计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220883
60(2): 262−273, 2023

模糊测试中的静态插桩技术

王明哲　姜　宇　孙家广
（清华大学软件学院　北京　100084）
（wmzhere@gmail.com）

Static Instrumentation Techniques in Fuzzing Testing

Wang Mingzhe, Jiang Yu, and Sun Jiaguang
（*School of Software*, *Tsinghua University*, *Beijing* 100084）

Abstract　Fuzzing testing is a well-established method for detecting software defects. Its basic idea is generating a large number of random inputs to explore the program behavior extensively and then to monitor the crashes and reveal the software defects behind the crashes. Obviously, purely random inputs cannot explore program behavior efficiently and a large number of program defects can hardly lead to crashes. To further enhance the effectiveness of fuzzing testing, static instrumentation techniques are often introduced in fuzzing testing to speed up the exploration of the program state space and improve the ability of defect detection. As a result, using static instrumentation has become a de facto practice in fuzzing testing nowadays. In this paper, we focus on the instrumentation requirements under the background of fuzzing testing. Besides introducing the basics of static instrumentation, we systematically analyze the typical schemes of static instrumentation from two perspectives, i.e., security hardening and guidance collection. In addition, we investigate the challenge of execution overhead. Specifically, for a comprehensive set of instrumentation schemes, we measure the execution speed of the instrumented program and compare it to non-instrumented programs of the baseline. Finally, based on the above analyses and measurements, we provide a primitive analysis over the optimization directions of static instrumentation.

Key words　static instrumentation; fuzzing testing; software defects; program analysis; overhead

摘　要　模糊测试是一种行之有效的软件缺陷检测方法.其基本思想是生成大量随机输入,从而广泛探索程序行为,并以此发现程序崩溃和崩溃背后的软件缺陷.显然,纯随机的输入无法高效探索程序行为,大量程序缺陷也难以导致崩溃.为了进一步提升模糊测试的有效性,模糊测试往往引入静态插桩技术,用于加快探索程序状态空间速度,提升发现缺陷的能力.因此,引入静态插桩已经成为当下模糊测试的经典实践.聚焦于模糊测试场景下的插桩需求,除了介绍静态插桩技术的基本原理外,从安全特性强化和导向信息收集两个视角出发,系统性地分析了当下静态插桩的典型方法.同时,针对插桩的额外开销问题,全面地测量了不同插桩方案下的程序的执行速度,并与基线的未插桩程序进行比对.最后基于上述分析和测量,初步展望了静态插桩的优化方向.

关键词　静态插桩;模糊测试;软件缺陷;程序分析;额外开销

中图法分类号　TP311

收稿日期:2022−10−21;**修回日期**:2022−12−13

基金项目:国家重点研发计划项目(2022YFB3104000);国家自然科学基金项目(62022046, 92167101, U1911401);微众学者计划(20212001829)
This work was supported by the National Key Research and Development Program of China (2022YFB3104000), the National Natural Science Foundation of China (62022046, 92167101, U1911401), and Webank Scholar Project (20212001829).

通信作者:姜宇(jy1989@mail.tsinghua.edu.cn)

大量系统软件侧重于性能，往往使用C/C++等具有底层控制能力的程序设计语言编写.然而，这些程序设计语言若存在不规范的使用，会产生数组越界、数据竞争等不安全行为，导致软件缺陷.部分缺陷会进而引发安全问题，产生任意代码执行等重大安全影响.因此，有必要识别和修复这些缺陷.

模糊测试是一种自动化的软件缺陷发现方法.不同于传统软件测试依靠人编写测试用例，模糊测试自动生成输入并以此执行程序.测试工具在程序运行时监控崩溃或超时等异常行为，从而发现异常行为背后的软件缺陷.由于构造的输入内容具有高度的随机性，能够触发开发人员难以想象的边界条件，这些条件包含了大量容易触发异常的场景.此外，自动的输入生成允许模糊测试进行动辄数千万次的测试，广泛探索程序的状态空间.

在实际的模糊测试实践中，使用静态插桩技术作为辅助已成为事实上的标准.静态插桩技术包含2个阶段；1)在源程序编译的过程中，编译器在程序的关键位点插入额外的逻辑(即桩点)；2)在程序运行时桩点被触发，从而收集执行的动态信息或改变程序的行为.在模糊测试的场景下，静态插桩技术主要用于安全特性强化和导向信息收集.该技术能够显著提升软件缺陷的发现能力，加快程序状态空间的探索速度.

安全特性强化插桩旨在提升软件缺陷的发现能力.模糊测试检测软件缺陷的方法主要依赖程序崩溃等操作系统的信号机制.然而，程序中出现的不安全行为仅有少部分会触发硬件异常(例如内存保护、除零错误)且被操作系统捕获.大部分类型的缺陷，例如绝大多数的单字节缓冲区溢出和有符号数的算术溢出，均不会触发硬件的保护机制.为了解决这些问题，静态插桩在潜在的风险位点进行额外的安全检查，从而在发生问题时通知模糊测试工具.

导向信息收集插桩旨在加快程序状态空间的探索速度.模糊测试生成的输入是随机的，因此存在难以有效探索程序的状态空间的问题，且该问题在输入高度结构化的程序和具有大量逻辑的复杂系统上尤为显著.为了解决此问题，有必要引入导向信息，使得模糊测试工具侧重于探索具有更高价值的程序状态，同时高效求解复杂约束.例如，动态测试技术发现软件缺陷的前提是触发缺陷所在的代码，所以，可以引入代码覆盖，从而将模糊测试导向尚未触发的代码位点.

此外，静态插桩引入了额外的逻辑.在提升模糊测试效果的同时，静态插桩也不可避免地降低了程序的执行速度，从而影响了模糊测试的整体吞吐量.一般地，插桩的能力越强，桩点数量越多，逻辑越复杂，带来的额外开销越大.由于模糊测试的整体效果依赖于海量执行对程序状态空间的探索，在模糊测试中使用静态插桩时，应合理取舍插桩能力和测试吞吐量.

目前，静态插桩研究主要有2类，一是提出了新的插桩特性，二是降低了已有方法的额外开销.这些工作尽管十分有效，但都局限于单一的插桩策略，缺乏对功能和性能的系统性分析和梳理.为此，本文面向模糊测试场景下静态插桩技术，从功能和性能2方面入手，系统性分析了典型的插桩技术，测量了它们的额外开销，并由此展望了插桩技术的发展趋势.

本文的主要贡献有3方面：

1)系统性地分析了静态插桩技术.对于安全特性强化插桩，本文从程序语言规范的角度说明了缺陷的根源，并详细分析了检测技术的核心思想.对于导向信息收集插桩，本文总结了编译期的桩点部署策略，以及运行时对导向信息的利用策略.

2)测量了插桩的额外开销.本文选取了有代表性的静态插桩方案，使用典型的模糊测试项目统计了这些插桩方案的额外开销.其中，选取的插桩方案涵盖了控制流特征采集、比较操作数收集、安全特性强化3个最常见的插桩需求.

3)分析了静态插桩技术的优势、局限和优化方向.基于分析和测量，本文结合学术界的最新工作，讨论了模糊测试场景下静态插桩技术的优化方向.

1 研究背景

1.1 模糊测试

模糊测试起源于20世纪90年代，其最初目的是测试UNIX实用程序[1].在此工作中，模糊测试工具Fuzz生成随机输入，接着将输入通过标准输入传递给目标程序，或通过终端仿真程序ptygig传递给交互式的目标程序，与此同时，测试工具等待程序的异常信号，生成输入、执行、等待异常的过程被脚本不断重复，从而发现软件缺陷.上述过程中，测试工具无需侵入目标程序，因此被称为黑盒模糊测试.

2008年Godefroid等人[2]提出了白盒模糊测试.之所以被称为白盒模糊测试是因为测试工具深入观察目标程序内部，从而系统性地探索程序的状态空间.具体地，模糊测试工具SAGE使用符号执行技术，

首先收集程序执行路径上的约束，接着对约束逐个取反并使用求解器生成新的输入，从而探索不同的分支. 包括 SAGE 在内的大量白盒测试工作尽管对程序有系统性的探索，但受制于约束收集和求解的性能，其发现软件缺陷的效果依然十分有限.

2014 年提出的 AFL 是首个灰盒模糊测试工具[3]. 不同于高开销的白盒测试，灰盒模糊测试工具 AFL 使用轻量级插桩技术，统计程序的控制流特征并以此作为导向信息. 具体地，程序执行轨迹的时空特征被映射到一个 64KiB 的位图上. 通过分析位图背后隐含的控制流特征，模糊测试工具能够识别出尚未发现的全新程序行为，从而继续深入探索. 基于导向信息的灰盒模糊测试十分成功，发现了大量的软件缺陷和安全漏洞. 因此，灰盒模糊测试获得了包括谷歌[4]、微软[5]、Adobe[6] 在内的工业界的广泛应用.

1.2 静态插桩和模糊测试

在实际的模糊测试实践中，静态插桩已成为事实上的标准[7]. 例如，OSS-Fuzz[4] 是谷歌发起的模糊测试持续集成服务，囊括了大量被广泛使用的基础软件项目. 其中，全部的测试项目都使用了至少一种静态插桩技术. 具体地，表 1 显示了 OSS-Fuzz 项目中启用模糊测试的 771 个测试项目中静态插桩的使用情况.

Table 1 Usage Statistics of Static Instrumentation in OSS-Fuzz

表 1 OSS-Fuzz 项目的静态插桩使用统计

插桩方案	类型	目标	数量
CmpLog	导向信息收集	收集比较的操作数	411
CompCov	导向信息收集	将整数比较拆分为多个比较	411
Ctx2	导向信息收集	收集上下文敏感的边覆盖	411
Sanitizer Coverage	导向信息收集	收集边覆盖率和其他信息	768
Address Sanitizer[8]	安全特性强化	检查内存安全问题	771
Undefined Sanitizer	安全特性强化	检查未定义行为	485
Memory Sanitizer[9]	安全特性强化	检查对未初始化内存的读取	188
Thread Sanitizer[10]	安全特性强化	检查线程安全问题	3

观察表 1 可以发现，当下模糊测试的静态插桩主要可以分为 2 个类型. 1)在导向信息收集方面，代码覆盖是模糊测试工具关注的核心信息. 除此之外，一半多的项目也关注对比较等约束的测试效率，使用了操作数收集等方法处理. 2)在安全特性强化的方面，全部的项目无一例外使用了 Address Sanitizer 进行内存安全检查. 大多数项目都检查了未定义行为，也有小部分项目检查了未初始化内存的访问. 只有 3 个项目检查了线程安全问题，这主要是因为 OSS-Fuzz 测试的软件多为基础库，其线程安全多由库的使用者维护.

1.3 插桩的基本类型

根据插桩的不同时机，插桩主要分为运行时进行的动态插桩和运行前完成的静态插桩. 动态插桩在程序运行中进行，一般对内存中的机器码进行补丁；静态插桩在程序运行前进行，将原始程序插桩并生成修改后的二进制文件用于后期执行.

根据插桩能力的不同，动态插桩一般分为轻量级插桩和重量级插桩 2 种类型. 轻量级插桩具有较低的复杂度和执行开销. 例如，LLVM[11] 中的 XRay 在程序开头注入额外的空指令. 在执行时，插桩逻辑可以将空指令替换为跳转指令，从而重定向函数的控制流并执行额外的插桩逻辑，若桩点不再需要，则可以将跳转指令回退为空指令. 然而，轻量级插桩局限于对单个指令的补丁，无法提供强大的分析能力，也无法对程序逻辑进行复杂的修改. 因此，在要求更强的插桩能力时，人们往往使用重量级插桩. 例如，DynInst[12] 对二进制程序进行复杂而全面的分析，将其抽象成从模块、函数到基本块和指令的完整体系. 在程序运行时可以灵活地对体系内的对象进行高层次分析和修改，实现内存安全检查等复杂的插桩需求.

由于动态插桩的对象一般是机器码，其蕴含的语义信息远少于源码和编译器内部表示，因此难以满足模糊测试的需求. 例如，内存安全检查工具为了检测栈上的缓冲区越界，需要获得栈上各个对象的内存布局. 然而，上述信息在源码递降到机器码的过程中已经消失了. 此外，在机器码上进行直接补丁的性能也弱于包含复杂优化的编译器，一般会带来高达数十倍的额外开销[13]，大大降低了模糊测试的吞吐量. 因此，当下的模糊测试一般使用基于编译器的静态插桩.

基于编译器的静态插桩一般以程序的源码或编译器内部的中间表示作为输入. 以编译器 Clang 为例：源码首先被 Clang 前端变换为 LLVM 中间表示，该中间表示经由 LLVM 中端，完成清理、归一化和优化；接着被 LLVM 的指令选择组件变换为机器中间表示，机器中间表示被 LLVM 后端进一步完成平台相关的优化和递降；最后 AsmPrinter 组件将该中间表示由 MC 层生成 ELF 等平台相关的对象文件. 在此过程中的各个环节都可以插桩. 例如，可以在 Clang 前端结合源码的语义信息，对程序设计语言内部的未定义

行为进行捕获和检测；可以在 LLVM 中端匹配和内存安全相关的指令（例如内存读写）并插入检测器，从而实现通用的内存安全强化.

2 安全特性强化插桩

2.1 内存安全检查

手动内存管理是系统程序设计中极其容易出错的环节. 在系统程序中，对内存的申请、使用和释放操作均可能带来内存安全问题. 本节列举 3 个经典的内存安全问题.

1）内存申请. 若申请的内存大小受攻击者直接控制，攻击者可以构造畸形的内存申请请求，从而使程序耗尽资源.

2）内存使用. 若访问偏移量超过了内存对象的大小，则会造成经典的缓冲区溢出问题.攻击者可以依此泄漏程序中的数据，或覆盖栈帧所保存的函数返回地址，造成任意代码执行.

3）内存释放. 若释放过早（或称为释放后使用，use-after-free）或释放了非 malloc 返回的对象，则可能使多个内存对象重叠，使得攻击者任意控制内存对象的值. 若忘记释放，则会造成内存泄漏，攻击者可以耗尽程序资源造成拒绝服务攻击.

这 3 个安全问题时常不能造成程序崩溃，因此不能被模糊测试工具所发现. 不能有效检测内存安全的另一个重要的原因是硬件的局限. 例如，x86 的微处理器中，内存管理单元支持的最小粒度为 4KiB. 若越界的地址恰好在当前页内，则无法触发硬件内存保护. 此外，由于系统软件注重执行效率，内存分配策略也主要侧重于降低开销而非增强安全性. 例如，现代微处理器的指令集架构一般要求内存对齐，否则会造成访问下降或硬件异常. 举一个简单的例子：在 AArch64 下，只能以 8B 的整数倍的地址访问 64b 整数. 在构造栈布局时，编译器需要将变量对齐到合适的地址，对象间会留下空隙. 若非法访问的地址恰好在 2 个对象间的空隙内，则极难发现缓冲区越界的问题.

为了增强内存安全检查的有效性，引入静态插桩是一个可行的方案. 从本质上讲，静态插桩实现了类似于微处理器内存管理单元的功能，但该功能基于软件实现，在牺牲性能的情况下带来了更强的检测能力. 一方面，静态插桩技术允许监控程序中的每一次内存读写操作，从而检查非法的内存使用；另一方面，静态插桩可以替换程序的运行时库，从而检查对系统库中的内存申请和释放函数的调用. 类似于内存管理单元，静态插桩也需要维护内存对象的元信息. 下面以 Address Sanitizer[8] 为例，详细说明静态插桩是如何实现内存安全检查的.

为了高效地管理内存对象的元信息，Address Sanitizer 使用影子内存技术. 换言之，任何地址的元信息都会被 Address Sanitizer 映射到一个保留的区间. 具体地，地址 Addr 的元信息存储地址 *ShadowAddr* 为 *Addr* / *kScale* + *kOffset*.其中 *kScale* 和 *kOffset* 均为平台相关的常数：*kOffset* 为影子内存的起始地址，在 x86 平台下 *kOffset* 值取 0x20000000；*kScale* 为压缩率（即几个字节的空间共享同一个字节的元信息），例如在 x86 平台下 *kScale* 值取 8. 压缩率越高，存储元信息所需的额外内存越少，但验证内存访问合法性的算法复杂度也可能上升.

影子内存中的每个字节有 [−128, 127）的取值范围. 以 x86 平台为例，影子内存中的每个字节负责记录 8B 的程序内存的元信息. 对于每个影子内存中的字节，0 表示当前影子内存所对应的 8B 均可访问；负数表示当前影子内存所对应的 8B 均不可访问；[1, 8）表示当前影子内存所对应的内存区间的有效长度，例如 3 表示前 3B 均可访问. 精心设计的内存布局和元信息储存方式大大简化了检查代码. 假设当前内存访问满足内存对齐的要求，容易得出内存安全检查桩点逻辑. 检查逻辑极其简单，只涉及少量位运算（由除法运算化简得到）和算术运算、单次内存读取、单次比较和分支运算.下面的代码显示了安全检查桩点的具体逻辑.

```
/* 检查 8B 的内存访问 */
void Check8ByteAccess(uint64_t *Ptr) {
  int8_t (*ShadowPtr = TO_SHADOW(Ptr);
  assert (*ShadowPtr == 0, "非法内存访问");
}

/* 检查 1B, 2B, 4B 字节的内存访问 */
void CheckNByteAccess(int8_t *Ptr, unsigned N) {
  int8_t *ShadowPtr = TO_SHADOW(Ptr);
  /* 全部 8B 均可访问：直接放行.*/
  if (*ShadowPtr == 0) return;
  /* 计算地址相对于元信息所映射内存区间起始位
  置的偏移量.*/
  size_t OffsetInShadow = Ptr − (uintptr_t)Ptr) & 0x7;
  /* 计算访问最后字节的相对于映射内存区间起始
  的偏移量.*/
```

```
  size_t BytesAccessed = OffsetInShadow + N;
  /* 若*ShadowPtr 为负，说明当前整块内存均不可访问.*/
  /* 若*ShadowPtr 为正，则当前字节记录了允许访问的长度.*/
  assert (BytesAccessed ≤ *ShadowPtr, "非法内存访问");
}
```

上述代码只是利用影子内存中的元信息进行安全检查，但元信息自身还需要维护. Address Sanitizer 的基本维护策略是使用禁区和隔离区. 在每个内存对象前后，Address Sanitizer 添加禁区并将其标记为不可访问，从而使数组越界等空间异常被检查到. 在堆中的内存对象被释放后，Address Sanitizer 将对象加入隔离区，延缓对该地址的重用，从而提升释放后使用等时间异常被检测到的概率. 为了使禁区和隔离区能够匹配当前的内存状态，Address Sanitizer 对内存对象的申请和释放进行插桩：

1)栈对象. 对于当前栈帧中存储的自动变量等类型的内存对象，Address Sanitizer 修改了内存布局从而在对象间插入禁区；此外，还在函数的入口将禁区标记为不可访问，在函数退出时删除这些标记防止误报.

2)堆对象. Address Sanitizer 将 *malloc* 和 *free* 等内存管理函数重新实现，从而在对象间插入禁区，在内存释放后置入隔离区. 同时，内存申请的上下文也被记录，用于生成开发者友好的错误报告.

3)静态对象. 对于全局变量、静态变量等类型的静态对象，编译器将这些数据写入到二进制文件中. 程序运行时，操作系统的动态链接器在程序运行前将这些数据映射回内存，对象的生命周期是整个进程. 除了在对象间插入禁区外，为了维护静态对象的元信息，Address Sanitizer 使用构造函数完成对禁区的标记. 具体地，Address Sanitizer 在编译期扫描全局变量，并将它们的源码位置、长度等信息写入到二进制的一个数组中. 同时，Address Sanitizer 生成构造函数 *asan.module_ctor*. 该函数被标记为构造函数，在主程序运行前获得控制权. 函数将元信息传递给运行时库所暴露的接口 *__asan_register_globals*，从而注册全局变量.

此外，Address Sanitizer 还截获了 *memcpy*、*fread* 等系统库函数，从而对常用的外部函数进行内存访问建模和检查.

2.2 未定义行为检查

ISO C11 标准中写到，“在使用不可移植的、错误的程序结构或错误的数据时，本国际标准对其没有规定要求”[14].例如，有符号数算术运算时发生的溢出就属于未定义行为，内存安全问题也是未定义行为的子集. 引入未定义行为，不光能简化编译器设计实现，还能带来额外的优化空间.

1)简化编译器设计实现. 例如，在循环等性能攸关的场景下，编译器往往需要分析 2 个指针是否可能指向相同的内存对象. 根据语言规范，float 指针和 int 指针不能指向同一处内存(即不存在别名)，否则为未定义行为. 基于此规范，编译器能够通过类型信息，在部分场景下直接确认不同指针不会互为别名，从而简化编译器的设计.

2)带来额外的优化空间. 例如，有符号数的算术运算在主流平台下均以 2 的补码形式进行. 因此，若不发生溢出，$a+1>a$ 在任何情况下均成立；若规定有符号数的算术运算溢出是未定义行为，编译器可以不考虑罕见的极端情况，从而将 $a+1>a$ 折叠成常量 true.

为了检查未定义行为，可以引入静态插桩. C/C++程序设计语言规范中规定了大量的未定义行为，例如 C11 标准的附录 J 中就列出了近 200 条未定义行为. 相比于只须处理内存相关原语的内存安全检查 Address Sanitizer，更广义的未定义行为检查 Undefined Behavior Sanitizer 需要紧密结合程序语义. 因此，Address Sanitizer 只须作为 LLVM 中端的一部分，而 Undefined Behavior Sanitizer 需要在更靠近源码的编译器前端 Clang 中进行. 具体地，源码在 Clang 的词法分析、语法分析和语义检查后，被变换为合法的抽象语法树；Clang 的代码生成(CodeGen)组件扫描语法树，在生成程序 LLVM 中间表示的同时，根据插桩选项生成额外的未定义行为检查代码. 为此，Undefined Behavior Sanitizer 需要紧密结合程序语义，并进行大量的适配.

下面的代码列出了返回有符号整数 $x+y$ 的值的函数 *foo* 以及安全强化版本 *foo_ubsan* 的 LLVM 中间表示. 该代码进行了少量编辑，从而删除无关细节. 不难发现，Clang 在代码生成时，*foo* 函数的加法指令是 add nsw，其中 nsw 表示不存在有符号计算的溢出(no signed wrap). 而 *foo_ubsan* 函数使用了 LLVM 的内部函数 *llvm.sadd.with.overflow*. 该函数返回 2 个值，一个是加法的结果 i32，另一个是布尔值 i1，它表示是否存在溢出. 若发生溢出，则跳转到函数 *handler.add_overflow*. 该函数调用运行时库的接口 *__ubsan_handle_add_overflow* 报告错误，接着跳转到 cont 恢复执行.

```
define i32 @foo(i32 %x, i32 %y) {
    %add = add nsw i32 %x, %y
    ret i32 %add
}
define @foo_ubsan(i32 %x, i32 %y) {
entry:
    %i = tail call { i32, i1 }
    @llvm.sadd.with.overflow.i32(i32 %x, i32 %y)
    %i1 = extractvalue { i32, i1 } %i, 1
    br i1 %i1, label %add_overflow, label %cont
add_overflow:
    tail call void @__ubsan_handle_add_overflow(...)
    br label %cont
cont:
    %i4 = extractvalue { i32, i1 } %i, 0
    ret i32 %i4
}
```

3 导向信息收集插桩

3.1 控制流特征采集

当下几乎所有的主流模糊测试工具均采用控制流特征作为导向信息. 其中, 覆盖位图是最常使用的控制流特征. 覆盖位图的核心思想是压缩程序执行的时空特征到固定大小的计数器数组. 首先, 数组的不同元素表示不同的控制流转移, 整体上反映了整体控制流转移的空间特征. 例如, 对于最简单的基本块覆盖而言, 可以将每一个基本块都分配独立的计数器并连续存放, 构成数组. 其次, 数组内元素的值和当前控制流转移的触发次数相关, 反映了单个控制流转移的时间特征. 例如, 假设每个基本块计数器的取值范围在 0~255 之间, 那么可以将计数器的值分为 8 类: 1, 2, 3, [4, 8), [8, 16), [16, 32), [32, 128), [128, 256). 在分类下, 每个计数器均映射到 8 个特征. 在某次执行中, 若模糊测试工具发现了先前从未发现过的特征, 则认为当前执行的输入较为新颖, 需要保存以供后续探索; 否则, 认为当前执行没有意义, 可以直接丢弃.

下面的代码总结了模糊测试工具 AFL[3] 的覆盖率处理算法. 函数 *checkCoverage* 接受数组长度 *N*、覆盖位图 *C* 和未知覆盖位图 *U*. *checkCoverage* 的第 1 个循环利用预先编制的查找表 *TO_BITMAP*, 将每个计数器的值映射到 8 个不同的特征中, 并以位图的形式存放. 换言之, 当计数器为零时, 转换后的位图也为 0, 说明没有任何特征; 当计数器为非零时, 转换后的位图的 8 位中有且只有一个位被置为 1, 置位的位置取决于分类结果. *checkCoverage* 的第 2 个循环扫描当前覆盖和全局的未知覆盖位图. 若当前覆盖的特征在全局的未知覆盖特征中出现, 说明当前执行是有价值的, 需要判断执行的价值等级. 若当前计数器是首次发现(即未知覆盖全部被置位), 说明当前执行发现了新覆盖, 将返回值设置为 2; 否则, 说明当前执行发现了已经覆盖计数器的不同计数模式, 将返回值设置为 1, 表示发现了新路径.

```
int checkCoverage(size_t N, uint8_t * C, uint8_t * U) {
    int R = 0;
    /* 将计数值转换为位图. */
    for (size_t I = 0; I < N; ++I)
      C[I] = TO_BITMAP[C[I]];
    /* 检查当前覆盖和全局的未知覆盖位图. */
    for (size_t I = 0; I < N; ++I) {
      /*确保当前位图的特征出现于未知位图中.*/
     if (C[I] & U[I] == 0)
      continue;
      /* 计算返回值类型.*/
     R = max(R, U[I] == 0xff?2 : 1);
      /* 删除未知位图中的对应特征. */
     U[I] &= ~C[I];
    }
    return R;
}
```

为了收集覆盖位图, 模糊测试工具往往使用静态插桩. 下面以 x86-64 平台下的 afl-gcc 为例, 说明覆盖率的收集方式. 在构建待测程序时, afl-gcc 包裹了系统编译器 gcc, 从而执行额外的操作. 若当前 gcc 调用的目标是编译源文件, afl-gcc 编辑对系统编译器 gcc 的调用参数, 使得 gcc 在生成对象文件时使用 afl-as 作为汇编器. afl-as 是带有插桩功能的汇编器, 它接受 gcc 生成的汇编文件进行插桩, 之后交给系统汇编器 as 生成对象文件. afl-as 对汇编文件进行模式匹配, 识别基本块的起始标签, 并插入桩点代码. 修改后的汇编文件调用系统的原始汇编器 as 生成对象文件. 桩点代码的主要逻辑是创建栈帧, 备份寄存器, 接着跳转到 *__afl_maybe_log* 函数中. 其中, *COMPILE_RANDOM* 是 afl-as 生成的随机数, 被用作当前基本块的标号. 具体的桩点汇编代码为:

```
    ; 开辟栈帧并备份寄存器
    leaq -(128+24)(%rsp), %rsp
```

```
movq %rdx, 0(%rsp)
movq %rcx, 8(%rsp)
movq %rax, 16(%rsp)
; 将 rcx 寄存器设置为一个编译期的随机数
movq $COMPILE_RANDOM, %rcx
; 调用桩函数
call __afl_maybe_log
; 恢复寄存器并移除栈帧
movq 16(%rsp), %rax
movq 8(%rsp), %rcx
movq 0(%rsp), %>rdx
leaq (128+24)(%rsp), %rsp
```

核心函数*__afl_maybe_log*完全由汇编写成，在此给出它的伪代码以便读者理解. *__afl_maybe_log*首先备份上下文 save_flags，然后判断存储于共享内存的覆盖位图是否完成初始化. 若是首次执行桩点，没有完成覆盖位图的初始化，则尝试使用 shmat 函数加载模糊测试工具所分享的共享内存；接着是核心的控制流到位图的映射环节，该环节计算了当前控制流所对应的计数器索引 *index*，并维护了上一基本块标号*__afl_prev_loc*；最后*__afl_prev_loc*在退出前恢复了保存的上下文. 不难发现，该映射本质上将上一基本块标号 *prev* 和当前基本块标号 *curr* 映射到 (*prev* >> 1) ^ *curr* 的计数器上，抽象了控制流的空间特征.对时空特征进行抽象的代码如下：

```
void __afl_maybe_log(int id) {
 save_flags ( );
 if (__afl_area_ptr == NULL) {
     获得共享内存，存入 __afl_area_ptr;
 }
 int index = __afl_prev_loc ^ id;
 __afl_prev_loc = id >> 1;
 __afl_area_ptr[index] += 1;
 restore_flags( );
}
```

3.2 比较操作数收集

仅收集控制流特征不能高效地探索程序的状态空间. 这是因为控制流特征只能反馈是否通过了某个复杂约束，却不能辅助对复杂约束的求解. 以颜色管理库项目 LCMS 为例，其输入是国际色彩联盟提出的 ICC 描述文件，该文件以 ASCII 字符串 acsp 作为开头. LCMS 在执行之始就验证输入文件是否为有效的 ICC 描述文件，且代码为：

```
#define cmsMagicNumber 0x61637370 // 'acsp'
if (_cmsAdjustEndianess32(Header.magic)≠
  cmsMagicNumber) {
 cmsSignalError(...);
 return FALSE;
}
```

代码读入输入的前 4 个字节到 *Header.magic* 中，并转换到大端序，判断转换后的值是否和 *cmsMagicNumber* 相等，若不相同则调用 *cmsSignalError* 函数报错. 显然，测试程序实际逻辑的前提是构造有效的文件头. 然而，若只收集程序控制流特征，则几乎不可能构造有效的文件头. 这是因为程序的控制流特征只能反馈该约束是否通过，不能在约束尚未解决前提供任何的导向信息. 在缺乏导向信息的情况下，模糊测试工具只能随机地生成输入，而恰好能构造原始输入的概率低至 2^{-32}.因此，对复杂约束进行求解来提供导向信息对模糊测试意义重大.

整数比较是程序的一类主要复杂约束. 除了 LCMS 的例子中 if-else 的例子外，整数比较还包括 switch-case 语句. 以 Clang 内置的 SanitizerCoverage 为例，SanitizerCoverage 首先扫描程序中的所有指令，若发现表示整数比较的 icmp 指令，则执行控制流分析，从而跳过执行频次高且无意义的循环控制条件；接着判断比较操作数的长度是否为常量，以此选择合适的回调函数，并最终创建对回调函数的调用. 若发现表示分支选择的 switch 指令，则为其创建常量数组，从而存储数据类型、选择数量等元信息以及 case 中的值；最后该常量数组和当前 switch 值会被作为参数调用回调函数.

下面的代码列出了典型的回调函数原型. *trace_cmp1* 是对两端是变量的单字节比较的回调. *trace_const_cmp*4 是对一端是常量的 4B 比较的回调，其中参数 *const* 是常量，*variable* 是变量. *trace_swith* 是对 switch 的回调，其中参数 val 是转换为 uint64_t 的条件变量，参数 *cases* 指向元信息. 通过实现下列函数，模糊测试工具可以收集并利用执行过程中的比较信息.

```
void __sanitizer_cov_trace_cmp1 (uint8_t Arg1,
 uint8_t Arg2);
void __sanitizer_cov_trace_const_cmp4 (uint32_t
Const,
 uint32_t Variable);
void __sanitizer_cov_trace_switch (uint64_t> Val,
 uint64_t * Cases);
```

除整数比较外，缓冲区比较也是程序中的一类主要约束. 这些比较大多由 C 标准库提供，例如

strcmp 等字符串比较函数，或 *memcmp* 等缓冲区比较函数. 此外，C++标准库的部分函数，例如 GNU libstdc++内部的 std::__cxx11::basic_string::compare 也可能被 STL 头文件所引用. 对于此类函数静态插桩的常用方法是使用链接器特性. 例如在 Linux 下，系统的 C 运行时库将 *memcmp* 等函数标记为弱符号，从而允许用户重载并优化 *memcmp* 等函数. 基于上述机制，libFuzzer 重新定义了需要插桩的缓冲区比较函数. 在链接时，重定义的强符号比系统的 C 运行时库有更高的优先级，因而能替换原程序对系统库函数的调用. 除了截获比较函数，模糊测试工具在运行时也需要调用被截获函数自身. 为了获取原始符号，libFuzzer 在运行时通过系统的动态链接器提供的 *dlsym* 和 *dlvsym* 函数，动态获取原始函数的指针.

基于对整数和缓冲区比较的插桩，模糊测试能够追踪到控制流变化外的比较操作数，从而辅助对复杂约束的求解. 例如，libFuzzer 通过 2 种方式利用比较操作数，从而提升模糊测试效果.

1）自动字典构造. 对于长度为 4B 和 8B 的整数比较和长度为 2B 及以上的缓冲区比较，libFuzzer 将比较的操作数加入字典中. 在生成随机输入时，可以根据字典中的记录项直接替换部分原始输入，或在原始输入中插入部分字典记录项. 这种策略对于简易的常量比较有非常好的求解效果.

2）比较特征提取. 当开启值特征采集功能时，libFuzzer 将比较操作转化为特征位图，从而使先前应用于控制流特征的导向算法能够应用于比较中. 具体地，libFuzzer 首先度量比较操作数间的距离：对于整形比较，计算两数的海明距离和数值差异；对于缓冲区比较，则计算 2 个缓冲区前缀的海明距离. 接着，libFuzzer 计算操作数距离和比较操作的机器码地址的哈希，该哈希值被对应到特征位图中的某个位. libFuzzer 将该位置设为 1，从而使执行完毕后的位图分析逻辑能够检测到比较后的变化，并最终保留具有新颖比较特征的输入文件用于后续探索.

4 静态插桩的额外开销

在模糊测试的实践中，静态插桩的能力会受到实际因素的制约. 首先，不同的插桩类型可能会互相冲突. 例如，Address Sanitizer 和 Memory Sanitizer 同时使用影子内存技术管理内存对象的元信息，由于内存布局不能互相兼容，二者不能同时打开. 此问题较容易规避，例如，可以使用强化了不同安全特性的二进制执行相同的输入. 此外，更重要的是，静态插桩后的程序执行变慢，其额外开销严重影响了模糊测试效果. 当下的主流模糊测试技术无一例外地使用随机输入生成的策略. 因此，对程序状态空间的探索离不开海量的输入执行. 然而，越是复杂的插桩特性往往需要引入更多的桩点，且每个桩点需要执行更多的逻辑. 重量级插桩会显著降低程序执行速度，降低模糊测试的整体吞吐量，进而影响整体测试效果. 由于额外开销是由桩点的额外逻辑带来的，本身是不可避免的. 为了指导模糊测试实践中的静态插桩选择，本文测量了不同类型插桩、相同类型插桩下实现的额外开销.

实验中待测项目全部选取自 FuzzBench[15]. FuzzBench 是谷歌发起的模糊测试工具评测平台，平台中包含了大量被广泛使用的开源软件. 这些开源软件有着截然不同的功能，且均为应用模糊测试的实际项目，因此具有高度的代表性. 除非插桩方式有特殊要求，本文使用完全相同的编译器版本 Clang 14.0.6 和编译器优化参数 O2 构造二进制，从而保证相同的测试基线.

实验的测量方法是在相同测试集的重复实验. 由于模糊测试具有随机性，生成的输入文件也不完全一致，因此直接测量模糊测试的执行次数会受到随机性的影响. 为了控制变量，对于每个项目，本文均选取某次模糊测试中保存的输入文件作为测试集；对于每个插桩方式，本文使用 Linux 社区的 perf tools 工具集，用于精确的性能分析和测量. 测量环境的微处理器为 AMD EPYC 7742，内存为 256 GB，内核版本为 Linux 5.4.

实验的测量结果使用未插桩的原始二进制进行标准化. 对于各个插桩方式的测量值，本文呈现其测量读数除以原始二进制的测量值的标准化结果，从而排除不同项目的差异. 例如，当测量执行时间时，1.0 表示执行时间和原始二进制相同，数值越高，表示插桩的额外开销越大.

4.1 整体评估

图 1 呈现了不同插桩方法下各个插桩程序的执行时间分布. 数据经过归一化处理，其中横线表示原始程序的执行时间，即 100%. 为了提升表述的简明性，图 1 中将 Sanitizer Coverage 简称为 SanCov. 由于 Sanitizer Coverage 具有不同的子功能，在此将一般模糊测试中使用的开启全部功能的版本称为 SanCov-All；为了对比纯控制流信息下的不同采集方式，将 Sanitizer Coverage 中只收集控制流信息的版本称为

SanCov-Edge, 从而和 4.2 节的方法一致. 此外, 对于安全特性强化的插桩模式, 将 Address Sanitizer 简称为 ASan, 将 Memory Sanitizer 简称为 MSan, 将 Undefined Behavior Sanitizer 简称为 UBSan.

从图 1 中可以看出, 各个插桩模式对程序执行速度均有显著的影响. 其中, 影响最小的是 SanCov-Edge, 平均会带来 26% 的额外执行时间; 影响最大的是 SanCov-All, Sancov-All 包含了边覆盖率、比较操作数收集等丰富功能; 此外, ASan 也会带来 146%~230% 的额外执行时间.

由于图 1 呈现的是不同项目的执行时间分布, 还可以发现其更为宏观的特征. 其中, SanCov-All 和 UBSan 的执行时间差别较大, 标准差分别为 171% 和 213%. 例如, SanCov-All 会带来 10%~577% 的额外执行时间; UBSan 也会带来 8%~803% 的额外执行时间. 需要说明的是, 图 1 中的分布是使用核密度估计方法得到的宏观趋势, 出现小于 100% 的值属于正常现象.

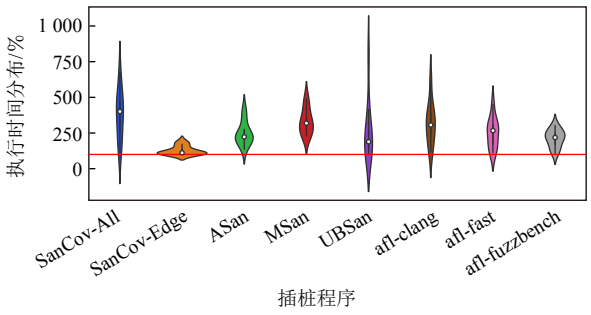

Fig. 1 Execution duration distribution of instrumented programs

图 1 插桩程序的执行时间分布

4.2 导向信息收集的额外开销

本文将测试工具 AFL 内置的导向信息收集插桩模式进行单独比较. 通过将 AFL 作为唯一的模糊测试工具, 能够避免不同测试工具所需的不同信息和不同信息的处理方式带来的影响. AFL 自身提供了 afl-clang 和 afl-fast 插桩工具. 对于 afl-fast 模式, 由于兼容性问题, 本文修改了插桩引擎代码, 使其能够适配于 LLVM 的新版 PassManager 组件. 此外, 本文还选取了谷歌 FuzzBench 平台上适配 AFL 的插桩模式, 记为 afl-fuzzbench.

如表 2 所示, 这 3 种模式中, 只有 afl-clang 使用汇编作为插桩对象, 其他模式使用 LLVM 中间表示; 只有 afl-fast 将桩点逻辑内联于程序的原始 LLVM 中间表示, 其他模式均在桩点处插入对外部运行时函数的调用指令.

图 2 显示了 AFL 插桩下不同程序的标准化执行

Table 2 Comparison of AFL's Instrumentation Modes

表 2 AFL 的插桩模式对比

模式	插桩对象	插桩方法
afl-clang	汇编	回调函数
afl-fast	LLVM 中间表示	桩点内联
afl-fuzzbench	LLVM 中间表示	回调函数

时间. 图中横线为原始程序的执行时间, 即 100%. 从整体上看, afl-clang 的性能较差, 额外开销为 219%; afl-fuzzbench 的性能较好, 额外开销为 112%. afl-clang 性能差的原因是基于汇编的插桩模式, 即只是使用基本的模式匹配识别基本块, 并未对原始程序的寄存器使用情况进行分析, 因此需要保存全部寄存器等大量的上下文, 存在大量的额外操作. afl-fast 基于 LLVM 中间表示, 上下文管理由 LLVM 后端在生成机器码时自动完成, 且高度优化. 然而, afl-fast 模式将计数器更新逻辑置于程序内部, 会带来可观代码膨胀. 以 harfbuzz 项目为例, 其在 afl-fuzzbench 模式下的二进制大小为 1.23MB, 但是在 afl-fast 模式下的二进制大下为 2.81MB, 相差 1.28 倍. 代码膨胀会增加微处理器的缓存压力和前端解码器压力, 因而影响程序执行速度. afl-fuzzbench 使用基于回调函数的方法, 尽管该方法需要增强函数调用和返回的成本, 但是现代微处理器的前端能够良好地预测执行流的切换, 因而较小影响.

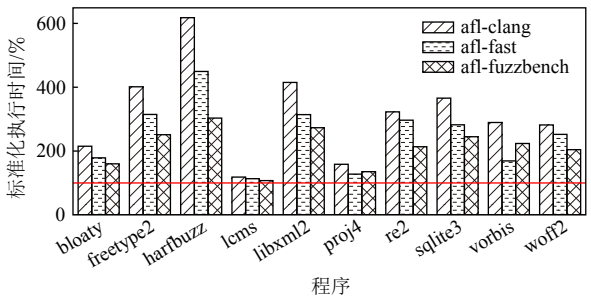

Fig. 2 Execution duration of programs under AFL instrumentation

图 2 AFL 插桩下的程序执行时间

此外, 尽管是相同的测试工具、相同的待测程序、相似的反馈信息, 但不同插桩模式下程序的性能差异极大. 例如, harfbuzz 在 afl-clang, afl-fast 和 afl-fuzzbench 插桩模式下, 执行速度分别为原始程序的 618%, 449% 和 303%.

图 3 进一步对比了 Sanitizer Coverage 不同模式下的插桩成本. 其中, SanCov-Edge 表示只统计最基本的控制流信息, 而 SanCov-All 表示还开启比较操

作数记录功能. 图 3 中的数据经过归一化处理，其中横线为原始程序的执行时间 100%.

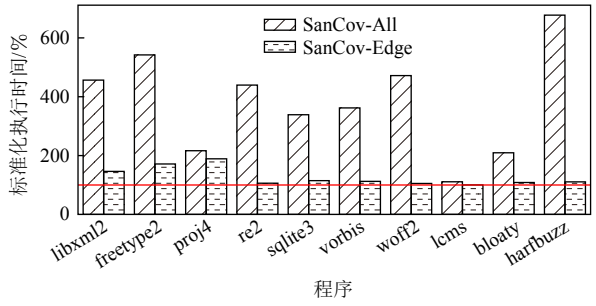

Fig. 3 Execution duration of programs under Sanitizer Coverage instrumentation

图 3 Sanitizer Coverage 插桩下的程序执行时间

对比图 3 中的 SanCov-Edge 和 SanCov-All 可以发现，丰富的比较操作数信息严重拖慢了程序的执行速度. 其中，最原始的 SanCov-Edge 模式只会带来 0%~89% 的额外开销. 然而，一旦开启了对比较操作数的追踪，SanCov-All 模式的额外开销便突增到 11%~577%.

比较操作数处理背后的巨大开销可以从 2 方面来解释. 首先，比较操作广泛存在于程序中. 以出现最大额外开销的 harfbuzz 为例，插桩后的程序中有近 11 000 个计数器桩点，此外还有 5 000 余个数值比较桩点和 70 余个缓冲区比较桩点. 虽然比较操作的桩点少于控制流桩点，但其量级却近似. 此外，对比较操作收集的插桩逻辑远比控制流特征的收集复杂. 以模糊测试工具 libFuzzer 为例，它的控制流插桩只须对固定位置的计数器做简单的自增操作；然而对于比较操作，它却需要以不同方式度量操作数的距离，计算当前比较的哈希，更新全局特征，还需要将操作数拷贝并编入字典.显然，复杂的桩点逻辑会给开启比较操作数导向的模糊测试工具带来巨大的额外开销.

4.3 安全特性强化的额外开销

图 4 显示了 OSS-Fuzz 项目中最流行的 3 类安全特性强化插桩给不同待测程序带来的额外开销. 需要注意的是，由于 MSan 可能的误报，导致 bloaty 项目不能正常编译，且 sqlite3 项目不能正常执行. 因此，本文剔除了上述不能正常评估的实验项.

从图 4 可以看出，从整体上：内存安全检查 ASan 会带来 146% 额外开销；未初始化内存读取检查 MSan 会带来 231% 的额外开销；未定义行为检查 UBSan 会带来 162% 的额外开销. 此外，不同项目和不同插桩方式的额外开销波动剧烈，并无显著规律，这是由插桩自身和程序自身的固有特性决定的. 例如，在检测未定义行为的 UBSan 中，大量的桩点都针对整数和指针运算. 因此，对于具有大量算术运算的压缩/解压缩类型的算法类程序（例如 woff2 和 vorbis），UBSan 具有巨大的额外开销. 类似地，ASan 对内存的读写都需要插桩，而 MSan 对内存的读写和值的传播都需要插桩. 综上，对于内存操作较为频繁的程序会在 ASan 和 MSan 插桩模式下有较大的额外开销.

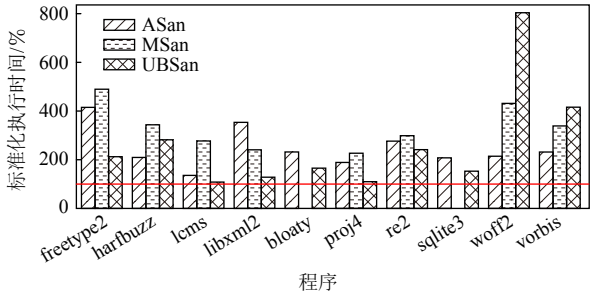

Fig. 4 Execution duration of programs under safety enhancement instrumentation

图 4 安全特性强化插桩下的程序执行时间

5 静态插桩的优化方向

5.1 基于硬件特性的优化

现代微处理器往往会提供不同的优化指令集. 一方面，可以利用硬件特性加速静态插桩. 例如，在收集控制流特征时精简插桩指令或加速分析[16]. 此外，还可以使用新的指令集架构加速内存安全检查[17-20]. 另一方面，可以利用硬件特性直接记录程序的控制流行为[21-23]. 虽然硬件本身能够以较低的成本收集程序行为，但在模糊测试上尚存在信息处理较慢的挑战[24].

5.2 基于程序分析的优化

利用编译器强大的静态分析能力可以减小插桩复杂度. 例如，可以通过控制流分析减少控制流插桩的桩点数量[25]，或是生成元信息从而辅助运行时的桩点删减[26-27]，或是增强对动态场景下覆盖率到源码的映射精度[28-29].

此外，运行时的动态分析信息也可以辅助模糊测试. 例如，可以通过引入模糊测试前的试运行步骤删除无关的安全检查[30-31]，还可以交替进行编译和运行，从而实现灵活的插桩调整[32].

6 结　　语

模糊测试是一种自动化的软件缺陷发现方法，

其实际应用往往伴随着静态插桩技术.本文介绍了模糊测试以及模糊测试背景下的静态插桩技术,统计了实际模糊测试中各类静态插桩技术的使用情况.针对内存安全检查、未定义行为检查、控制流特征采集、比较操作数收集这4类具有代表性的静态插桩技术,本文分析了其设计思想和实现技巧,对开发动态分析工具有参考意义.除了分析插桩的功能,本文还统计了不同插桩模式的额外开销并分析了额外开销背后的原因.基于上述内容,本文最后给出了静态插桩的两大优化方向,即利用硬件特性和程序分析极速执行或提升程度.

作者贡献声明:王明哲负责完成实验并撰写论文;姜宇提出了文章思路和实验方案;孙家广提出指导意见并修改论文.

参 考 文 献

[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32−40

[2] Godefroid P, Levin M Y, Molnar D A. Automated whitebox fuzz testing[C/OL] //Proc of the Network and Distributed System Security Symp 2008. Reston, VA: The Internet Society, 2008 [2022-10-12]. https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[3] Zalewski M. American fuzzy lop[CP/OL]. [2022-10-12]. http://lcamtuf.coredump.cx/afl/.

[4] Serebryany K. OSS-Fuzz-Google's continuous fuzzing service for open source software[C/OL] //Proc of the 26th USENIX Security Symp. Vancouver, BC: USENIX Association, 2017 [2022-10-12]. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

[5] Campbell J, Walker M. Microsoft announces new project OneFuzz framework, an open source developer tool to find and fix bugs at scale - microsoft security blog[EB/OL]. (2020) [2022-10-12]. https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/.

[6] Brereton F. Binspector: Evolving a security tool[EB/OL]. (2015) [2022-10-12]. https://blogs.adobe.com/security/2015/05/binspector-evolving-a-security-tool.html.

[7] Serebryany K. Sanitize, fuzz, and harden your C++ code[C/OL] //Proc of USENIX Enigma Symp. San Francisco, CA: USENIX Association, 2016 [2022-10-12]. https://www.usenix.org/conference/enigma2016/conference-program/presentation/serebryany

[8] Serebryany K, Bruening D, Potapenko A, et al. Address Sanitizer: A fast address sanity checker[C] //Proc of 2012 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2012: 309−318.

[9] Stepanov E, Serebryany K. Memory Sanitizer: Fast detector of uninitialized memory use in C++[C] //Proc of the 13th Annual IEEE/ACM Int Symp on Code Generation and Optimization. Washington, DC: IEEE Computer Society, 2015: 46−55

[10] Serebryany K, Potapenko A, Iskhodzhanov T, et al. Dynamic race detection with LLVM compiler[C] //Proc of the 2nd Int Conf Runtime Verification. Berlin: Springer, 2011: 110−114

[11] Lattner C, Adve V S. LLVM: A compilation framework for lifelong program analysis & transformation[C] //Proc of the 2nd IEEE / ACM Int Symp on Code Generation and Optimization. Washington, DC: IEEE Computer Society, 2004: 75−88

[12] Bernat A R, Miller B P. Anywhere, any-time binary instrumentation [C] //Proc of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools. NewYork: ACM, 2011: 9−16

[13] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation[C] //Proc of the ACM SIGPLAN 2007 Conf on Programming Language Design and Implementation. New York: ACM, 2007: 89−100

[14] ISO/IEC JTC 1/SC 22. ISO/IEC 9899: 2011 Information technology—Programming languages—C[S]. Geneva, CH: International Organization for Standardization, 2011

[15] Metzman J, Szekeres L, Simon L, et al. FuzzBench: An open fuzzer benchmarking platform and service[C] //Proc of the 29th ACM Joint European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2021: 1393−1403.

[16] Wang M, Liang J, Zhou C, et al. RIFF: Reduced instruction footprint for coverage-guided fuzzing[C] //Proc of 2021 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2021: 147−159

[17] Liljestrand H, Nyman T, Wang K, et al. PAC it up: Towards pointer integrity using ARM pointer authentication[C] //Proc of the 28th USENIX Security Symp. Berkeley, CA: USENIX Association, 2019: 177−194

[18] Yoo S, Park J, Kim S, et al. In-Kernel Control-Flow integrity on commodity OSes using ARM pointer authentication[C] //Proc of the 31st USENIX Security Symp. Berkeley, CA: USENIX Association, 2022: 89−106

[19] Li Yuan, Tan Wende, Lv Zhizheng, et al. PACMem: Enforcing spatial and temporal memory safety via ARM pointer authentication[C] //Proc of the 2022 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2022: 1901−1915

[20] Bernhard L, Rodler M, Holz T, et al. xTag: Mitigating use-after-free vulnerabilities via software-based pointer tagging on intel x86−64[C] //Proc of the IEEE 7th European Symp on Security and Privacy. Piscataway, NJ: IEEE, 2022: 502−519

[21] Delshadtehrani L, Canakci S, Zhou B, et al. PHMon: A programmable hardware monitor and its security use cases[C] //Proc of the 29th USENIX Security Symp. Berkeley, CA: USENIX Association, 2020: 807−824

[22] Ning Zhenyu, Zhang Fengwei. Understanding the security of arm debugging features[C] //Proc of 2019 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2019: 602−619

[23] Iannillo A K, Natella R, Cotroneo D, et al. Chizpurfle: A gray-box

android fuzzer for vendor service customizations[C] //Proc of the IEEE 28th Int Symp on Software Reliability Engineering. Piscataway, NJ: IEEE, 2017: 1−11

[24] Schumilo S, Aschermann C, Abbasi A, et al. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types[C] //Proc of the 30th USENIX Security Symp. Vancouver, BC: USENIX Association, 2021: 2597−2614

[25] Hsu C C, Wu C Y, Hsiao H C, et al. INSTRIM: Lightweight instrumentation for coverage-guided fuzzing[C/OL] //Proc of the 25th Annual Network and Distributed System Security Symp. Reston, VA: The Internet Society, 2018 [2022-10-12]. https://doi.org/10.14722/bar.2018.23014

[26] Nagy S, Hicks M. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing[C] //Proc of IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2019: 787−802

[27] Zhou Chijin, Wang Mingzhe, Liang Jie, et al. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling[C] //Proc of the 35th IEEE/ACM Int Conf on Automated Software Engineering. Piscataway, NJ: IEEE, 2020: 858−870

[28] Wang Mingzhe, Wu Zhiyong, Xu Xinyi, et al. Industry practice of coverage-guided enterprise-level DBMS fuzzing[C] //Proc of the 2021 Int Conf on Software Engineering: Software Engineering in Practice. Piscataway, NJ: IEEE, 2021: 328−337

[29] Gan S, Zhang C, Qin X, et al. Collafl: Path sensitive fuzzing[C] //Proc of 2018 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2018: 679−696

[30] Wagner J, Kuznetsov V, Candea G, et al. High system-code security with low overhead[C] //Proc of 2015 IEEE Symp on Security and Privacy. Pidcataway, NJ: IEEE, 2015:866 −879

[31] Zhang Jiang, Wang Shuai, Rigger M, et al. SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs[C] //Proc of the 15th USENIX Symp on Operating Systems Design and Implementation. Vancouver, BC: USENIX Association, 2021: 479−494

[32] Wang Mingzhe, Liang Jie, Zhou Chijin, et al. Odin: On-demand instrumentation with on-the-fly recompilation[C] //Proc of 2022 ACM SIGPLAN Int Conf on Programming Language Design and Implementation. New York: ACM, 2022: 1010−1024

Wang Mingzhe, born in 1996. PhD. His main research interests include fuzz testing and program analysis.

王明哲, 1996 年生. 博士. 主要研究方向为模糊测试和程序分析.

Jiang Yu, born in 1989. PhD, associate professor, PhD supervisor. His main research interests include cyber physical systems and software systems security.

姜　宇, 1989 年生. 博士, 副教授, 博士生导师. 主要研究方向为信息物理融合系统和软件系统安全.

Sun Jiaguang, born in 1946. Professor, PhD supervisor. Academician of the Chinese Academy of Engineering. His main research interests include computer graphics, computer-aided design, and software engineering and systems.

孙家广, 1946 年生. 教授, 博士生导师. 中国工程院院士. 主要研究方向为计算机图形学、计算机辅助设计、软件工程与系统.