相比传统模糊测试，智能模糊测试[6]利用了插装技术，产生的测试用例可以计算执行后的代码覆盖率，再应用遗传算法等，使变异测试用例时不再盲目，可以有目标的对特定测试用例应用变异策略，来进一步获得更深的执行路径或者执行分支，提高模糊测试的代码覆盖率，进而提高软件漏洞挖掘的能力。插装技术又分为动态二进制插桩和基于源码插装。动态二进制插桩是指在二进制程序执行过程中，在指令、函数和基本块等执行前后插入给定的代码来获取指令、函数和基本块的相关信息，代表性的插装平台 Valgrind[7]、Pin[8]、DynamoRIO[9]等。基于源码插装是指利用 GCC、CLANG 等编译工具的中间结果，将给定代码在基本块前后进行插装，而后编译工具将插装后的结果编译成二进制程序，代表的插装平台 AFL[10]等。但是，动态二进制插桩和基于源码插装影响了程序的执行速度，使用不当还可能带来其他问题。

从图 1.2 模糊测试工具的发展可以看出，为了软件漏洞的挖掘更加高效，模糊测试工具引入了遗传算法、进化算法等，优化了待变异的测试用例集的选取和变异后测试用例的选取，并充分应用 forkserver[10]、in-process[11]、分布式[12]等技术加快模糊测试的执行。

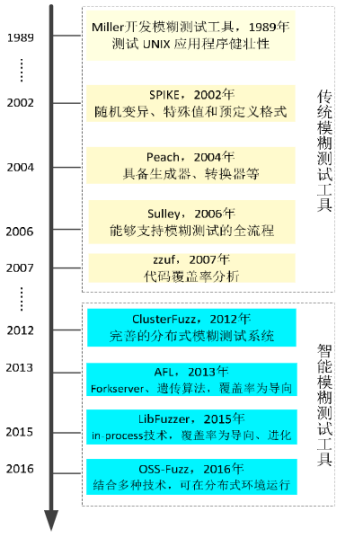

图 1.2 模糊测试工具的发展

目前对模糊测试的研究集中在利用遗传算法、白盒测试的符号执行方法、导向启发式算法等提高测试用例集的选择和测试用例的生成质量，扩大模糊测试的代码覆盖率；通过在分布式环境或者多核环境下引入并发技术来提高模糊测试的执行速度，加

快软件漏洞的挖掘。

Jääskelä, Esa[13]将遗传算法应用在以覆盖率为导向的模糊测试中，它能在一个Fuzzing 程序中自动平衡多个变异策略执行的可能性，这个平衡的目的是为了扩大模糊测试的代码覆盖率，优化测试用例的变异效率。

Osber 等[14]提出一种算法，用于从黑盒访问程序的一组输入示例中合成符合程序输入的上下文无关的语法编码。该算法解决了现有语法推理算法的缺点，这些算法过度笼统，并且速度非常慢。文中利用该算法模糊测试具有高度结构化输入的程序，结果显示，比起其他两个模糊测试工具，使用了该算法的模糊测试工具代码覆盖率不断增加。

Parvez 等[15]提出一种使用符号化执行产生测试用例的方法，这个测试用例可以达到潜在可能的漏洞目标状态（ potentially buggy “target” statement），扩大代码覆盖率。为了解决程序路径的指数级增长问题，作者提出了一个新的模式解决程序执行路径问题，而不耗费掉内存。

欧阳永基等[16]提出一种基于异常分布导向的智能模糊测试方法。该方法针对二进制程序测试，建立了样本构造模型：首先根据计算能力收集测试用例样本集；然后随机选择初始测试用例进行测试；最后，在测试结果基础上初始化构造模型的参数，根据模型优先选择有效的输入属性构造新样本并进行新一轮测试，通过重复上述步骤，在不断迭代测试中更新模型参数，为下一轮新测试样本构造提供指导。

Tielei Wang 等[17]实现了 TaintScope，它是一种校验和感知的导向型模糊测试工具，它能生成有效测试用例，自动检测软件存在的漏洞。由于传统模糊测试工具构造的大多数测试用例无法通过校验和检测，而被丢弃，无法对程序进行深入测试，因此作者提出了校验和感知技术。TaintScope 利用动态污点跟踪技术对输入测试用例中的敏感数据及校验和字节进行跟踪，敏感数据是对重要 API 参数有影响的数据。一方面，在生成测试用例时，只对敏感数据进行修改，有针对性地生成测试用例；另一方面，通过修改相应寄存器的值，使得生成的测试用例可以绕过校验和检测。

Stephens 等 [18]实现了混合漏洞发掘工具 Driller，它综合利用了模糊测试和具体符号执行两者的优势，进而找到更深层次的软件漏洞。模糊测试可以用来执行程序的某

个部分，具体符号执行用来生成输入的测试用例。Driller 使用具体符号执行来探测模糊测试工具感兴趣的路径，并生成模糊测试工具无法生成的输入测试用例。文章提出的方法可以提高生成测试用例的质量，找到更多的软件缺陷。

Valdez 等[19]设计并实现了 Confuzzer，使用具体符号执行技术和动态污点跟踪技术对获取不到源码的二进制进行模糊测试，以此找到二进制程序易受到攻击的输入。该系统的设计改进了具有大分支因子或大量基于确定控制流的复杂条件的模糊测试程序的性能。

对 AFL 和并行模糊测试研究的论文主要有下述几篇。

Cha 等[20]提出利用白盒测试的符号执行方法来处理程序中的路径，检测输入测试用例比特位置之间的依赖。根据这个依赖关系，针对给定的程序和种子对，计算出一个最佳的变异率。优点：用此算法优化 AFL，24 小时内发现的 bug 数比未优化的 AFL 多 18.5%。缺点：依赖被测程序类型，对开始输入的测试用例有要求。

Marcel Böhme 等[21]提出一种输入种子的选择策略，使得低频路径对应的种子输入更优先被选择。大部分的输入测试用例执行的都是少数的几条路径（称为 high frequency path，高频路径），这降低了模糊测试效率。优点：高频路径执行次数变少，模糊测试效率提高。此策略优化 AFL 后，显著地提高了发现崩溃的速度，并且发现了新漏洞。缺点：多进程执行时，冗余测试用例较多。

梁洪亮等[6]提出了一种可用于并行化环境中的路径取反算法和一种加入随机数据的复合测试用例生成方式。该路径取反算法给每个测试用例赋予一个边界变量，利用该变量限定每个测试用例可进行取反操作的范围，同时在该范围中对多个条件进行取反[6]。该复合测试用例的生成主要借助传统模糊测试技术生成随机数据，将该随机数据与混合符号执行生成测试用例相结合，从而生成复合化的测试用例。优点：算法能有效应用于并行环境中，一定程度上避免了多主机并行处理时重复工作。缺点：处理复杂软件时不能完全实现自动化测试。

Beterke[22]设计并实现了 Evofuzz 模糊测试工具，作者发现 AFL 是专门为单线程设计的，在多台机器上测试极不方便。作者保留了 AFL 的部分优点设计了一套专门在分布式环境下运行、以覆盖率为导向的进化模糊测试工具。作者证明了 Evofuzz 的有效

性，但是该工具仍存在一些问题，性能差，未实现类似 forkserver 或者 in-process 这样提高速度的功能，变异策略待优化，对待测试中的崩溃无法区别唯一性。

对于传统的专门为单机设计的模糊测试工具，要获得较高的代码覆盖率，需要花费大量的测试时间，并且目前程序越来越庞大复杂，这种传统的模糊测试工具在有限时间内很难获得令人满意的代码覆盖率，软件漏洞挖掘的效率很低，这是传统模糊测试工具面临的挑战。对此，国内外的研究者通过引入并发技术，充分利用计算机硬件条件，增加单位时间模糊测试进行的次数，加快路径覆盖速度，从整体上提高模糊测试的效率[23-27]。

1.2.2 AFL 介绍

AFL[10]（American Fuzz Lop）是当前最前沿、最先进的模糊测试工具之一，它是由谷歌员工 Michal Zalewski 开发。自 2013 年发布以来，它以有效、快速、稳定、易用的优点在安全领域广受称赞。据 AFL 官网粗略统计，AFL 在 125 套不同种类的著名软件中，发现软件漏洞达 260 个之多。它使用简便，不需要先行复杂的配置，能无缝处理复杂的现实程序。被测程序有无程序源码均可，有源码时可以对源码进行编译时插桩，无源码可以借助 QEMU 的 User-Mode 模式进行二进制插装。支持多平台（ARM、X86、X64）、多系统（Linux、BSD、Windows、MacOS）。与其他基于插桩技术的模糊测试器相比，AFL 具有较低的性能消耗，有各种高效的模糊测试策略和小技巧。

AFL 最优秀之处在于一是引入遗传算法如图 1.3，借助对程序的插装信息，每一轮变异的测试用例都是最有价值的，使得测试用例的变异不再盲目；二是将目前提高模糊测试速度的手段应用的淋漓尽致，forkserver 代码插入被测程序，使得每次 fork 被测程序时避免了程序初始化的开销。persistent mode 省去了重新启动程序的开销，更进一步加快了执行速度；三是提供了一系列实用的工具，如测试用例精简工具（afl-cmin、afl-tmin）、崩溃分析工具（afl-crash-walk）、查看程序对特定测试用例的反馈（afl-showmap）等；四是 AFL 对每个测试用例执行变异策略的次数把握的很好，因为对每个测试用例变异次数过多，产生大量无用测试用例浪费了 CPU 时间，而变异次数过少，达不到理想的代码覆盖率。

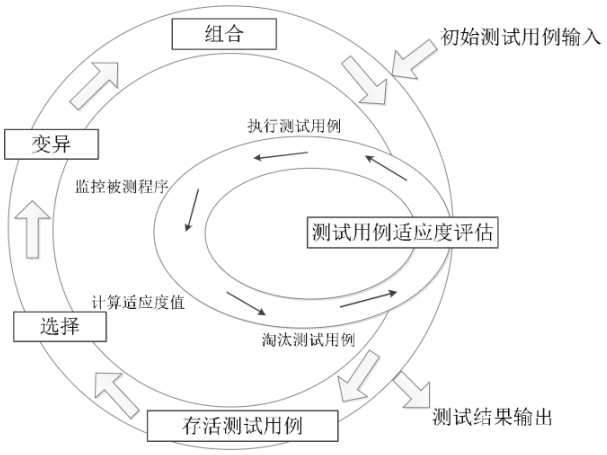

图 1.3 AFL 遗传算法的应用

AFL 系统框架如图 1.4 所示，总共由 9 部分组成，插装部分：有源码程序使用修改过的 gcc 或者 g++对程序进行编译时插装，二进制程序使用 QEMU 进行插装；变异策略部分：比特翻转、字典、特殊值等；forkserver：fork 和 execv 的汇编代码；监控状态：依据程序执行时的信号；选择测试用例：依据测试用例选择算法；路径比对：先比较哈希值，而后与 tuple 比较；覆盖率计算：已发现 tuple 数与总 tuple 数相除的结果；信息显示：提供执行结果的显示界面；工具：用于处理测试用例等。

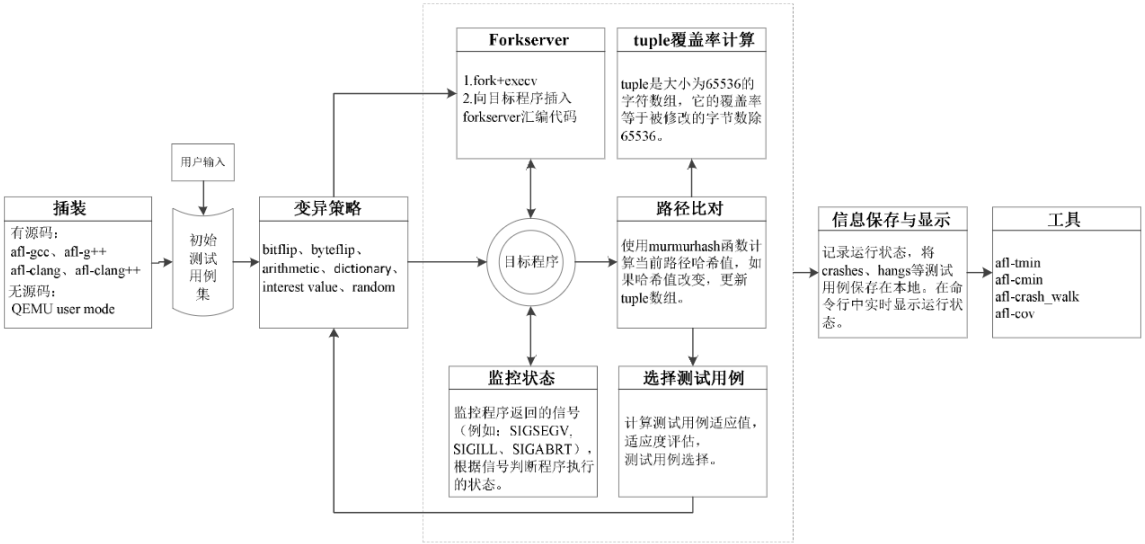

图 1.4　AFL 系统框架

1.2.3 并发无锁技术

近年来多核处理器发展迅速，桌面处理器，服务器处理器、移动处理器等至少有

两个核心。与单核处理器相比，多核处理器能够以更低的频率处理更高的工作负载[28]，因此，为了充分利用多核资源，并发编程越来越受重视。并发数据结构是并发编程的核心，它对并发程序的性能有重要的影响，更高的并发度和扩展性是并发数据结构研究的重点，图 1.5 是并发数据结构的分类。

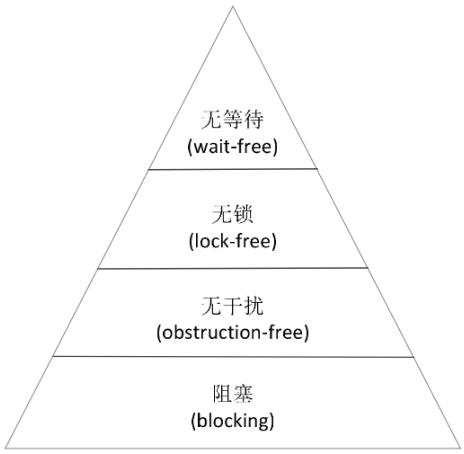

图 1.5 并发数据结构的分类

早期的并发数据结构都是阻塞并发数据结构，使用诸如信号量、管程、自旋锁同步机制来对并发线程进行同步。它的本质是传统非并发数据结构与同步机制的简单结合，将数据结构的修改操作作为临界区对待，并使用同步机制进行保护。这些同步机制虽然保证了多线程环境下并发数据结构的安全，但都需要操作系统进行干预，同步操作过程中会导致进程在用户态与内核态之间的切换，带来大量系统开销。

无干扰是并发级别最低的非阻塞演进条件，不需要使用锁，但并不能保证多线程并发执行[29]。无干扰算法一般引入后退机制，一检测到冲突的线程就终止，让较早执行的线程优先完成，而选择后退的时间等比较复杂，所以实用的无干扰算法较少。

无锁可以保证有一些调用能够在有限步骤内完成，无等待是无论其他操作如何，这种方法的每一次调用都可以在有限的步骤内结束[30]。无等待演进条件比无锁要强，无等待可以保证自身的执行在有限步骤内完成，任何其他线程的延迟都不会阻塞自身继续执行。但是无等待更难实现，目前实用的无等待数据结构很少。

Valois[31]使用 CAS 同步原语实现了第一个无锁链表，避免了一般方法开销大的缺陷。Harris[32]使用两步删除实现无锁并发链表，支持线性查找、删除操作。Michael[33]在 Harris 工作基础上，提出了高效的基于 CAS 的无锁链表，并将算法用在无锁哈希表

数据结构上，这个哈希表是桶固定大小，实现了无锁插入和删除。Treiber[34]实现了第一个无锁栈，实现了用 CAS 操作对头指针进行原子修改。随后，Hendler[35]等人解决了 Treiber 实现的无锁栈存在的问题，并突出了可扩展的无锁栈。Herlihy[36]等人提出了基于数组的无锁队列，Michael[37]等人提出了可扩展的无锁队列，允许在队列两端并发操作。Shalev 等人在 Michael 提出的无锁哈希表的基础上，实现了动态分裂有序哈希表，实现了哈希表桶的动态增长。

1.2.4 国内外研究现状小结

从图 1.2 可以看出，模糊测试工具正在由传统模糊测试工具向智能模糊测试工具发展，模糊测试工具引入了遗传算法、进化算法等，优化了待变异的测试用例集和变异后测试用例，主要是为了提高测试用例质量，使软件漏洞的挖掘更加高效。而模糊测试工具充分应用 forkserver、in-process、分布式[12]等技术，则是为了提高模糊测试的速度。

文献[13-27]关于模糊测试的理论研究主要是为了解决模糊测试工具遇到的问题，研究了多种方法如遗传算法、白盒测试的符号执行方法、导向启发式算法等提高测试用例的生成质量，减少冗余测试用例，扩大模糊测试的代码覆盖率；通过引入并发技术在分布式环境或者多核环境下来提高模糊测试的执行速度，加快软件漏洞的挖掘。

1.3 研究工作的意义和主要研究内容

从模糊测试工具的发展和国内外研究现状可以看出，变异策略是模糊测试的关键组成部分，它关乎到测试用例的生成质量，对代码覆盖率有直接影响。虽然目前利用遗传算法、符号化执行的方法和其他启发式算法使测试用例的生成质量有了提高，但是对待某些特定漏洞效果不理想。AFL 变异策略比较有代表性，本文通过分析 AFL 现有策略的缺陷，提出了 String Match 变异策略，针对某些程序可以快速发现程序执行新路径，提高代码覆盖率。

为了提高模糊测试的速度，模糊测试工具采取了诸如 forkserver、in-process 和引入并发技术在分布式环境下测试等方法，这些方法只关注了单个模糊测试进程的执行速度，或者模糊测试进程在分布测试环境中的调度等，没有研究并行的多个模糊测试进程在执行过程中如何通过交互和协作来提高模糊测试的速度。本文针对此类缺陷，设