高技术通讯 2012 年 第22卷 第8期:804~811

doi:10.3772/j.issn.1002-0470.2012.08.005

# 提高录制-重放技术透明性及易用性的方法研究①

王 楠②*** 韩冀中③** 方金云*

(*中国科学院计算技术研究所 北京 100190)
(**中国科学院信息工程研究所 北京 100093)
(***中国科学院研究生院 北京 100190)

**摘 要** 为提高调试计算机系统的效率,提出了一种可有效地将非确定错误转化为确定错误的录制-重放机制。针对录制-重放系统加载中的透明性问题及重放中的易用性问题,通过开源项目 ReBranch 提出了一些解决方法。首次提出了基于解释器(interpreter)的透明加载机制,以解决加载机制的透明性问题;首次提出了一种基于文法压缩的异常序列检测机制,以提高重放的易用性。探讨了 ReBranch 的几个成功应用,并通过实验验证了异常检测算法的有效性。

**关键词** 调试,录制,重放,文法压缩,异常检测(AID)

## 0 引 言

计算机系统越发复杂,其非确定错误也就越多,对其调试的效率也就会越低。为提高调试效率,有人提出了录制-重放的调试方法[1],这种方法通过在程序运行时记录非确定信息并在调试时加以重放,可以将非确定错误转化为确定错误,进而允许程序员进行循环调试(cyclic debugging)。近年来,研究者们提出了不同的录制-重放方案,取得了一些成果。进行循环调试的基础是错误重现[2](bugs reproducing)。有研究表明[3],在一些场合下,调试的效率取决于重现错误的效率。但是,随着云计算和大规模并行系统的发展,人们越来越多地遇到非确定错误。这种错误一般是由难以控制的原因触发的,例如进程调度顺序、网络数据包延迟、数据竞争等。用传统的固定输入的方法难以重现这类错误。有些非确定错误通过压力测试能够随机地重现,但是有些错误仅在生产环境中才会发生。另外,调试这类错误还会遇到探针效应,即当调试时错误总是不发生。针对这种错误,目前通常的解决方法是通过人工检查程序输出的日志信息。

最近的研究开始考虑将录制重放技术部署在生产环境中,用于录制那些仅在生产系统中出现的错误。这些研究主要致力于降低录制开销。但要看到生产环境中的录制重放面临透明性和易用性问题。首先是录制机制的透明性问题。已有的研究一般都考虑了对应用程序的透明性和对环境的透明性问题。有研究[4]将透明性问题总结为"避免手工操作(包括重新编译、重新连接、修改代码等)"和"避免新硬件"两条。现有的录制重放机制基本上都能达到这些要求。但是对于加载机制的透明性,现有的研究并不充分。已有的录制机制多采用显式加载器或 LD_PRELOAD 方式加载。将这些加载方式引入复杂的生产环境会面临障碍。复杂系统的生产环境通常使用专门的控制机制。随着系统复杂程度的提高,配置这些控制机制的难度也相应提高。这些控制机制用于启动和终止程序,并对程序运行时状态进行监控。但是现有的录制机制加载方式会对复杂系统造成干扰。例如,有些机制(如传统的 fork()+ptrace()+execve()加载方式)会向运行时环境报告错误的进程号,使得根据进程号进行控制的机制(例如 Web 服务器控制 cgi 程序)无法正确运行;有些机制会导致进程名发生变化,使得根据进程名进

① 863 计划(2012AA01A401, 2011AA01A203),国家自然科学基金(61070028, 61003063, 60903047)和中国科学院先导专项(XDA06030200)资助项目。
② 男,1984 年生,博士;研究方向:大规模数据处理,大规模系统调试;E-mail:pi3orama@gmail.com
③ 通讯作者,E-mail:hanjizhong@iie.ac.cn
(收稿日期:2011-08-08)

行控制的机制(例如使用 killall 命令的脚本)无法正确运行。最严重的问题是,在复杂环境中使用显式加载器需要调试人员在控制系统中为所有相关的启动命令加上加载器或设置 LD _ PRELOAD,调试完毕后再去除它们。这加重了调试人员的负担,增大了出错的风险。易用性问题同样重要。本文关注的是重放的易用性。已有的研究基本上实现了自动重放,如 Snitchaser[5] 允许调试人员通过 gdb attach 命令在 gdb 的控制之下进行重放。但是在实际使用中还存在一个重要的易用性问题:重放位置的选择。生产环境下系统通常昼夜不断地运行,录制系统将产生大量日志。重放机制允许开发人员通过重放界面根据这些日志进行重放。但是受到重放系统性能的限制,从头重放如此长度的日志依然需要大量时间。如果被调试的程序没有最终崩溃或死锁,而只是中间的一些处理出现了问题,则需要有一种方法从大量处理产生的日志中找到几个出现异常的问题。现有的方法多没有这方面的机制,而将这项工作完全交给人工进行。由于 Logs 容量大,手工的方法工作量很大。这限制了录制重放方法的可用性。该问题本质上属于入侵检测(intrusion detection, ID)问题。现有的入侵检测或异常检测算法多使用统计分析、概率模型或机器学习的方法。很多方法假设日志是由 Markov 过程生成的。这种假设使得异常检测算法对高层次的异常行为不敏感。

本文提出了基于解释器(interpreter)的加载机制以实现加载的透明性;提出了一种新的异常检测算法以实现重放的易用性。

## 1　相关研究

### 1.1　录制重放

目前有很多关于录制和重放的技术。Liblog[6] 通过 LD _ PRELOAD 截获 glibc 的接口函数并加以录制,出错后使用录制中得到的日志进行重放调试。Jockey[7] 采用同样的加载方式,但是通过二进制插桩录制所有系统调用的执行结果。WiDS[8] 在 Windows 下定义了一组文件和网络操作的接口,使用这组接口编程可以进行录制和重放。R2[9] 和 liblog 类似,在 Windows API 层面上进行录制。Snitchaser[5] 通过 vDSO 录制 Linux 的系统调用。上述工作都是粗粒度的录制。

对于多线程程序,由于数据竞争的原因,任何潜在的访存都是非确定性的,因此为多线程程序设计的录制机制多采用细粒度的录制。以 BugNet[10] 为代表的方案通过特殊硬件跟踪并录制包括所有访存操作、中断在内的非确定事件;PinPlay[11] 等纯软件方案通过二进制翻译插桩等技术录制所有访存操作与同步操作。以 ODR[12] 和 PRES[13] 为代表的方案不录制所有非确定事件,但是重放之前需要通过大量计算恢复它们。

目前少有研究关注录制工具对运行时环境透明的问题。Lu 等人[2] 提出过两个指标:(1)避免手工修改代码和重新连接;(2)避免引入特殊硬件。实际上近来有很多录制机制都达到了这两个要求。但是目前的研究都没有考虑前文提到的在大规模系统中部署跟踪录制工具给环境造成干扰的问题。

### 1.2　入侵检测

入侵检测分为误用检测(misuse intrusion detection, MID)和异常检测(anomaly intrusion detection, AID)。MID 对异常的行为进行建模,之后可以找到它们并加以汇报。MID 对未知的异常效果不佳。AID 对正常的行为进行建模,如果被检测的行为偏移模型太多则汇报。AID 有能力检测未知的异常行为。对于异常检测的研究已经持续了很久[14]。很多工作研究基于概率统计的方法和机器学习算法[15-19]。这些方法在一些特定的日志,例如系统调用序列的异常检测方面取得了良好的结果。很多现有的异常检测方法基于 Markov 假设,即假设一个日志项出现的概率仅依赖于前一个日志项。Markov 假设使得异常检测算法对低层的异常行为很敏感,为防止误报甚至需要引入一些平滑算法[17]。另一方面,基于 Markov 假设的异常检测算法对于高层次的异常行为效果不佳。为检测高层次的行为异常可以采用高阶 Markov 模型或 n-gram 模型。但是阶数的设定依赖于对检测目标的理解。且上述两种模型都存在数据稀疏与计算复杂等问题。

## 2　录制重放框架

图 1 描述了一个透明与易用的录制重放调试流程。首先,被调试的应用程序被透明录制器加载,收集它的正常行为日志。之后开始录制,收集其运行时日志。如果运行时出现错误,需要从日志中将错误序列检出。为从大量运行时日志中检出异常序列,需要经过以下步骤:(1) 将正常行为日志分解为正常序列,进行训练,获得正常行为模型。(2) 将运行时日志分解为候选序列。(3)利用模型将候选日

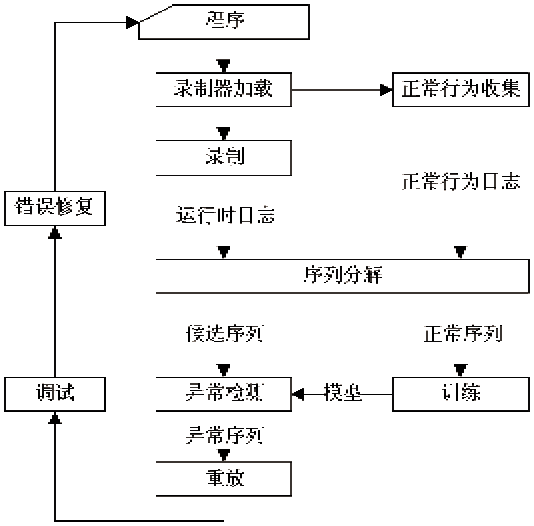


**图 1 透明易用的录制重放调试**

志分类为正常序列和异常序列。得到异常序列后，再由重放器进行重放。此阶段可以在重放器的协助下进行循环调试。最终找到错误原因并对程序进行修复。

在该框架中，为获得透明性，录制器加载机制需要降低其对环境的影响；为达到易用性，异常检测机制需要有效地对候选序列进行分类。本文通过开源项目 ReBranch[20] 实现了上述框架。ReBranch 开发了基于 interpreter 的加载机制和基于文法压缩的异常序列检测机制。

## 3 基于 interpreter 的加载机制

前文已经探讨过，为满足在复杂生产环境中录制其组件的需求，录制机制的加载方式要做到透明。我们对透明加载机制的要求包括以下几点：(1) 避免显式加载机制。有时启动目标程序的命令分散在大量脚本中，有时目标程序是通过专门的控制程序启动的(如被 php、CGI 调用的程序)。显式加载机制会影响这些控制机制。(2) 避免进程号、进程名称发生变化。有些控制方式(如 shell，killall 脚本)依赖于进程号和进程名，如果加载机制使得 shell 得到的进程号或者 ps 得到的进程名称发生变化，这些控制方式会受到影响。(3) 加载机制同时还应满足传统的要求：避免修改源代码；避免重新编译连接；避免引入特殊硬件；避免修改操作系统。

本文通过仔细研究 Linux 加载可执行程序的过程，提出了基于 interpreter 的加载机制，能够满足上述全部要求。在这种加载机制的支持下，只需要修改原先的可执行程序中的一个字节，该程序接下来的执行就会被录制，生产环境的控制机制不需要做任何改变。接下来本文详细阐述其原理。

绝大部分 Linux 平台使用 ELF 可执行文件格式。ELF 格式支持动态链接库(shared objects)，大部分库需要在程序正式执行前加载。为简化设计，Linux 内核不负责动态库的加载，而由 glibc 提供一个 interpreter 完成，内核仅需根据程序 program header 中 INTERP 字段的内容为其加载合适的 interpreter 即可。Interpreter 加载后，内核将使新建的程序从 interpreter 的入口地址执行。绝大部分 ELF 可执行程序都使用 interpreter。在 x86 环境下，所有应用程序的 interpreter 都是/lib/ld-linux.so.2 文件；在 x86_64 环境下，32 位程序依然是该文件，64 位程序则一般是/lib/ld-linux-x86-64.so.2。

本文提出的加载方法利用了 Linux 平台 ELF 文件加载过程中使用 interpreter 的原理。其核心在于提供一个与真正 interpreter 文件名仅相差一个字节的位置无关 ELF 文件作为加载器(如/lib/xd-linux.so.2)。通过此机制，只需要选择欲录制的可执行程序，将其 INTERP 字段中的字符串修改一个字节即可。该字段位于程序头部固定的位置，便于手工修改。通过此项修改，在内核加载可执行文件时，加载器会自动被加载并在应用程序开始运行前得以运行。图 2 用示例的方式说明了加载机制。录制前，事先将加载器命名为/lib/xd-linux.so.2。当试图录制目标程序(./executable)时，只需要在程序头部寻找 ld-linux.so.2 并将第一个字母修改为‘x’即可。可以看出，本文提出的加载机制实施很简便。

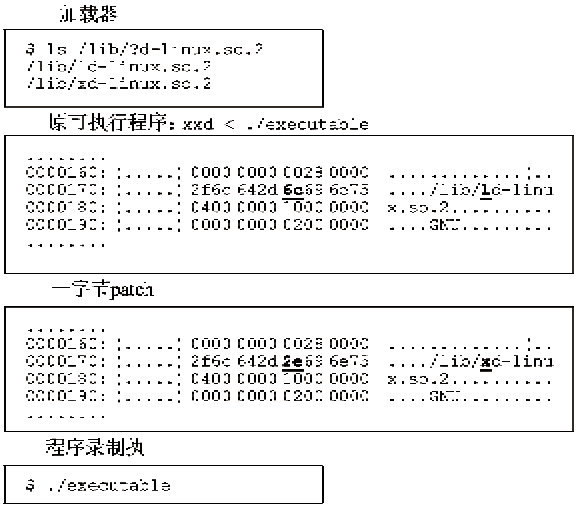


**图 2 一字节 patch 示例**

这种加载方式达到了上述几个要求:(1) 加载器是通过操作系统固有的启动机制被加载的,不需要通过命令行或者 LD _ PRELOAD。对于用户或控制脚本而言完全透明,避免了显式加载器。(2)加载机制不使用 fork( ) + ptrace( ) + execve( )的传统加载方法,不会产生额外的进程,内核加载的应用程序就是欲跟踪的应用程序本身,因此进程名、进程号等进程相关信息都不会发生变化。(3) 不修改源码、内核,不重新编译、连接,纯用户态,加载机制对程序员和运行时环境都透明。

在被修改过的 ELF 可执行文件被内核加载后,ReBranch 加载器将首先被运行。图 3 说明了加载器的工作流程。显然,上述加载器并非通常的可执行程序或动态库。在实现中需要考虑以下问题:

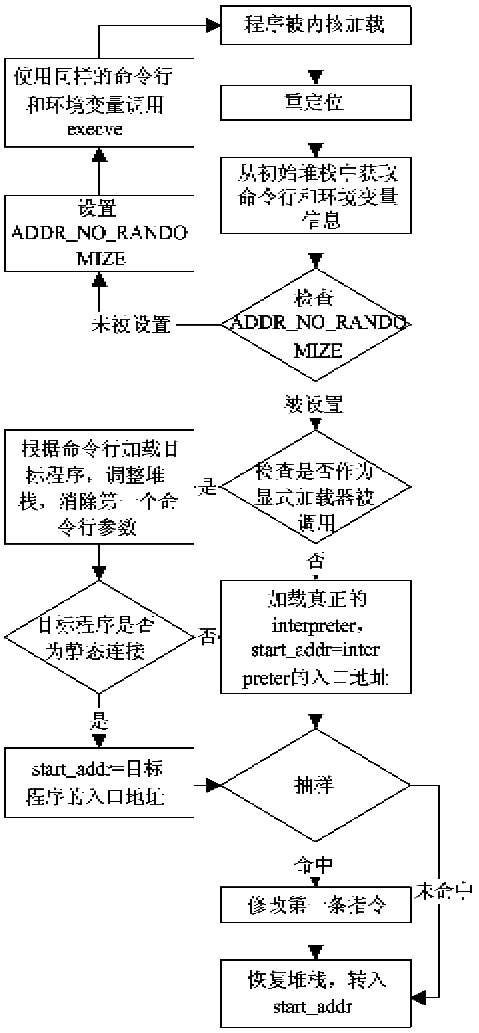


**图 3　透明加载机制**

(1)加载器本身是位置无关代码,需要经过重定位操作才能正常运行。而本加载器取代了原本用于重定位的 interpreter,因此本加载器需要在加载后立刻对自身进行重定位操作。

(2)由于加载器开始运行时所有动态库均无法使用,加载器本身必须是自我包含(self-contained)的。ReBranch 使用了部分从 Linux 内核源码树中移植的字符串处理函数。

(3)可执行程序需要 interpreter 进行重定位、加载动态库等工作,因此加载器在完成其初始化后需要将真正的 interpreter 读取并将控制权转移给它。

(4)静态连接的可执行程序不使用 interpreter。虽然可执行文件少有静态连接的,但考虑到完整性,ReBranch 加载器依然提供对静态连接程序的支持。对于这类程序,加载器需要作为显式加载器运行,替内核加载可执行文件。这种加载方式会引起进程名称的变更。

(5)加载完成后,需要在适当时候开始录制。为此,加载器通过修改目标可执行文件 ELF header 中 entry 地址处的指令,使录制机制在程序动态连接、重定位之后,正式开始执行前接管控制。

(6)对于生产系统中频繁运行的可执行程序,录制其每一次执行会给整个系统造成很大的性能影响。为此,本加载器使用一个随机数发生器决定是否对本次执行进行录制。通过读取环境变量中的一个配置变量,使用者可以控制被录制的比例。

表 1 比较了 ReBranch 和其它一些调试、录制重放机制在加载机制透明性方面的区别。从表中可以看到,和传统的加载机制相比,本文提出的加载机制透明程度最高。

**表 1　ReBranch 和其它调试机制加载方式比较**

| 工具 | 加载方式 | 控制脚本 | 进程号 | 进程名 |
|---|---|---|---|---|
| GDB[31] | ptrace( ) | 不可用 | 改变 | 不变 |
| Valgrind[23] | 显式加载器 | 需变更 | 不变 | 改变 |
| Liblog[6] | LD _ PRELOAD | 需变更 | 不变 | 不变 |
| Snitchaser[5] | ptrace( ) | 需变更 | 改变 | 不变 |
| ReBranch | Interpreter | 不变 | 不变 | 不变 |

## 4　基于无损压缩的异常序列检测

*ReBranch* 在录制阶段进行指令级的细粒度录制,通过插装技术将所有非确定转移指令的目的地址记录在日志中。录制阶段结束后,如果发现了 bug,就需要根据这些日志重现 bug 发生前后的过程。现有的研究,包括 ReBranch,都能提供自动化的重放工具。ReBranch 提供了两种重放工具:(1)

ReBranch 本身可以作为一个 gdb server，调试人员可以通过 gdb 远程协议控制重放过程。(2)根据 ReBranch 录制得到的信息，结合程序中的调试信息可以将其转换行号日志，如图 4 所示。ReBranch 提供了一个图形界面，可以根据行号日志进行自动重放。图形界面提供前进、断点等功能。由于粒度很细，ReBranch 对一个程序进行录制后能得到大量日志。如上文所述，即使通过自动重放，从这些日志中找出值得关注的部分依然是一个耗时的问题。

```
...
buffer_init:/../../buffer.c:29
buffer_init:/../../buffer.c:30
chunk_init:/../../chunk.c:40
chunk_init:/../../chunk.c:41
...
```

**图 4 处理后的 ReBranch 日志**

本文提出了一种基于无损压缩的异常检测算法，用于从大量上述行号日志中检测出异常的部分以便重放。在实施该算法之前，需要首先根据对目标程序的理解将行号日志分解成候选序列的集合，使得集合中的每个候选序列都是有意义的。例如，若录制的对象是 Web 服务器，则可以将其处理请求的函数入口所在行作为序列开始，将函数出口所在行作为序列结束，以此为标准进行图 1 中的序列分解。这样，得到的每个序列都代表该 Web 服务器处理一个请求的过程，异常的序列代表的过程和其它请求有较大差异。在 ReBranch 中，首先需要收集一些已知为正常的处理日志并进行序列分解，作为训练集合。

后文详细阐述该算法的原理。

在基于统计或概率模型的异常检测算法之外，Lee 等人指出，还可以根据序列的信息含量(熵)来进行异常检测[23, 24]。他们发现异常的样本含有较多信息，通过条件熵与信息增益等信息论指标就可以找出异常的样本。但是目前此类的研究都直接对熵进行估算，且估算方法依然为基于统计或 Markov 模型的方法。另一方面，Shannon 在其工作[25]中揭示了无损压缩和熵之间的关系，提出了重要的结论：熵是无损编码的下界，即一个理想的无损压缩编码方法得到的数据长度应该能够反映被编码数据的信息量。在无损压缩方面，有很多工作超越了统计模型和 Markov 假设，利用了序列结构信息。如应用最为广泛的 LZ77[26]依赖于发现和利用被压缩序列中重复的成分，文法编码[27]则致力于寻找序列的层次结构。

本文的异常检测算法结合了以上两方面的研究，提出利用无损压缩衡量序列的熵，再根据熵评价一个序列的异常程度。算法可以分成以下几步：

(1)训练：将一个已知为正常的序列集合 $N$ 使用压缩算法 C 进行压缩，$L_0 = |C(N)|$。

(2)评价：对每个候选序列 $x$，将集合 $N \cup \{x\}$ 使用同样算法进行压缩，$|C(N \cup \{x\})| = L_x$。令 $I_x = L_x - L_0$，$D_x = I_x / |x|$。

(3)选择：根据 $D_x$ 对候选序列进行分类，$D_x$ 较大的为异常序列。

上述算法中，$D_x$ 为候选序列 $x$ 中每个符号对序列整体信息量的贡献程度。容易理解，如果有两个序列 $x_1$ 和 $x_2$，$I_{x_1} = I_{x_2}$，但 $|x_1| > |x_2|$，即一个较长的序列 $x_1$ 产生的信息量与较短的序列 $x_2$ 相同，则认为较短的序列异常程度较大是合理的。

上述算法中，压缩算法 C 的选择对于异常检测的效果至关重要。常见的通用无损压缩算法如 gzip，bzip2，rar 等有时不适合作为算法 C，原因如下：(1)大部分通用压缩算法基于 LZ77[26]算法。该算法使用了滑动窗口，在压缩时被移出滑动窗口的数据无法影响正在压缩的数据。这会导致一些靠前的正常序列无法被匹配，和它们相似的候选序列会被分类成异常序列。(2)通用压缩算法以字节流为压缩对象，以字节为压缩的最小单位，使用 256 个字节作为固定字母表。本算法考虑的是从行号序列中找出模式，使用这样的字母表会影响模式的发现。(3)通用压缩算法需要将数据组合成流，这使得一些本不存在的模式被无意中建立。例如，1234 和 1236 是两个原本无关的正常序列，如果合成流 12341236 并压缩，模式 3412 就会被建立。这会使得直观上和两个正常序列都很不同的序列 3412 被识别为正常序列。

文法压缩能够解决上述前两个问题。文法压缩算法是近 10 年来发展起来的一种无损压缩技术[27, 28]。该算法将串 $x$ 转化成文法 $G_x$，其中 $L(G_x) = \{x\}$。对于高度结构化的数据，用于存储 $G_x$ 所需空间会显著减小。Kieffer 等在文法压缩方面做过很多工作，包括证明了文法压缩方面的几个重要结论[27]，提出了一个文法压缩的贪婪算法并使用算术编码对生成的文法进行编码[28]等。Nevill-Manning 等独立地提出了 SEQUITUR 算法，并证明了它是线性时间复杂度的[29,30]。SEQUITUR 算法也是一种文法压缩算法，但是生成的语法不够紧致。

本文采用的算法 C 是基于 Kieffer 等提出的贪

婪算法改进而得到的。该算法不需要字母表和滑动窗口。且由于 ReBranch 处理后的日志是高度结构化的程序行号序列,文法压缩的压缩比非常高。本文所作的改进主要是消除序列之间的顺序关系,避免跨样本模式。图 5 列出了算法的伪代码。其关键之处在于,在压缩的时候就将不同样本分开处理,确保每个样本都对应一条文法规则(伪码中的 $p_{seq}$);另外不将 $p_0$ 放在文法压缩算法的参数 $G$ 中。这样处理可以确保文法压缩中新增的模式不会跨越样本的边界。

```
# LogTransform函数返回起始产生式p0和产生式集合G
# 其输入Log是序列的集合
# 返回的p0具有S0→α1 α2… αn 的形式
# 每个αx展开后都是Log中某个序列。
# G为文法中除p0外的产生式
# GrammarTransform函数为一般的文法编码函数，其根据已有的
文法规则集合G对seq进行编码，其间可能会向G增加新的规则，
或删除规则。
def LogTransform(Log):
  p0=S0→ε
  G={ }
  for seq ∈ Logs
    pseq=GrammarTransform(seq, G)
    #pseq具有αseq→x1x2…形式，其中αseq为一非终结符
    if αseq展开后长度为1:
      continue
    将pseq加入集合G。
    将αseq追加到p0右边。
  return (p0, G)
```

**图 5　LogTransform 算法**

我们使用最终生成的语法中使用的符号个数来评价一个 Log 含有的信息,即:$L = \sum_{p \in \{p_0\} \cup G} |p|$。

## 5　实验分析

我们使用 ReBranch 实际调试的两个错误对所提出的透明性与易用性进行评价。两个错误分别是 memcached 106 号 bug 和 lighttpd 2217 号 bug。我们选择的两个错误均是难于重现的非确定性错误,最终在 ReBranch 的重放下得到解决。

lighttpd 的 2217 号错误是当 CGI 请求结束得非常快时,后续的有些 CGI 请求超时。引起这个错误的原因是一个和非阻塞IO与进程退出之间的竞争条件。该错误在行号序列上的表现是出现错误的请求经过了一些特定的行,而正确的请求没有经过这些行。这属于一个低层次的异常。

memcached 106 号错误实际上是两个独立的错误构成的。修复了一个 UDP 导致的死锁问题后,当 memcached 收到一个特殊的 UDP 包后,随后的请求就会随机出现错误。该错误体现在行号序列上是一个典型的高层次错误:正常的请求和错误的请求经过了相同的行,异常体现在序列的结构中。

在调试过程中 ReBranch 的透明性加载起到了重要作用:我们在不进行重新编译的情况下录制一个复杂系统中的某些组件,录制器的加载仅涉及修改目标程序的一个字节,对系统其余部分没有造成任何影响。

**异常检测实验**

上述两个错误都是非确定错误,且不造成系统崩溃。因此需要从若干请求中提取出异常的串。在实际调试时,对于 lighttpd 错误,我们需要从 1000 个 http 请求中找出两个错误的请求;对于 memcached 错误,我们从 1003 个请求中找出 3 个错误的请求。

对于 lighttpd,我们用 500 个正确的 http 请求作为训练集;对于 memcached,我们用 1000 个正常的 UDP 请求作为训练集。

在序列分解中,对于 lighttpd,每个序列从 connection _ state _ machine( ) 函数的入口开始,到该函数出口结束。对于 memcached,以 event _ handler( ) 函数的入口和出口作为序列的起点和终点。这样每个序列都表示一个事件的处理过程。

表 2 列出了用于异常检测的数据集规模。

**表 2　异常检测数据集规模**

| | 训练序列 | 训练日志项 | 候选序列 | 候选日志项 |
|---|---|---|---|---|
| lighttpd | 2501 | 3337395 | 4996 | 6661425 |
| memcached | 1442 | 862636 | 1609 | 883226 |

经过训练,lighttpd 训练数据集被压缩成 2793 个符号;memcached 训练数据集被压缩成 582 个符号。

我们列出两个数据集中根据训练结果进行的异常检测选出的 $D_x$ 最大的 5 个序列。表 3 分别列出了这些序列。

表 3 异常检测结果:最大的 5 个

| lighttpd | memcached |
|---|---|
| $D_{654}$ = 0.039653 | $D_{1237}$ = 0.031765 |
| $D_{3990}$ = 0.039653 | $D_{1608}$ = 0.031765 |
| $D_{655}$ = 0.019231 | $D_{1609}$ = 0.031765 |
| $D_{3991}$ = 0.019231 | $D_{1}$ = 0.009091 |
| $D_{22}$ = 0.018868 | $D_{6}$ = 0.009091 |

从表中可以看出,lighttpd 数据集中有两个序列(654 和 3990) $D_x$ 值显著大于其它的序列(是排名第 3 的序列 $D_x$ 值的两倍);memcached 数据集中有三个序列(1237,1608 和 1609) $D_x$ 值显著大于其它序列(是排名第 4 的序列 $D_x$ 值的 3.5 倍)。经人工检验,发现这 5 个序列的确和问题相关。

表 4 列出了上述训练和评价过程使用的时间及吞吐率。从表中可以看出训练和评价过程都在较短时间内完成了。

表 4 训练与评价速度

| | 训练 | | 评价 | |
|---|---|---|---|---|
| | 时间(s) | 吞吐率(日志项/s) | 时间(s) | 吞吐率(日志项/s) |
| lighttpd | 90.1 | 36918.1 | 496.3 | 13422.2 |
| memcached | 10.1 | 85409.5 | 28.7 | 30817.4 |

我们也考虑了将基于统计的异常检测算法如 Tian 等提出的系统调用序列异常检测算法[17]应用于本问题的可能性。该算法根据训练集计算出 Markov 模型,之后计算候选样本的概率,报告低概率的样本。在 lighttpd 实验中,该算法是有效的,因为异常的处理转移到了训练集中不曾遇到的行。该算法给这类事件一个极低的概率或 0 概率。但是对于 memcached 实验,单纯考虑行号之间的转移,则候选集中的每个转移都曾经在训练集中出现过,基于 Markov 假设使算法无法找出这一高层次的错误。可见对于在选择重访对象的问题本算法更加合适。

## 6 结 论

本文研究了录制重放技术中的两个关键性问题:透明性与易用性,并提出了基于解释器的透明加载机制和基于文法压缩的异常序列检测方法。就我们所知,本研究是首先关注录制重放机制加载过程的研究,也是提出通过文法压缩进行异常检测的研究。本文提出的方法在开源项目 ReBranch 中得到了应用。实验结果表明,透明加载机制对于大规模系统调试有重要意义;基于文法压缩的异常序列检测算法能够有效地从大量日志中找到异常的几个问题。ReBranch 已经被应用于几个真实的项目中,有效地提高了调试的效率。

**参考文献**


[1] Ronsse M, Bosschere K D, Christiaens M, et al. Record/replay for nondeterministic program executions. *Communications of the ACM*, 2003, 46(9):62-67

[2] Park S, Zhou Y, Xiong W, et al. PRES: probabilistic replay with execution sketching on multiprocessors. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, USA, 2009. 177-192

[3] Lu S, Park S, Seo E, et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, Seattle, USA, 2008. 329-339

[4] Lu S, Li Z, Qin F, et al. BugBench: Benchmarks for Evaluating Bug Detection Tools. In: Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools, Chicago, USA, 2005. 1-5

[5] Wang N, Han J, Fu H, et al. Reproducing non-deterministic bugs with lightweight recording in production environments. In: Proceedings of the IEEE 29th International Performance Computing and Communications Conference (IPCCC 2010), Albuquerque, USA, 2010. 89-96

[6] Geels D, Altekar G, Shenker S, et al. Replay debugging for distributed applications. In: Proceedings of the annual conference on USENIX'06 Annual Technical Conference, Boston, USA, 2006. 27-27

[7] Saito Y. Jockey: A user-space library for record-replay debugging. In: Proceedings of the sixth international symposium on Automated analysis-driven debugging, New York, USA, 2005. 69-76

[8] Liu X, Lin W, Pan A, et al. WiDS checker: Combating bugs in distributed systems. In: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07), Cambridge, USA, 2007. 19

[9] Guo Z, Wang X, Tang J, et al. R2: An application-level kernel for record and replay. In: Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08), San Diego, USA, 2008. 193-208

[10] Narayanasamy S, Pokam G, Calder B. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In: Proceedings of the 32nd annual international symposium on Computer Architecture, Madison, USA, 2005. 284-295

[11] Patil H, Pereira C, Stallcup M, et al. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, Toronto, Canada, 2010. 2-11

[12] Altekar G, Stoica I. ODR: output-deterministic replay for multicore debugging. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09), Big Sky, USA, 2009. 193-296

[13] Park S, Zhou Y, Xiong W, et al. PRES: probabilistic replay with execution sketching on multiprocessors. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, USA, 2009. 177-192

[14] Denning D E. An intrusion-detection model. Software Engineering, *IEEE Transactions on Software Engineering*, 1987, (2):222-232

[15] Qiao Y, Xin X, Bin Y, et al. Anomaly intrusion detection method based on HMM. *Electronics Letters*, 2002, 38(13):663-664

[16] Cho S B, Park H J. Efficient anomaly detection by modeling privilege flows using hidden Markov model. *Computers & Security*, 2003, 22(1):45-55

[17] Tian X, Cheng X, Duan M, et al. Network intrusion detection based on system calls and data mining. *Frontiers of Computer Science in China*, 2010, 4(4):522-528

[18] Hyun Oh S, Suk Lee W. An anomaly intrusion detection method by clustering normal user behavior. *Computers & Security*, 2003, 22(7):596-612

[19] Eskin E, Arnold A, Prerau M, et al. A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data. Applications of Data Mining in Computer Security, 2002, 77 - 102

[20] ReBranch: A Debugging Tool for Replay Bugs. 2011. http://code.google.com/p/rebranch/

[21] GNU. GDB: The GNU Project Debugger. At http://www.gnu.org/software/gdb, 2009

[22] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, San Diego, USA, 2007. 89-100

[23] Lee W, Xiang D. Information-theoretic measures for anomaly detection. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, Oakland, USA, 2001. 130-143

[24] 潘峰, 蒋俊杰, 汪为农. 异常检测中正常行为规则性的度量. 计算机研究与发展, 2005, 42(8):1415-1421

[25] Shannon C E. A mathematical theory of communication. *Bell System Technical Journal*, 1948, 27(3):379-423

[26] Ziv J, Lempel A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 1977, 23(3):337-343

[27] Kieffer J C, Yang E. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 2000, 46(3):737-754

[28] Yang E H, Kieffer J C. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. I. Without context models. *IEEE Transactions on Information Theory*, 2000, 46(3):755-777

[29] Nevill-Manning C G, Witten I H. Compression and explanation using hierarchical grammars. *The Computer Journal*, 1997, 40(2 and 3):103

[30] Nevill-Manning C G, Witten I H. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 1997, 7(1):67-82

# Study on improving the transparency and usability of record-replay systems

Wang Nan[*,***], Han Jizhong[**], Fang Jinyun[*]

([*]Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([**]Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093)

([***]Graduate University, Chinese Academy of Sciences, Beijing 100190)

**Abstract**

In view of the tendency that non-deterministic software bugs of a computer system become more frequent with the growing of system complexity, this study proposed a record-replay mechanism as an effective solution to find the root cause of those bugs to improve the efficiency of the system's debugging. The transparency and the usability, two important features of a record-replay system, were studied. An interpreter-based loading mechanism was proposed to implement transparent loading. A novel anomaly detection algorithm was proposed for mining buggy execution sequences from logs. Those two approaches have been introduced into the ReBranch, an open source debugging tool, and have helped developers to fix some real bugs. To the best of the authers' knowledge, this is the first work aiming at transparency loading, and also the first work applying grammar-based compression to anomaly intrusion detection.

**Key words**: debugging, recording, replay, grammar-based compression, anomaly intrusion detection (AID)