**Trace-link-recovery基本原理：**

为了改善在前文提出的问题，本研究提出了一种针对操作系统内核开发出的实现测试用例和内核代码以及版本更新的补丁进行可追溯链接的专用方法。在这种方法中，摆脱了传统建立可追溯性链接的方法，即在不需要编写补丁的开发工程师与编写测试用例的测试工程师进行频繁的沟通，也不需要两方遵循固定规则编写代码的前提下，建立有效的可追溯性链接。在这种方法中，以系统调用以及内部的内核函数为中间桥梁。通过分别建立内核源码或者补丁与内核函数的可靠连接，以及测试用例与内核函数的可靠连接，从而实现将内核源码或者补丁与测试用例的可靠性链接。建立两者的链接所需时间只需一次，在之后的版本更新中，只需要对建立好的链接进行更新维护即可。此外，建立两者的链接是通过共同执行的内核函数来决定，所以其中不需要领域专家介入即可完成。

具体原理如【图一】所示：

通过内核追踪工具跟踪每个测试用例，获取测试用例与内核函数的多对多的对应关系，之后再通过分析补丁修改的内核代码，实现补丁与内核函数的对应关系。如果补丁和测试用例对应着同样的内核函数，则我们认为补丁与测试用例可以依照我们的方法建立可靠性链接。

**测试用例分析：**

本研究利用动态分析跟踪的手法来建立测试用例与内核函数的关系。我们假定系统更新之前的系统版本为v1，在v1版本下，将可以正常以可执行文件执行的测试用例整理为一个测试集。逐一对测试集中的测试用例进行运行并利用ftrace进行实时跟踪，ftrace会以程序的系统调用为借口开始对测试用例所调用的每一个内核函数进行跟踪和记录。对每一个测试用例形成一个测试用例对内核函数的调用关系图并存入trace.dat文件中，之后通过命令解析，可得出程序的内核代码执行路径表。表中包含：程序运行的进程号，CPU运行情况，调用的内核函数名称，调用函数的时间，函数调用及返回关系。之后，利用我们设计的脚本工具进行整理去重，形成测试用例和内核函数的对应关系表（如【表一】所示），在运行完所有测试用例之后。我们可以在v1版本下，形成一个测试用例与内核函数之间的一个多对多的对应关系。为了之后方便实行trace-link，我们将测试用例向内核函数的对应关系，转化为内核函数向测试用例的对应关系（如【表二】所示）。

在测试用例分析，即【表二】的构建过程中，会遇到以下常见问题：

Q1：如何确定ftrace跟踪的到的函数（表一中的函数）全部都为测试用例执行的内核函数，没有在测试用例执行期间，其他系统进程执行的函数，造成结果的误差。

A1：在运行每一个测试用例期间，系统中肯定会存在其他进程进行内核函数调用，得到的结果会出现冗余，针对这个问题，我们有两种解决方案：

1. 通过改变ftrace的追踪选项，只追踪固定进程号的系统进程调用的内核函数，其他执行的内核函数全部忽略。
2. 在解析trace.dat文件后，得出的输出文件，内部含有进程号以及执行系统调用的进程名称，可进行二次筛选。

对比两种方法，第二种方法虽然没有第一种方式省时，但比较便于操作，也不会因为操作失误造成实验结果的丢失，所以我们的方法采取了第二种解决方案，来解决在内核函数跟踪时可能产生的干扰问题。

Q2：全部执行所有的测试用例会消耗大量的时间，是否为了实现trace-link值得消耗这些时间成本。

A2：执行对所有测试用例的跟踪需要大量的时间，时间是运行所有测试用例和跟踪时间的总和，但对于频繁更新的内核来说，实现trace-link是十分必要的，形成图二的表的操作只需要一次，之后只需要对这个对应关系表进行维护和更新即可，相较于取得的效果，这些时间成本是完全合理并可接受的，具体时间成本的评估以及对之后系统更新产生的正向帮助所节约的时间后文会加以论述。

Q3：在进行ftrace追踪、解析以及分析数据的过程当中，会产生大量的中间文件，这些中间文件会占磁盘多大空间，是否会影响系统正常运行。

A3：对测试用例的跟踪会产生大量的中间文件，如追踪出的trace.dat文件，解析出的中间文件以及表一表二的数据等，会大量占据磁盘空间。我们提出了以下两个解决方案：

1. 在逐一运行测试用例时，生成了每一步新文件都对之前的中间文件进行完全删除，每个测试用例只保留【表一】一组数据，占用磁盘空间可以忽略不计。
2. 对于基于单纯循环的压力测试，我们认为每次循环调度的内核函数不会有太大差异，可以通过降低循环次数来减小中间文件的大小，这个方法同时对Q2的时间成本也产生了正向的帮助。但由于这个方法需要更改测试用例的源码，需要领域专家的介入，在这个版本的工具中这个方法只停留在理论可行，没有进行实现。

**内核补丁分析：**

与建立测试用例和内核函数可靠性链接不同的是，在建立内核补丁和内核函数之间的可靠性链接时，我们利用的不是动态的跟踪，而是利用静态的分析。我们假定系统从v1至v2更新了10个补丁，那么我们则对10个补丁进行分别分析，通过自动化或半自动化的分析手法，分析得出内核补丁所对应的内核函数。由于内核补丁是针对内核代码行进行的修改，通过分析将内核代码行归纳到内核函数名即可。内核补丁与内核函数应该是一个一对一或者是一对多的关系【表三】。通过表二与表三的综合，形成了内核补丁和测试用例的一对一或者一对多的可靠性链接。

内核补丁修改有以下几种情况，我们逐一进行分析：

