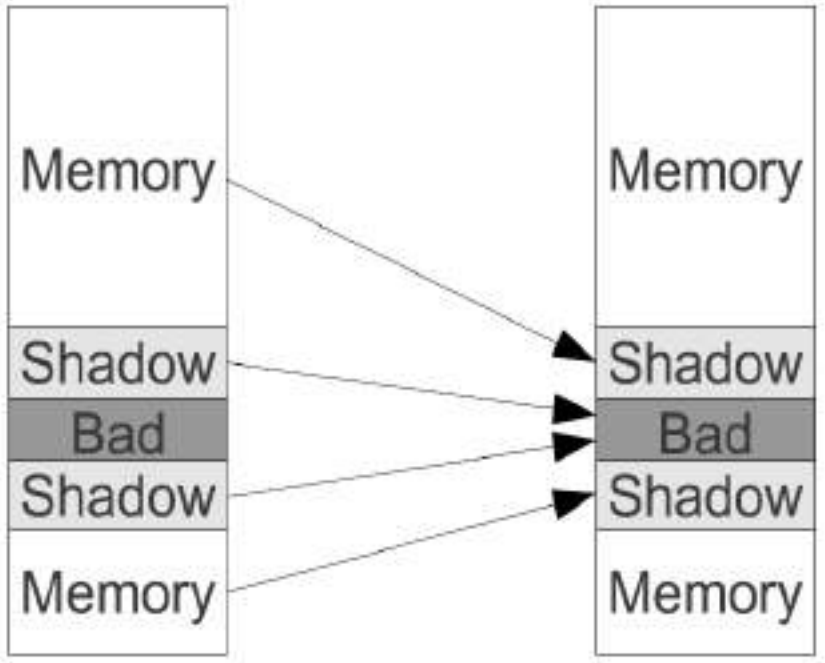
- 模糊测试（Fuzzing）：自动生成海量的输入来执行程序，监视程序运行过程是否发生异常。
- 手工测试
- 蜕变测试
- 差分测试



黑盒模糊测试 (Miller et al.’ 89)

收集程序运行时的信息，判断程序是否发生异常

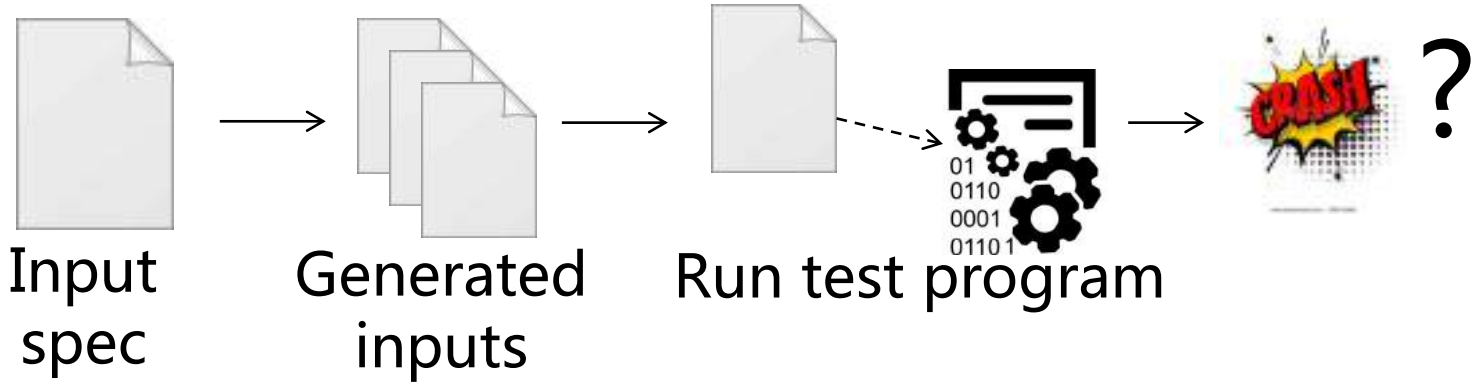
- 地址消毒：由谷歌提出，通过影子内存对内存访问进行保护，捕捉非法访问



地址消毒技术

模糊测试 – 改进

基于生成的模糊测试



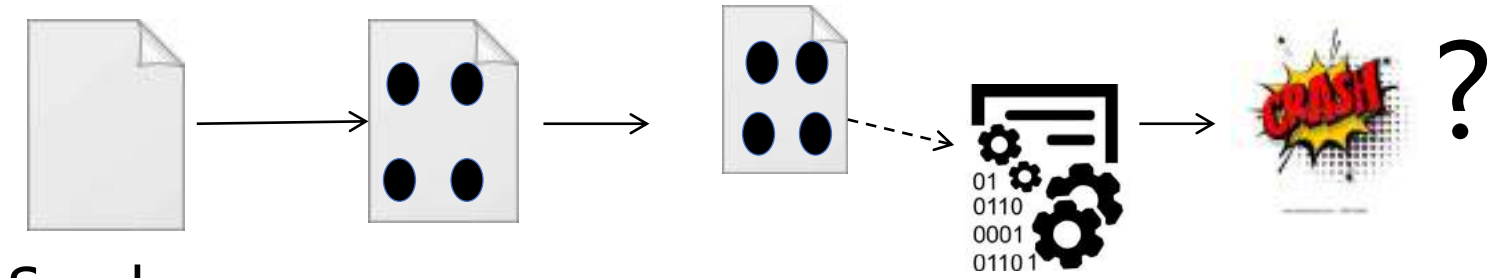

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size c
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);     // Width
                s_push_int(0x14, 1);     // Height
                s_push_int(0x8, 3);      // Bit Depth
                s_push_int(0x3, 3);      // ColorType
                s_binary("00 00");       // Compressi
                s_push_int(0x0, 3);      // Interlace
```

Example: Sample PNG Spec

基于变异的模糊测试



- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus
- Use fuzzing tool (or script)
  - Collect seed PDF files
  - Mutate that file
  - Feed it to the program
  - Record if it crashed (and input that crashed it)

Example: Fuzzing a PDF viewer

动态测试技术成功案例 – Google的AFL模糊测试工具

AFL介绍

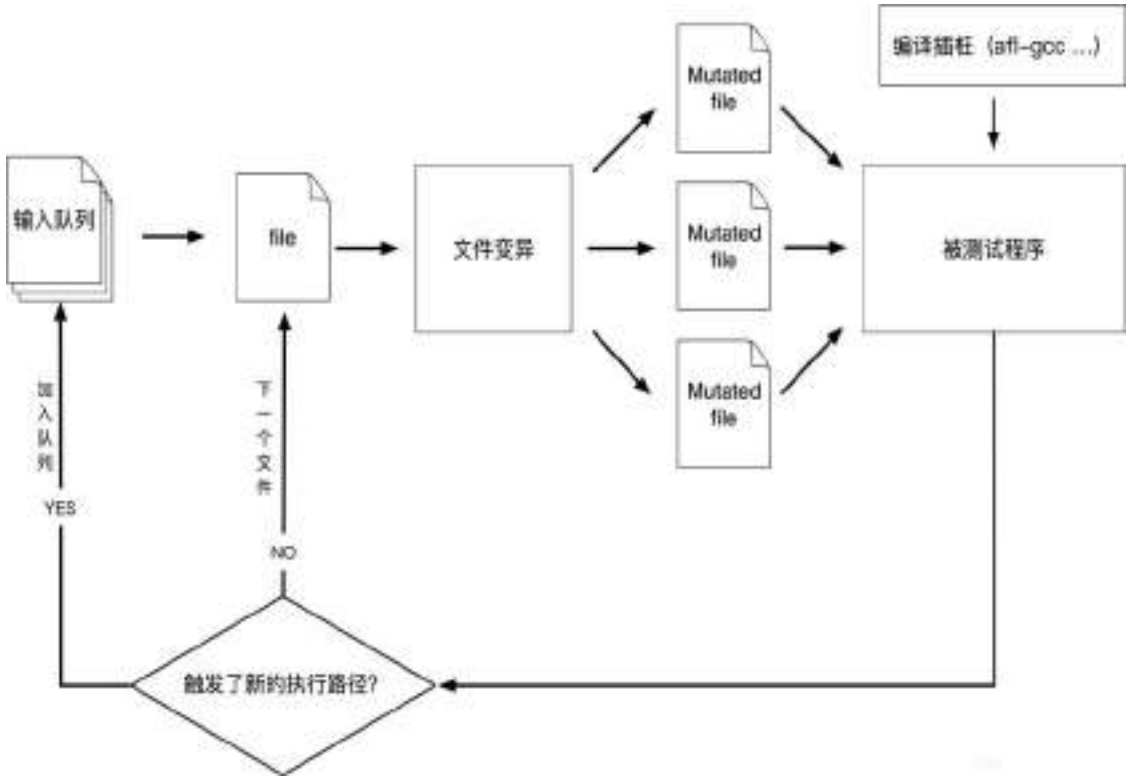
- AFL(American Fuzzy lop)是一款基于覆盖引导的灰盒模糊测试工具，最早由Google的安全研究员Micha Zalewski开发。
- 它在诞生之初便在知名软件项目中检测到数十个重大软件错误，包括X.Org Server, PHP，OpenSSL，pngcrush，bash，Firefox，BIND，Qt，and SQLite。
- 目前，作为最广泛使用的Fuzzing工具，AFL已经发现超过15000个漏洞，且基于AFL做二次开发的开发安全研究工作在近年已经有超过100篇高水准学术论文发表。

AFL的原理思想

- 自动化地在无穷多个输入空间里寻找有限的输入，触发漏洞
- 采用遗传算法来有效地提高测试用例的代码覆盖率

AFL的工作流程

1. 从源码编译程序时进行插桩，以记录代码覆盖率（Code Coverage）
2. 选择一些输入文件，作为初始测试集加入输入队列（queue）
3. 将队列的文件按一定的策略进行‘突变’
4. 如果经过变异文件更新了覆盖范围，则将其保留添加到队列中
5. 上述文件会一直循环进行，期间触发了crash的文件会被记录下来



Iterative testing, keep GOOD seeds, report bugs

动态测试技术成功案例 – Google的地址消毒剂（Sanitizer）框架

Sanitizer工具集介绍

- Sanitizers 是由 Google 研发团队提出的用于动态检测软件中常见内存错误的工具集
- 在 LLVM 框架上实现，已集成在gcc、clang编译器中，在工业界被广泛使用
- Google 工程师公开了它们的源代码和算法：https://github.com/google/sanitizers

AddressSanitizer工具介绍

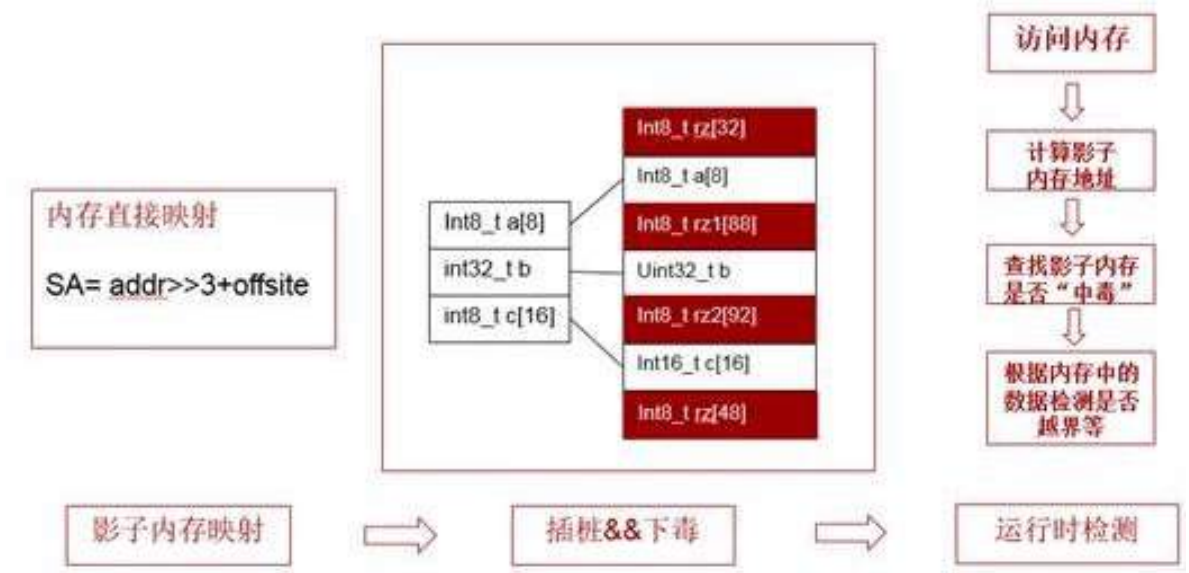
- 一款针对C/C++程序中内存访问缺陷的动态检测工具；在编译时对源码进行插桩，在运行时检测缺陷

AddressSanitizer工具支持检测的内存安全问题

- 使用已释放内存（野指针）
- 堆变量越界
- 栈变量越界
- 全局变量越界
- 函数返回局部变量地址
- 内存申请释放类型不匹配

AddressSanitizer工作原理

- 利用影子内存去记录程序中任意一个字节内存是否可安全访问
- 访问内存时查询影子内存，捕捉非法访问。



形式化验证

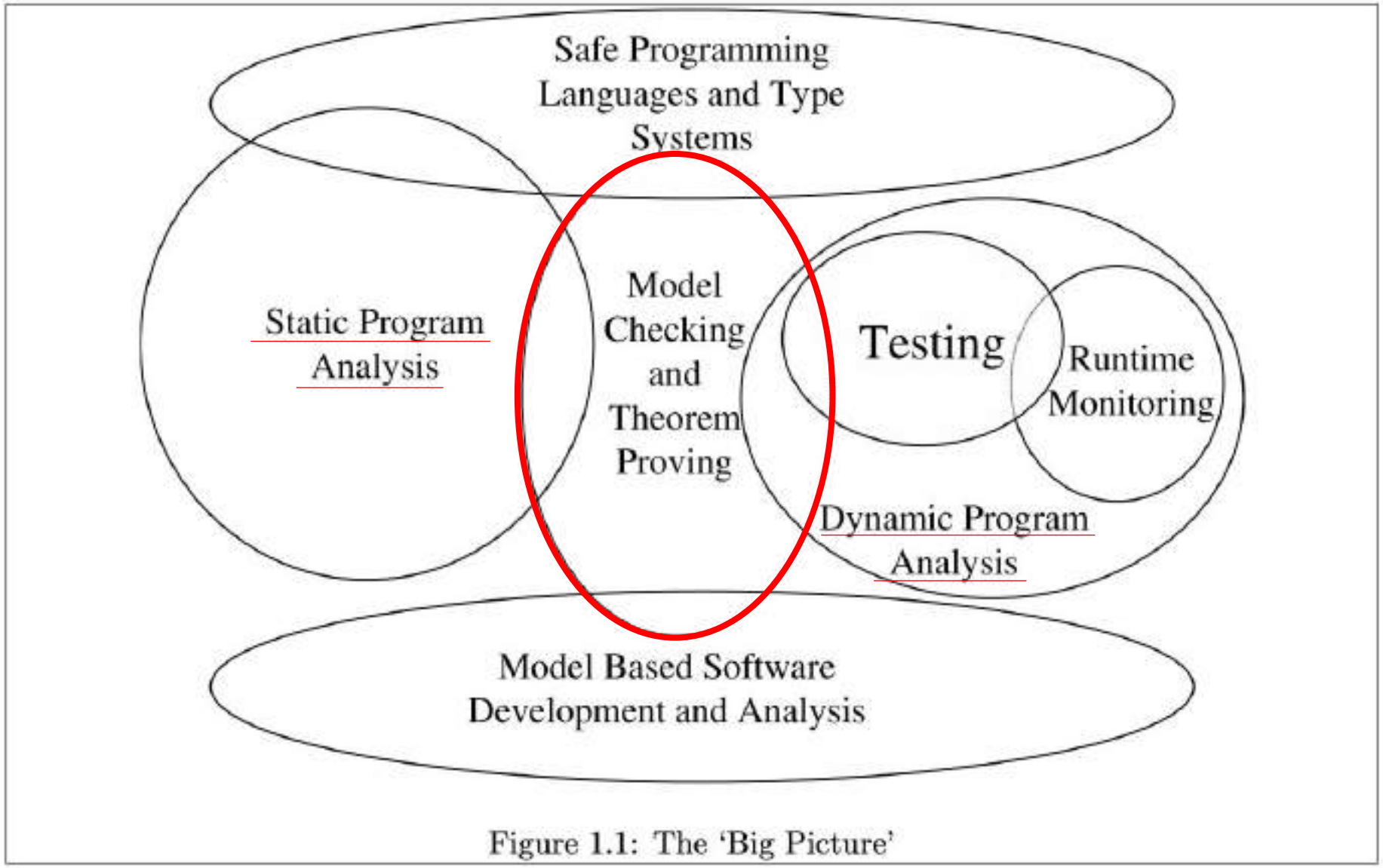


Figure 1.1: The 'Big Picture'

---- Sen K. Scalable automated methods for dynamic program analysis[J]. 2006.

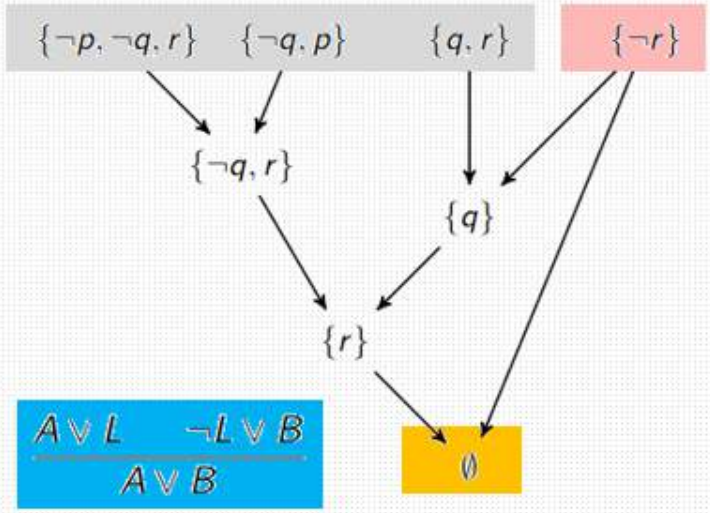
形式化验证

形式化验证（verification）技术分类

统一的目标：给定一个系统的形式化描述，判断其是否满足某一性质
都不能解决：性质之间是否一致、要求系统具备的性质是否完整（很多是需求validation的工作）

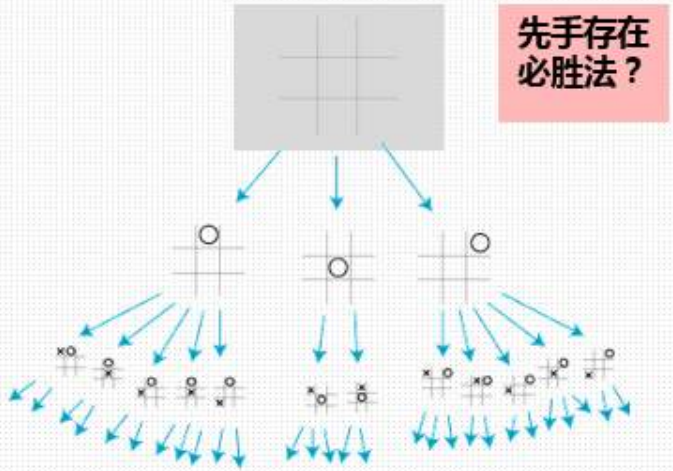
定理证明	模型检测	抽象解释

用规则推理的方式来寻找系统和性质之间的关系



■哪一步选哪些规则能证明/伪该关系？
■人工选和自动选的边界在哪里？

先穷举系统行为结果，再判断其是否符合给定性质



■状态多到计算机无法穷举，哪怕对状态进行了抽象和分类？
■在状态图中判断某性质的复杂度？

根据验证性质，对系统行为进行针对性抽象，求是否存在不动点

```
int x = 2;
int y = 4;
while (1)
{
    if (x == 3)
        break;
    x = x + y;
}
```

```
int x = 偶数;
int y = 偶数;
while (1)
{
    (x是偶数)
    if (x == 奇数)   // 奇数!=偶数
        break;

    (x是偶数)
    x = x + y;       // 偶数+偶数=偶数
    (x是偶数)
}
```

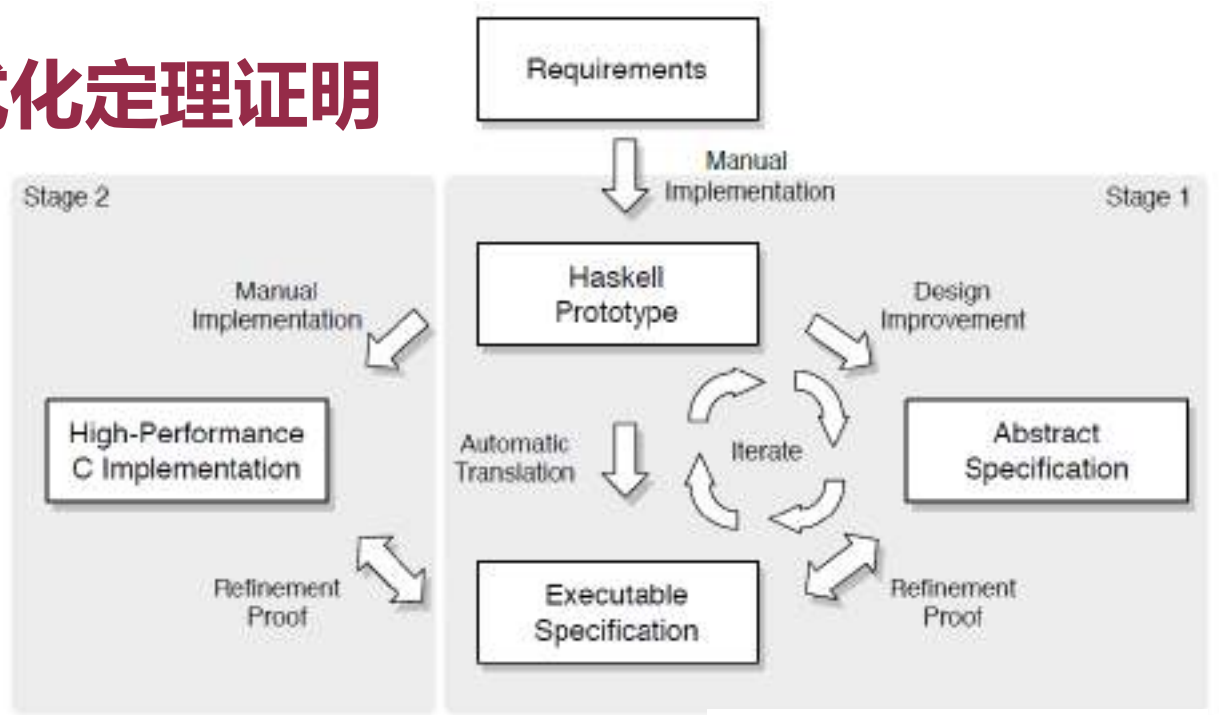
■ 抽象域囊括的状态大于系统真实状态，容易虚警

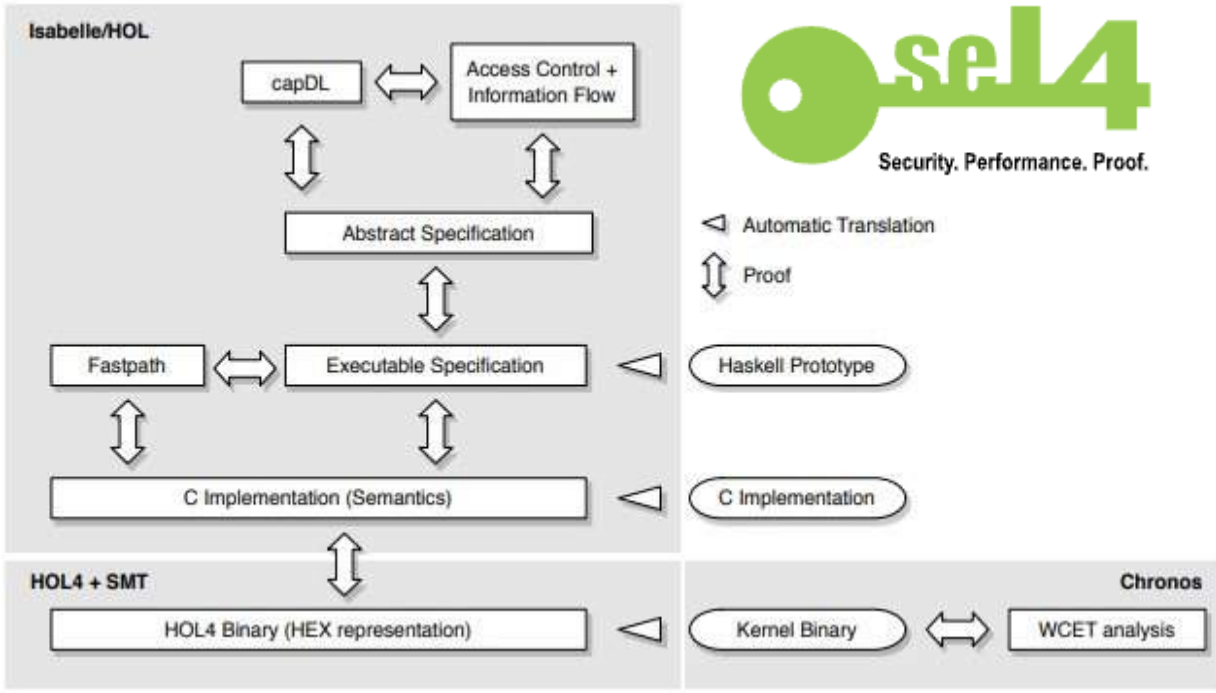
形式化验证技术成功案例 – seL4的形式化定理证明

- seL4: 首次对整个OS内核的首个完整形式化验证
  - 包括Kernel设计精化证明、包括功能正确性，安全性等的证明
  - Kernel设计和证明分2阶段：第一阶段包括抽象规格（what）和具体规格（how），以及二者之间的精化证明；第二阶段包括高性能C代码实现以及C代码满足具体规格的证明
  - 抽象规格层之上还有安全性模型（访问控制、信息流等）及相关证明
  - C代码下包括二进制实现的证明和WCET（最弱执行时间）分析
  - 使用Isabelle/HOL交互式定理证明（主要人写证明；机器检查）
- 重量级形式化方法最成功的应用：
  - 投入成本高，但证明提供了比CC最高级别EAL7更高的保障（EAL7不要求“代码实现设计模型”的形式化证明）

任务	开发（Haskell+C）	正确性证明	安全性证明	二进制验证
投入	2.2人年	20.5人年	4.1人年	2人年

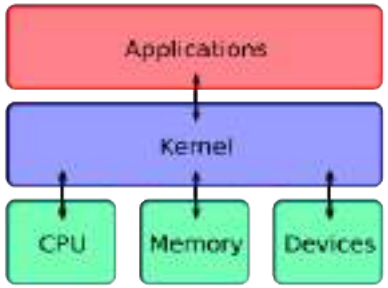
  - seL4：每行源代码成本：362美金（后续新Kernel只需127美金：首次尝试9人年花在在形式化语言框架、证明工具和证明自动化）
  - CC-EAL6典型成本：1千美金/代码行
- seL4已经被广泛使用在很多领域
  - 在DARPA高可信网络军事好系统（HACMS）的一项实验中，基于seL4的定制飞行控制软件黑客无法攻击/破解





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：

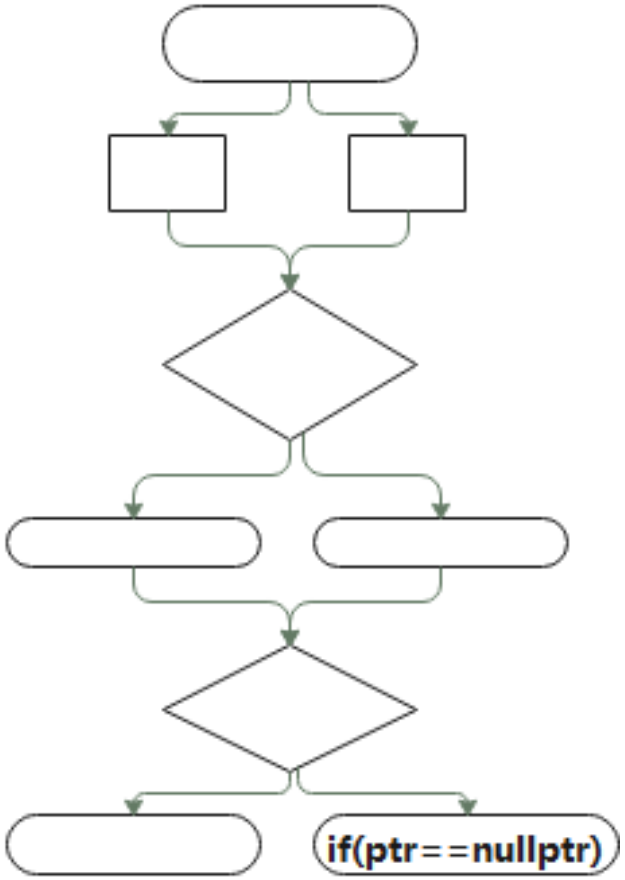
技术路线	代表工具	准确率	覆盖率	可扩展性	评价
静态分析	Cppcheck/Infer	低	高	较好	分析成本低但误报率较高
动态分析	AFL/Asan	高	低	好	分析结果准确但漏报率较高
形式化验证	RefinedC/Frama-C	高	高	差	仅适用于不计成本的高安场景

如何针对不同场景结合使用多种不同技术的组合拳？

挑战一：路径爆炸

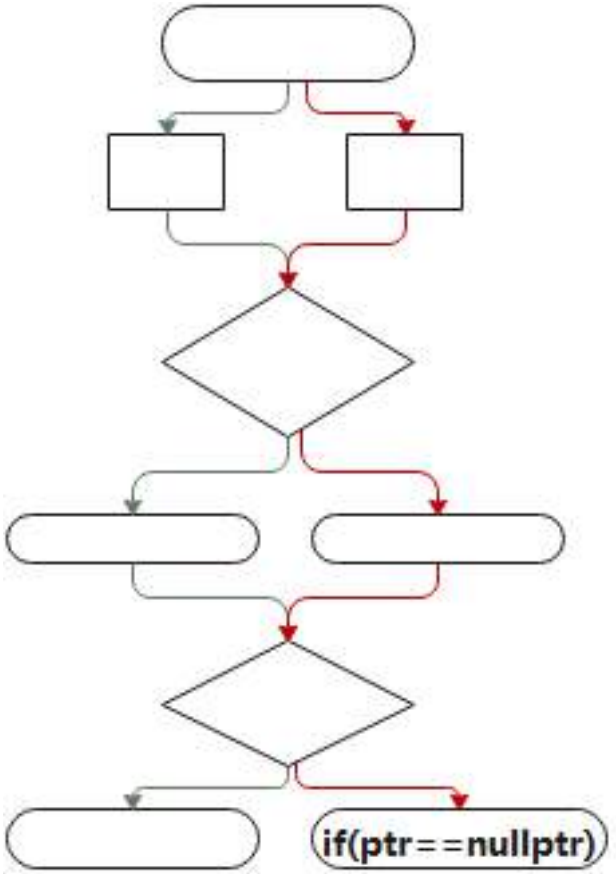
程序中的路径组合指数增长，超出静态分析工具在有限资源和有限时间内能够承受的最大上限

根据数据依赖和控制依赖的关系，进行自下而上或者自上而下的程序切片，可以有效抑制路径爆炸



路径指数增长！！！

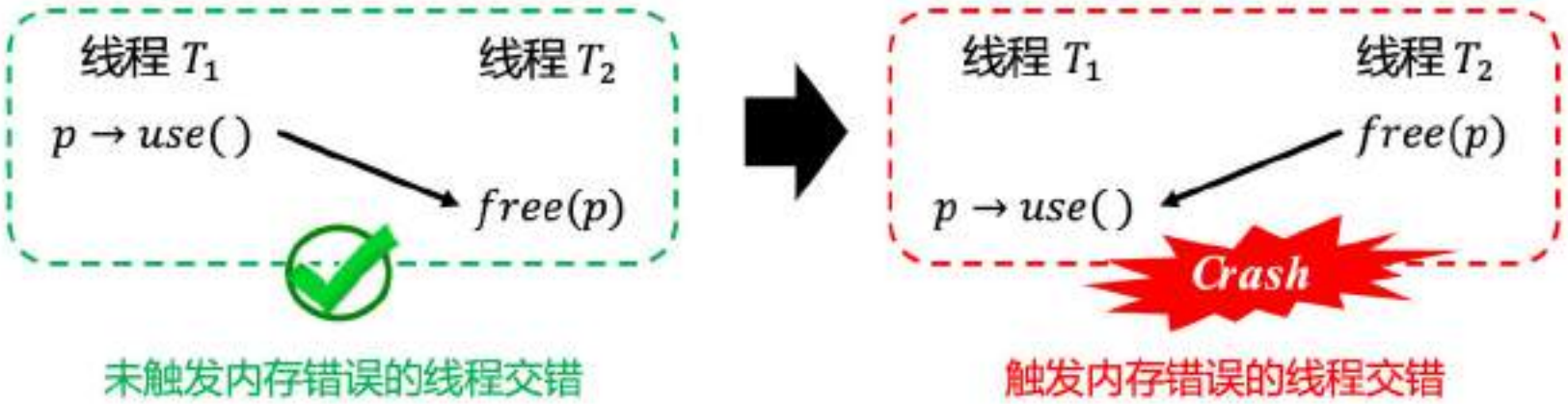
10^80 宇宙中的原子数
2^361 围棋的局面
2^n 程序中的路径



只对相关的路径和变量进行分析

挑战二：并发执行交错

- 多线程执行顺序具有不确定性：一些潜在漏洞的触发是随机发生的；并发漏洞通常仅在罕见的、特定的执行交错下才暴露出来



未触发内存错误的线程交错　　触发内存错误的线程交错

- 执行交错的空间十分庞大：复杂度随线程数及线程执行指令长度的增长呈指数级增长；穷尽地枚举所有可能的线程交错通常不可行

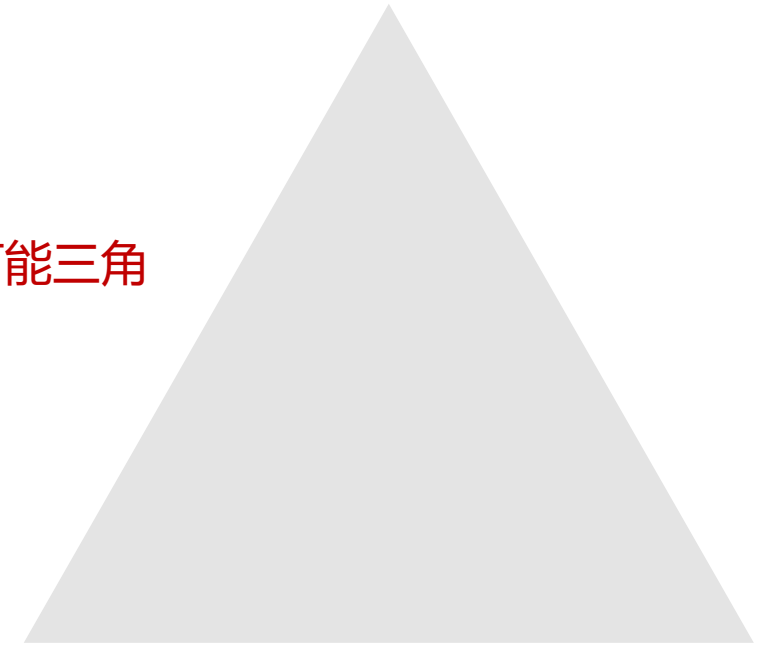
如果两个线程分别具有m和n条指令，那么所有可能的线程交错数量是：

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

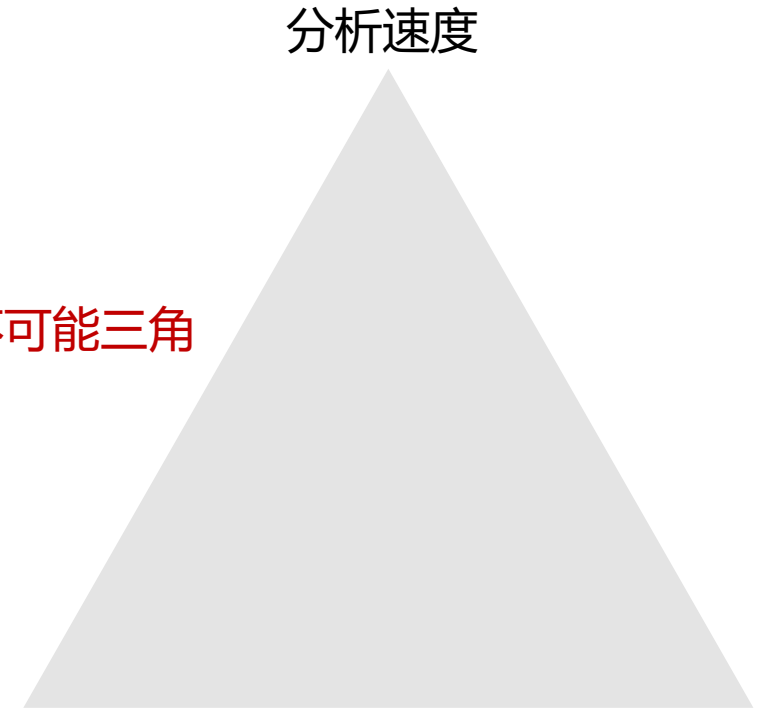
$m = n = 4$	$m = n = 8$	$m = n = 16$
70	13K	601M

挑战三：程序分析主要指标

蒙代尔不可能三角



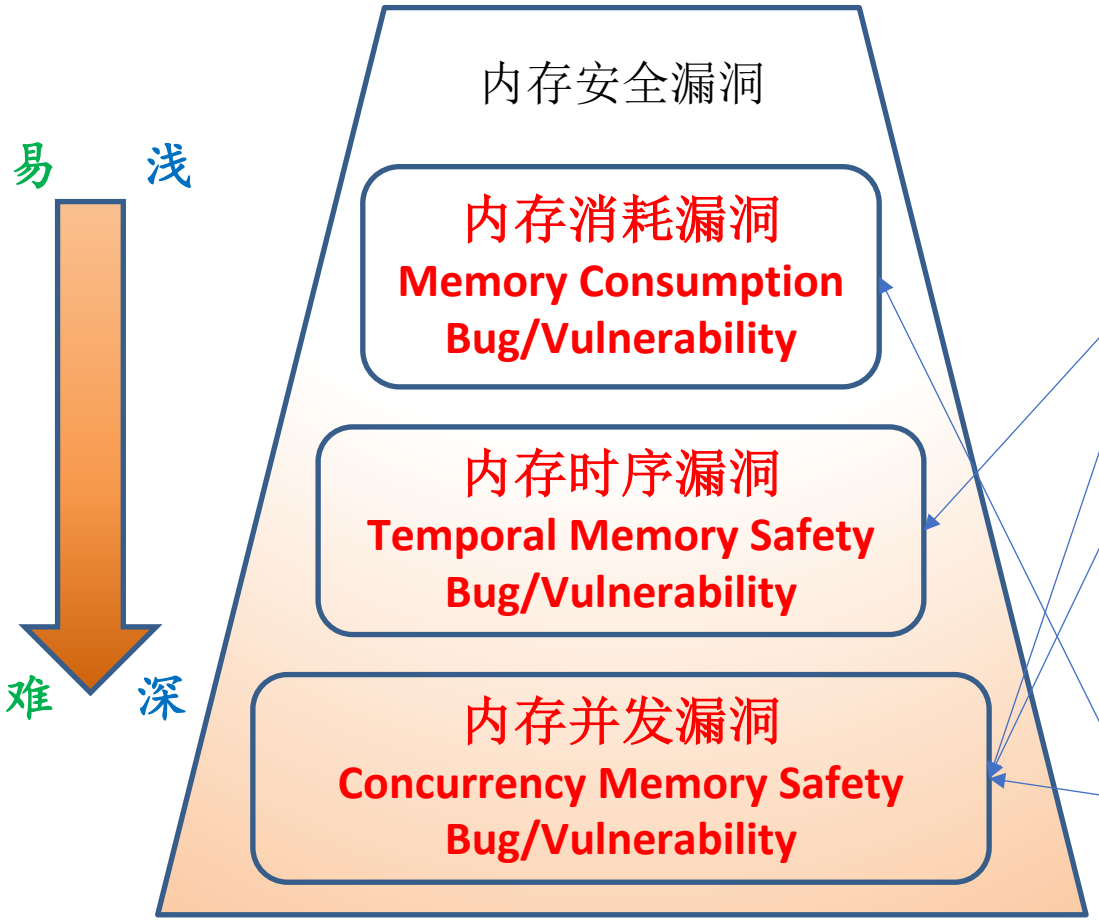
静态分析不可能三角



目　录

CONTENTS

三类隐藏深、难发现、危害大的内存安全漏洞

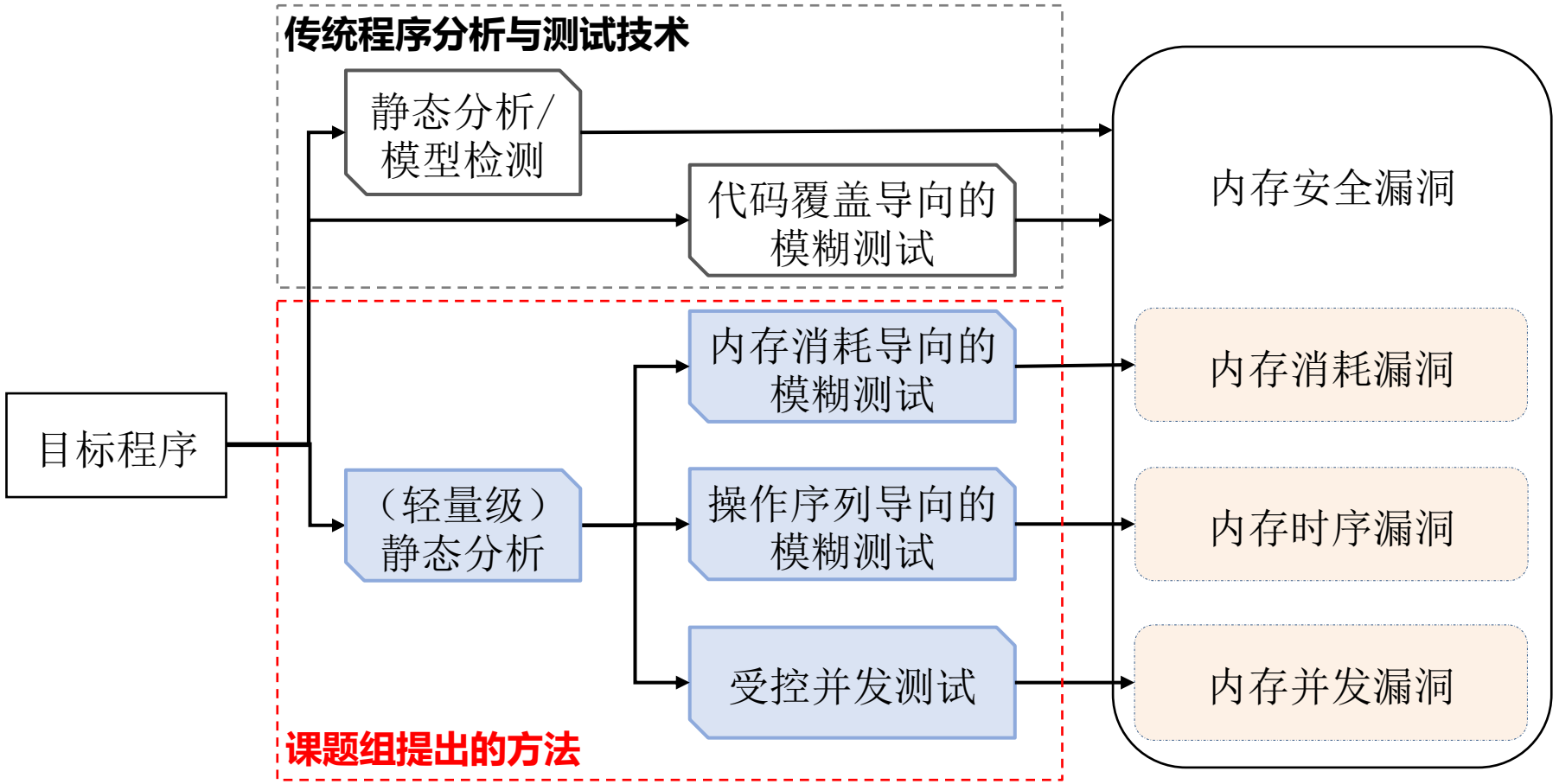


Rank	ID	Name	Score
1	CWE-787	Out-of-bounds Write	64.20
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11
4	CWE-20	Improper Input Validation	20.63
5	CWE-125	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

课题组研究进展：通用可扩展的内存安全漏洞检测框架

- 侧重自动化的分析与测试，融合了轻量级程序分析技术与自动化测试技术
- 旨在发现业界实际应用软件（十万行-百万行代码）中的内存安全问题，提升软件质量
- 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



ICSE' 20a: MemLock: Memory Usage Guided Fuzzing

ICSE' 20b: Typestate-Guided Fuzzer for Discovering Use-after-free vulnerabilities

ICSE' 22: Controlled Concurrency Testing via Periodical Scheduling

内存消耗漏洞检测技术研究（ICSE 2020a）

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。

不受控的递归调用漏洞（CVE-2018-17985）

```
struct demangle_component *
cplus_demangle_type (struct d_info *di) {

  // 'peek' is a single character extracted from the input directly
  char peek = d_peek_char (di);

  switch (peek){
    ...
    case 'P':
      ret = d_make_comp (di,
        DEMANGLE_COMPONENT_POINTER,
        cplus_demangle_type (di), NULL);
      break;
    case 'O':
    ...
  }
  ...
}
```

递归函数

输入	递归调用深度	种子保留状态
p	1	Interesting
pp	2	Interesting
pppp	4	Interesting
ppppppp	128	Interesting
pppppppp	260	Discard
ppppppppp	...	Discard
pppppppppppppp	35006+	Trigger the bug

不受控的内存分配/内存泄漏（CVE-2018-4866）

```
class EXIV2API DataBuf {
public:
  // Constructor with an initial buffer size
  explicit DataBuf(long size) : pData(new byte[size]), size(size) {}
  ...
  byte* pData; // Pointer to the buffer
  size_t size; // The current size of the buffer
};

void Jp2Image::readMetadata() {
  while (io_->read((byte*)&subBox, sizeof(subBox)) ==
    sizeof(subBox) && subBox.length ) {
    subBox.length = getLong((byte*)&subBox.length, bigEndian);
    DataBuf data(subBox.length); // Allocation without checking
    ...
    io_->seek(position - sizeof(box) + box.length, BasicIo::beg);
  }
}
```

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

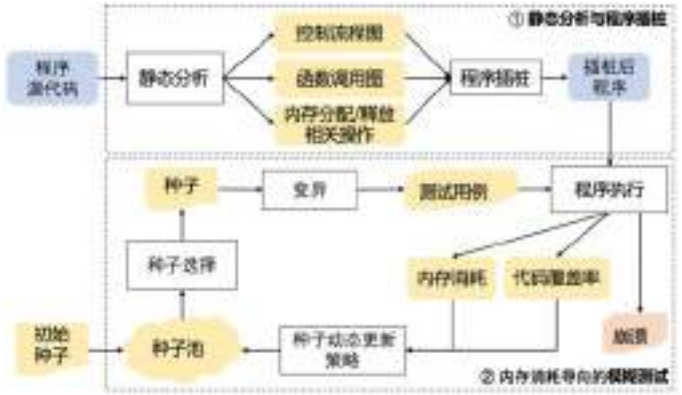
输入（size）	种子保留状态
2048	Interesting
8192	Discard
61440	Discard
2097152	Discard
53687092	Discard
...	Discard
4294967296+	Trigger the bug

问题挑战：

- 现有的技术手段对于发现内存消耗漏洞存在一定的盲区。
- 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。

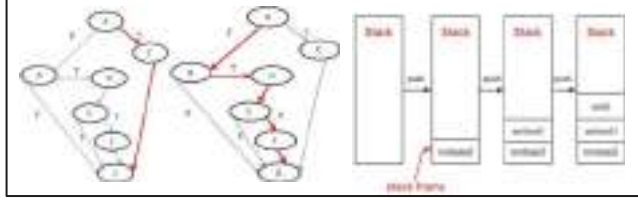
基于轻量级静态分析与模糊测试的检测手段

- 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



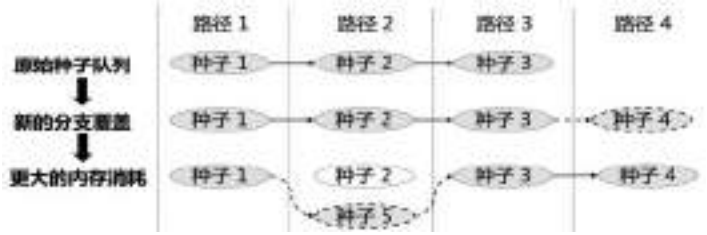
阶段一：静态分析与程序插桩

- 分析三类静态信息（控制流程图、函数调用图、内存分配/释放操作）
- 通过程序插桩在运行时收集动态信息



阶段二：内存消耗导向的模糊测试

- 使用内存消耗和覆盖率信息指导模糊测试
- 提出种子动态更新策略来有效维护种子池



Cheng Wen et al. MemLock: Memory Usage Guided Fuzzing, accepted by ICSE 2020

内存时序漏洞检测技术研究（ICSE 2020b）

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。

问题挑战：

- 内存时序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

启发性示例：

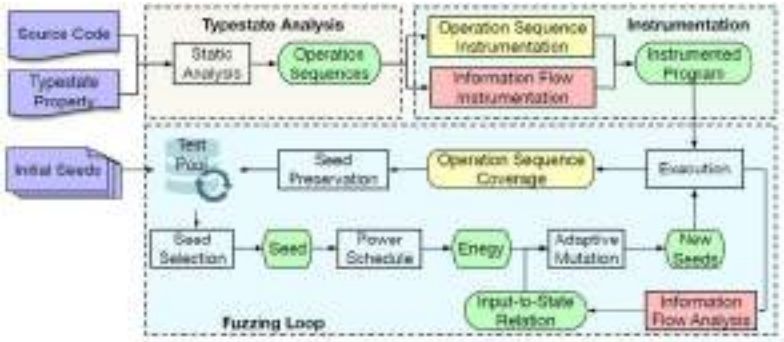
按照以下顺序执行一系列特定的内存操作将触发UAF漏洞

- ✓ 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

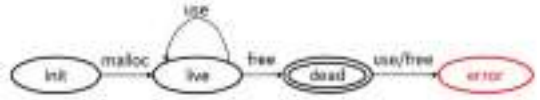
```
1  void main() {
2    char buf[7];
3    read(0, buf, 7)
4    char* ptr1 = malloc(8);
5    char* ptr2 = malloc(8);
6    if(buf[5] == 'e')
7      ptr2 = ptr1;
8    if(buf[3] == 's')
9      if(buf[1] == 'u')
10       free(ptr1);
11   if(buf[4] == 'e')
12     if(buf[2] == 'r')
13       if(buf[0] == 'f')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

基于轻量级静态分析与模糊测试的检测手段

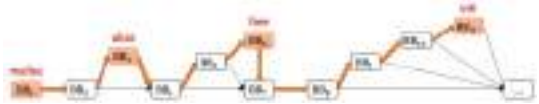
- 采用轻量级的程序静态分析技术识别违法了内存时序的安全使用规则的操作序列
- 使用操作序列导向的模糊测试技术逐步生成能出发内存时序漏洞的输入



整体流程



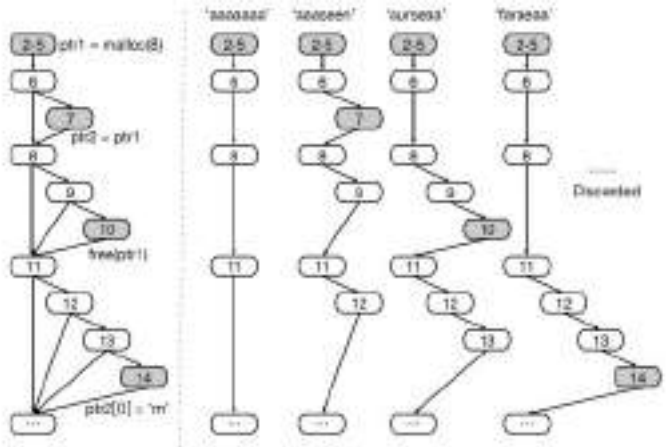
类型状态属性建模



静态分析识别潜在的操作序列

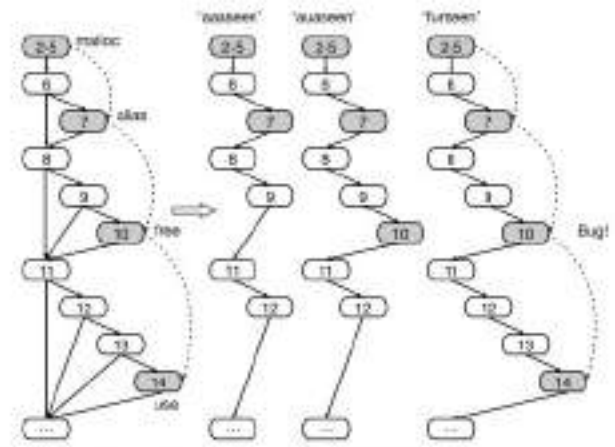
传统的代码覆盖导向的模糊测试技术

- 即便即便种子/测试用例覆盖了所有分支。仍然难以触发内存时序漏洞



操作序列导向的模糊测试技术

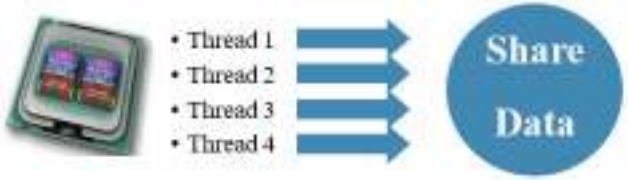
- 逐步覆盖操作序列，直到最终覆盖了完整的操作序列，触发内存时序漏洞



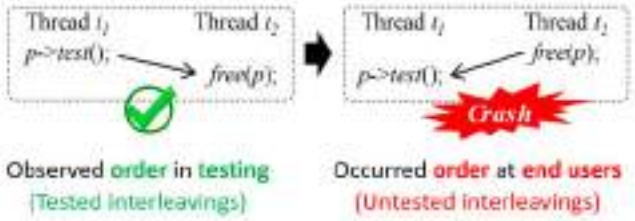
Haijun Wang et al. Typestate-Guided Fuzzer for Discovering Use-after-free Vulnerabilities, accepted by ICSE 2020

并发漏洞检测技术研究（ICSE 2022）

并发程序在现代多核系统中有着广泛的应用；并发编程更容易引入缺陷



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。

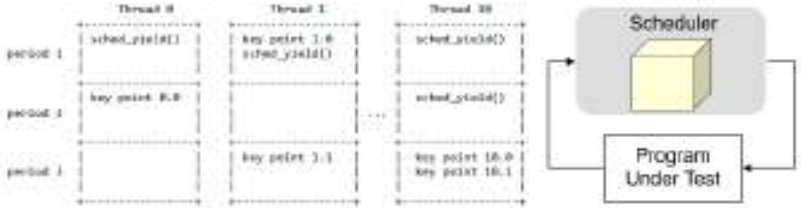


问题挑战：

- 并发程序执行过程中的线程交错具有不确定性，需测试不同的线程交错情况以保证并发程序的正确性。
- 并发程序中的一些潜在的错误难以被发现，它们只有在特定的线程交错下才会触发。
- 即便测试过程中触发了并发错误，也难以再次复现。

周期性调度方案

将并发程序的执行划分为多个周期，每周期只执行部分指令；基于OS的任务调度主动控制线程交错。



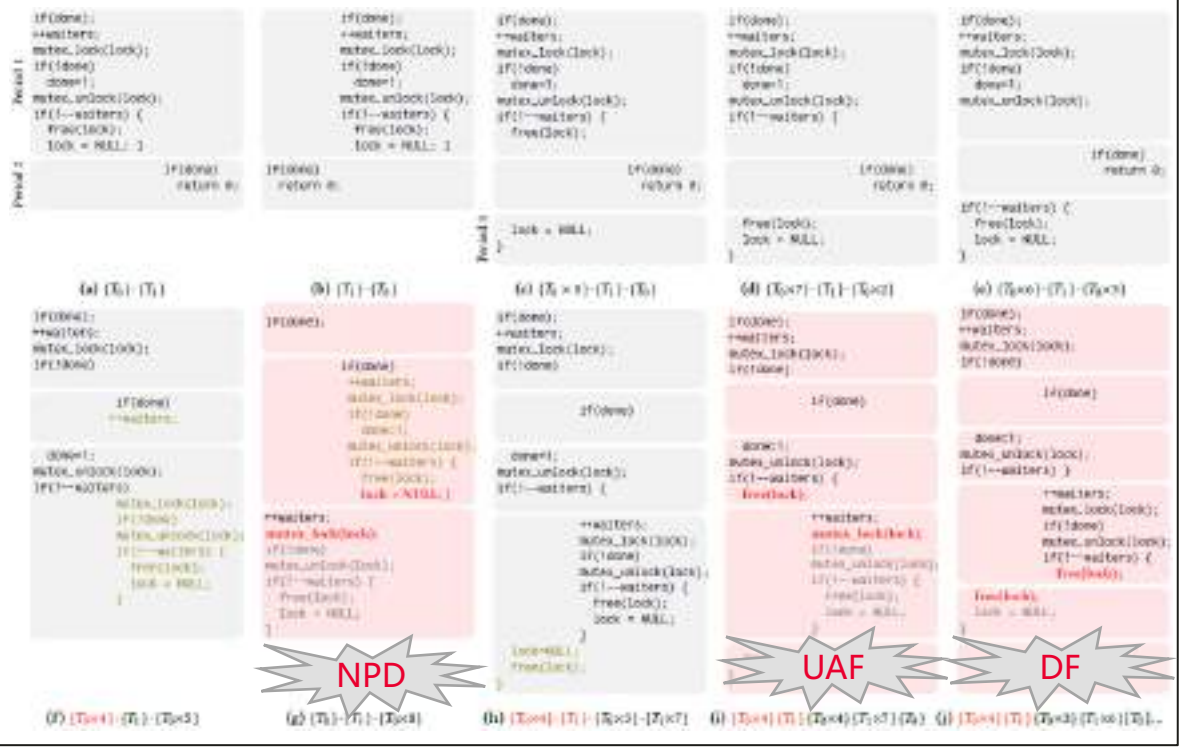
（有界）无状态模型检测

- 主动控制程序执行时线程交错情况
- 系统化地测试不同的线程交错情况
- 通过限定周期数上界和允许并发执行，减小需探索的线程交错空间

```
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 }
```

允许并发执行的调度

通过周期性调度主动控制部分语句的执行顺序，同时允许程序的其余部分并发执行。当线程数较多时，这种优化可以更有效地探索线程交错空间。



Cheng Wen et al. Controlled Concurrency Testing via Periodical Scheduling, accepted by ICSE 2022

开源软件漏洞挖掘实践（71 CVEs）

CVE（Common Vulnerabilities and Exposures）的全称是公共漏洞和暴露，是公开披露的网络安全漏洞列表。它是一个与信息安全相关的资料库，收集漏洞的详细信息以便于公众查阅。

我们在被广泛使用的开源软件中新发现并报告了上百个安全攸关的软件漏洞（71 CVEs）。

CVE-2022-26291, CVE2020-36375, CVE-2020-36374, CVE-2020-36373, CVE-2020-36372, CVE-2020-36371, CVE-2020-36370, CVE-2020-36369, CVE-2020-36368, CVE-2020-36367, CVE-2020-36366, CVE-2020-18900, CVE-2020-18899, CVE-2020-18898, CVE-2020-18897, CVE-2020-18395, CVE-2020-18392, CVE-2019-16166, CVE-2019-16165, CVE-2019-15140, CVE-2019-11471, CVE-2019-7704, CVE-2019-7703, CVE-2019-7702, CVE-2019-7701, CVE-2019-7700, CVE-2019-7699, CVE-2019-7698, CVE-2019-7697, CVE-2019-7665, CVE-2019-7664, CVE-2019-7663, CVE-2019-7662, CVE-2019-7154, CVE-2019-7153, CVE-2019-7152, CVE-2019-7151, CVE-2019-7150, CVE-2019-7149, CVE-2019-7148, CVE-2019-7147, CVE-2019-7146, CVE-2019-6293, CVE-2019-6292, CVE-2019-6291, CVE-2019-6290, CVE-2018-20712, CVE-2018-20657, CVE-2018-20652, CVE-2018-20651, CVE-2018-20593, CVE-2018-20592, CVE-2018-20591, CVE-2018-20002, CVE-2018-18701, CVE-2018-18700, CVE-2018-18607, CVE-2018-18606, CVE-2018-18605, …

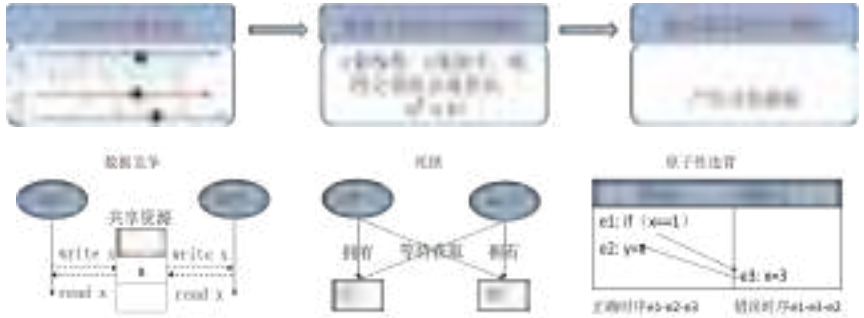
企业合作：并发程序通用安全性分析（华为）

产品线代码规模大（亿级）且逻辑复杂，由于多线程并发导致的数据竞争、空指针解引用等问题频发，传统可靠性测试很难发现此类问题，需要更加有效的检测手段。

分布式系统的消息时序处理逻辑复杂，如路由器组件频繁接收和处理消息，而消息处理在业务上有前后时序的依赖关系，当前缺乏分布式消息时序问题的检测手段。

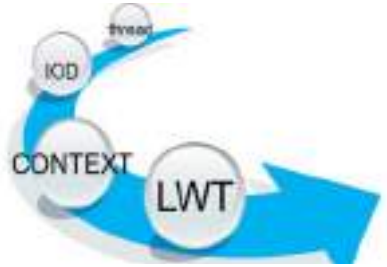
技术难题

多线程场景



分布式场景

工程难题



部分产品线会采用自研的并发库（如数存LWT），而现有的工具基于通用Pthread库，无法用于使用自研并发库程序的分析与测试。

公司部分产品使用较低版本的gcc编译器，导致业界许多基于LLVM的分析工具无法使用。

公司部分产品基于Arm架构，大多数并发问题检测工具只支持x86架构，支持Arm架构的工具较少。

大部分嵌入式产品需要在真实环境上板测试，对检测工具的性能开销有较高要求，现有业界开源的工具很难达到性能要求。

企业合作：P4 – 并发程序通用安全性分析平台（华为）

P4是一个面向并发程序的通用安全性分析平台，它融合了程序静态分析、受控并发测试和预测分析等技术，强调自动化、可扩展性和可用性，能对由于线程并发交错引起的内存安全（*e.g.*, use-after-free, double-free, null-pointer-dereference）、数据竞争、死锁等问题完成自动式识别。

并发问题频发的场景

- 存储LWT轻量级线程
- 路由器异步消息时序
- 无线嵌入式多线程程序
- …

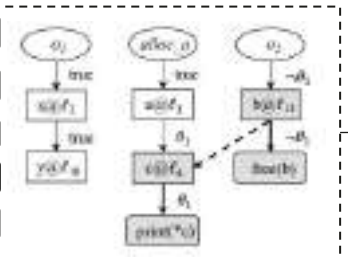
源代码

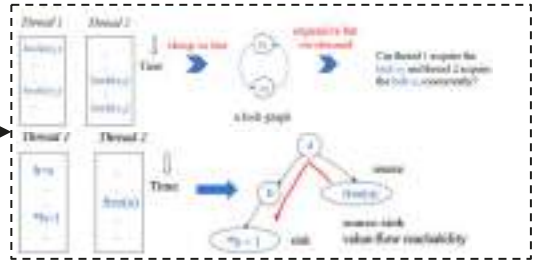
插桩/编译

关键操作

静态分析：不执行程序的情况下对并发程序的代码语义和行为进行分析

- Call Graph
- Value Flow Graph
- HB relation
- Lockset
- Visible Operation

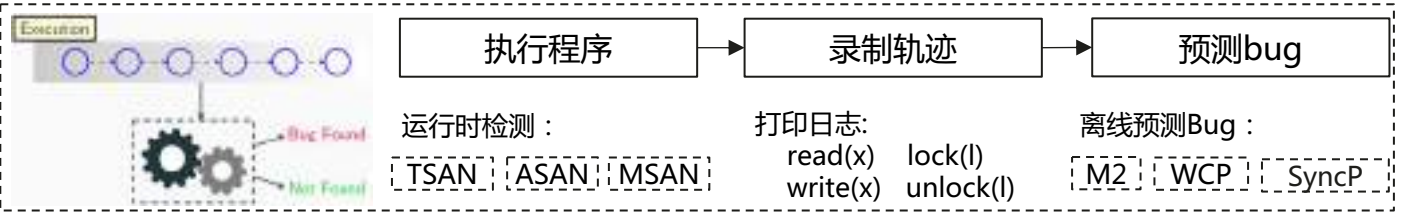




受控并发测试：主动控制线程调度，随机搜索/显式枚举可能的线程交错情况

偏序关系约减

预测分析：对于程序的动态执行轨迹，基于事件的偏序关系搜索并发缺陷



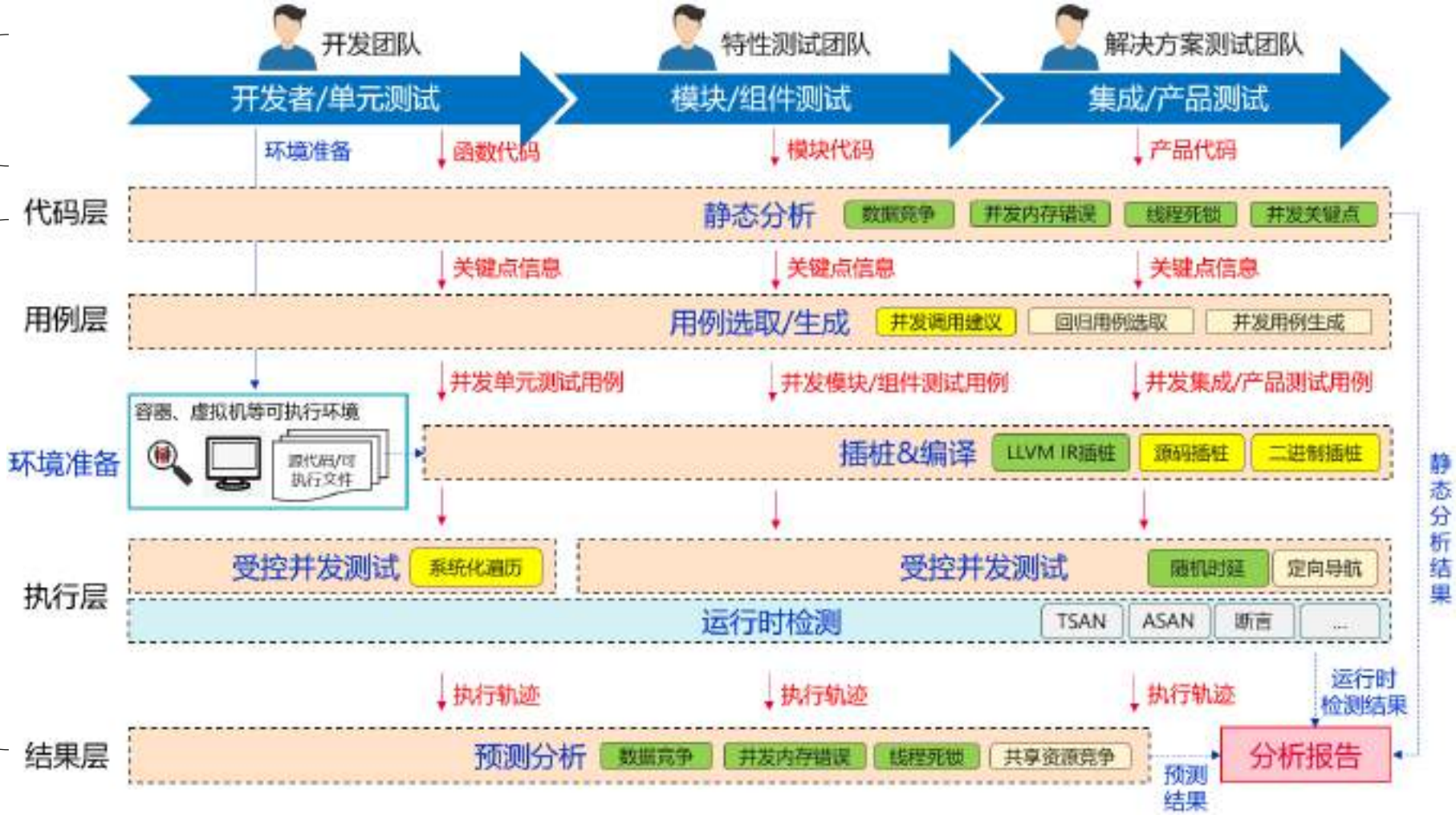
提供不同的执行轨迹

企业合作：P4 – 并发程序通用安全性分析平台（华为）

目标：突破多线程并发与分布式时序问题的程序分析检测技术，构建静态分析、用例选取/生成、受控并发测试、预测分析等技术相结合的并发程序安全性分析框架，面向开发与测试团队建立多层防线、覆盖各个测试环节，通用并发问题的发现和复现的效率提升5倍以上。

- 覆盖**从开发者测试到产品测试**的各个环节
- 在**代码层、用例层、执行层、结果层**建立多道防线
- 面向开发团队与测试团队提供不同的受控并发测试策略。

构建基于**静态分析、用例选取/生成、受控并发测试、预测分析**技术的并发程序动静态分析框架，对由并发交错引起的内存安全、数据竞争、死锁等问题进行自动识别，四条技术路线各有优势，既可独立工作，也可相互协作。



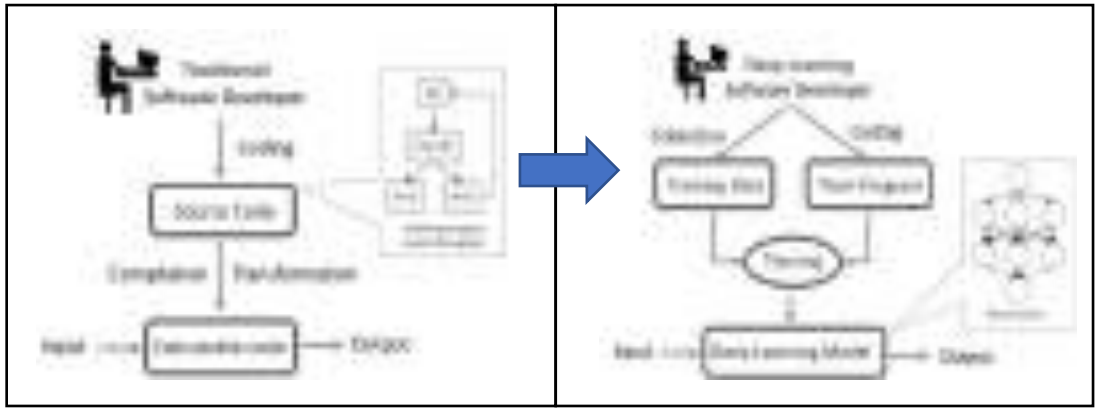
目　录

CONTENTS

软件缺陷分析与质量保障：趋势与展望

- 软件定义一切的时代下系统和场景越来越复杂多样，对安全和可信保障的要求越来越高，需要进行软件缺陷分析与质量保障的场景也越来越多
- 软件缺陷分析与质量保障的研究在持续推进：更先进的分析技术、更高效实用的分析工具、更多的应用场景、多技术融合等
- 可以考虑将静态分析、动态测试、形式化验证、人工智能等技术有效结合起来，针对不同场景使用不同技术的组合拳

新兴领域的机遇与挑战：随着人工智能、异构计算、量子计算的兴起，新应用与新计算范式也来带来了新的安全问题，软件缺陷分析与质量保障技术也面临了新的机遇与挑战。



神经网络等人工智能技术开启了软件2.0的新时代。软件不再是完全由人编写，而是能够在某种程度上“编写自己”

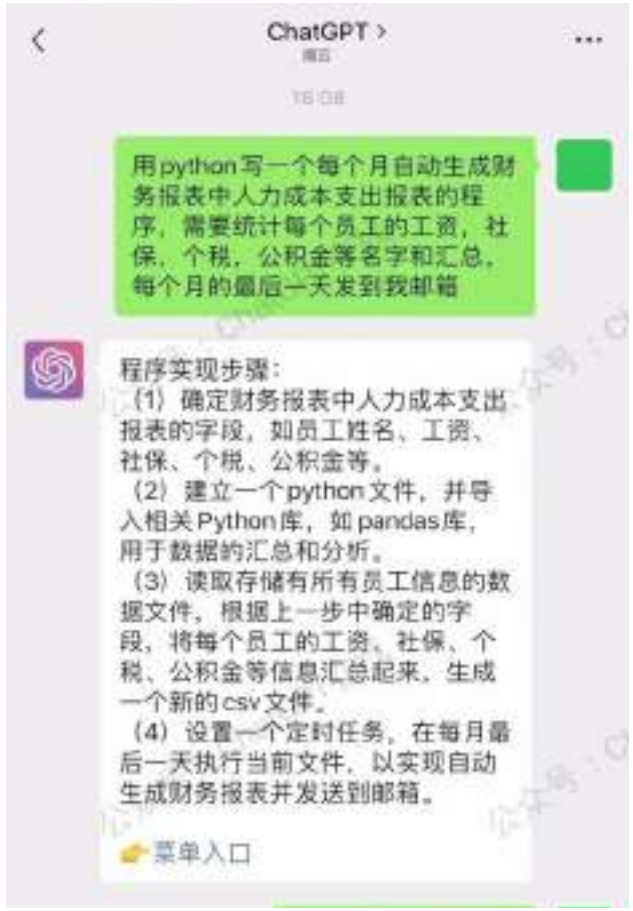
常见异构计算平台：
- CUDA： CPU+GPU
- OpenCL： CPU+GPU+FPGA

应用方向：
云计算、边缘计算、终端超级计算机（如自动驾驶）

现在的计算架构不再单纯用到CPU，更涉及多种异构架构；不同架构的软件质量分析需求正不断提高

如何发现AI写的代码中的缺陷/漏洞？

让AI写程序（例如ChatGPT）



此类AI 系统在人类难以提取规则又数据量足够的领域，如计算机视觉、语音识别、语音合成、自动驾驶等，数据驱动正在逐渐占据统治地位。

谢 谢！

如果您对软件分析与测试、软件工程、程序设计语言和形式化方法感兴趣，或者想进一步了解课题组的研究工作，可以与我们联系：wencheng@xidian.edu.cn