

软件崩溃研究进展

顾咏丰¹, 马萍¹, 贾向阳¹, 江贺², 玄跻身^{1*}

1. 武汉大学计算机学院, 武汉 430072

2. 大连理工大学软件学院, 大连 116621

* 通信作者. E-mail: jxuan@whu.edu.cn

收稿日期: 2019-01-28; 修回日期: 2019-06-06; 接受日期: 2019-08-13; 网络出版日期: 2019-11-04

国家重点研发计划(批准号: 2018YFB1003901)、国家自然科学基金(批准号: 61872273, 61502345)和湖北省技术创新重大专项(批准号: 2017AAA125)资助项目

摘要 软件崩溃指程序的意外中断, 是软件故障的一种严重表现形式。软件崩溃危害巨大: 一方面, 崩溃发生的反复性会带来较差的用户体验并影响公司的声誉; 另一方面, 崩溃发生的突然性会给用户造成巨大的甚至无法挽回的损失。随着现代软件的规模和复杂性不断增大, 如何有效地防止和应对软件崩溃成为了热门研究问题。本文从软件崩溃的分析、重现、定位修复这3个方面出发, 简要地回顾和总结了近年来软件崩溃的研究进展。

关键词 软件崩溃, 崩溃分析, 崩溃重现, 崩溃定位, 崩溃修复, 程序调试, 程序异常处理

1 引言

软件开发过程伴随着程序员与软件故障的不断斗争。程序员在日常开发和维护过程中会面临大量的软件故障^[1]。处理和修复这些故障一直是程序员的梦魇, 研究表明, 现代软件有超过45%的开发成本会消耗在定位和修复软件故障的过程中^[2]。软件故障由软件自身缺陷造成且原因众多: 有的源自内在的算法实现问题, 如非法除零、访问数组越界、内存泄漏等; 有的则源自复杂的业务逻辑, 如并发操作出错、流程处理出错、数据库访问失败等。这些故障的表现形式大体分为两种: 第1种称之为需求违背(violation), 指的是软件的实际输出与预期输出不符; 第2种称之为崩溃(crash), 指的是程序的意外中断^[3], 它造成的后果将更加严重。

虽然软件崩溃是软件故障的一种严重表现形式, 但它在生活却中十分常见: 电脑操作系统的崩溃让我们的工作前功尽弃, 服务器的崩溃让我们无法获取应有的服务, 手机应用的崩溃让我们不能与他人沟通交流, 网络结点的崩溃让我们无法畅游网络。这些崩溃都让我们的生活工作变得困难。一方面, 软件崩溃的频繁发生会使得用户体验变差, 进而影响软件公司的声誉。另一方面, 软件崩溃的突发性往往让用户猝不及防, 严重的系统崩溃将带来巨大的甚至无法挽回的损失。1990年, 美国电话电报公

引用格式: 顾咏丰, 马萍, 贾向阳, 等. 软件崩溃研究进展. 中国科学: 信息科学, 2019, 49: 1383–1398, doi: 10.1360/N112019-00018
Gu Y F, Ma P, Jia X Y, et al. Progress on software crash research (in Chinese). Sci Sin Inform, 2019, 49: 1383–1398,
doi: 10.1360/N112019-00018

司 (AT&T) 的通话网络出现故障^[4]. 故障导致相邻的交换机间不断相互误报“正在修复”的信号, 陷入了“重启 – 开机”的死循环中. 随着故障范围不断扩大, 事故发生 9 小时后, 整个美国的电话系统崩溃. 航空、酒店、租车、信用卡等相关产业受到了严重影响, 事故方 AT&T 公司在不到半天的时间里遭受到了 6000~7500 万美金的巨额损失, 更为严重的是其一百多年积累的“可靠的电话公司”的形象也轰然倒塌. 2006 年, 美国航空航天局 (NASA) 发射的“火星全球勘测者号”探测器出现突发性崩溃^[4]. 事后调查显示由于软件更新的失误, 使得探测器的内存地址的存储出现了错误, 导致了该火星探测器的系统崩溃, 并与地面人员失去联系. 这个令人心痛的崩溃不仅耗费了 NASA 高达 2.4 亿美金的开发成本, 还让人类失去了一次宝贵的探索太空的机会. 软件崩溃的危害巨大, 研究人员希望能够深入分析和研究软件崩溃, 厘清崩溃的错误原因和修复方法, 从而有效地预防和处理崩溃. 虽然软件崩溃不断得到人们的重视, 但该研究领域所遇到的挑战依然严峻, 主要包括崩溃问题自身的复杂性和崩溃信息记录的不完善性^[5,6].

- **崩溃问题自身的复杂性.** 软件崩溃作为软件故障的一种严重表现形式, 比起需求违背更加复杂. 由于发生崩溃时程序并未执行正确的路径, 崩溃现场难以提供可解释性的信息便于程序员理解和分析, 程序员不得不投入大量的精力对代码和报错信息进行分析. 这对程序员来说是一个巨大的挑战.

- **崩溃信息记录的不完善性.** 一方面, 在用户软件出现崩溃时, 程序员很难远程获取到详细的客户机运行环境; 另一方面, 在项目开发过程中, 计算资源受限导致部分崩溃日志信息出现错误或者不完善. 因此, 程序员面对这些不完善的崩溃信息时, 很难理解和分析这些软件崩溃, 进而很难去重现、定位, 以及修复这些崩溃.

本文将依次介绍软件崩溃的基本研究近况. 第 2 节介绍软件崩溃的基本概念及相关术语, 第 3~5 节依次介绍软件崩溃的 3 个方面的研究进展, 即崩溃分析、崩溃重现、崩溃定位与修复, 第 6 节总结全文并展望未来的研究方向.

2 基本概念及术语

本节介绍了软件崩溃和崩溃报告系统中的基本概念和相关术语. 我们首先以真实崩溃实例来展示产生软件崩溃的过程, 接着描述管理人员如何通过崩溃报告系统来处理和管理这些崩溃.

2.1 软件崩溃

软件崩溃 (software crash) 指的是程序的意外中断, 它是软件故障的一种严重表现形式^[3]. 在项目实际开发和维护场景中, 软件崩溃屡见不鲜, 由于崩溃本身的复杂性和崩溃信息的不完善性, 修复崩溃往往需要耗费程序员大量的精力和时间.

以真实项目为例, Apache Commons Lang¹⁾ 是 Apache 社区中的著名开源项目, 它为 Java 开发者提供了一系列有用的工具类, 实现了数值转化、字符串操作、对象序列化等功能. 其中 NumberUtils 类的 createNumber() 函数的功能是将输入字符串转化为相应的数字, 且该函数会根据输入字符串的位数将数字划分为不同类型 (如整数型、大整数型、长整数型等). 但程序员发现当输入字符串 “0x80000000” 时, 函数 createNumber() 出现了数值转化异常的崩溃, 即 NumberFormatException. 这个崩溃被程序员记录在第 Lang-747 号 bug 报告²⁾ 中. 程序员在 bug 报告中提交了新的测试函数 testLang747(), 并在该函数的第 256 行的断言处重现了这个崩溃.

1) Apache Commons Lang. <http://commons.apache.org/proper/commons-lang/>.

2) Bug 报告 Lang-747. <http://issues.apache.org/jira/browse/LANG-747>.

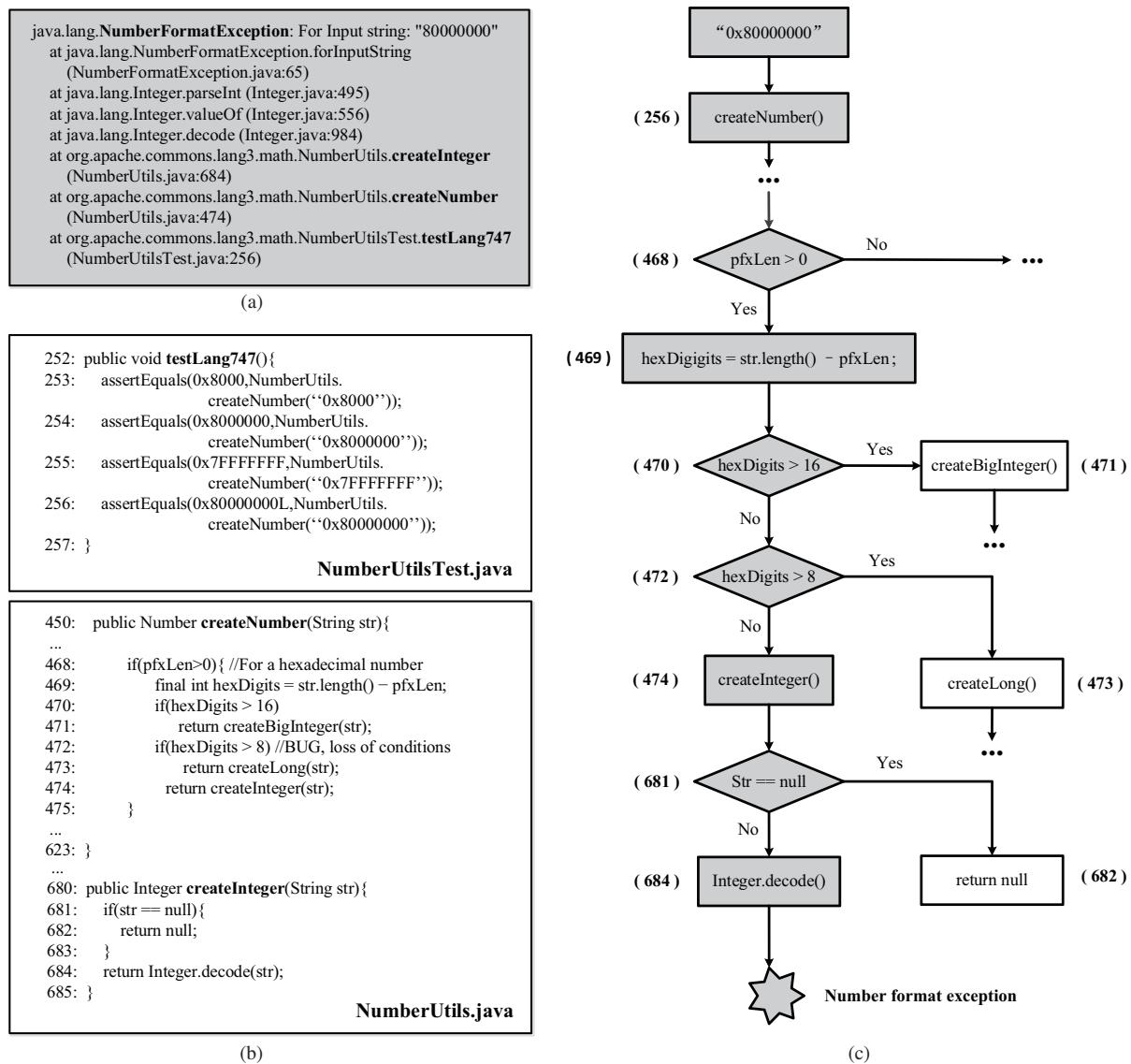


图 1 Apache Commons Lang 项目的 747 号崩溃的代码片段

Figure 1 Code snippet of crash Lang-747 in the Apache Commons Lang project. (a) Stack trace of Lang-747; (b) code snippet of Lang-747; (c) control flow graph of the code snippet (a number in a bracket denotes a line ID)

图 1 展示了 Lang-747 号崩溃的相关信息。其中图 1(a) 为崩溃的栈踪迹。栈踪迹 (stack trace) 提供了软件崩溃时所记录下来的报错信息，里面包含了崩溃异常类型、崩溃位置、函数调用次序等信息^[7]。图 1(b) 为崩溃的相关代码。由图可知崩溃的相关代码是 createNumber() 和 createInteger() 函数。图 1(c) 为程序的控制流图，无关部分已省去。控制流图 (control flow graph, CFG) 描述了程序所有可达的执行路径。它的根节点是程序的起点，叶子节点是程序的终点。控制流图为程序员提供了一个宏观的程序框架。

程序员在修复此崩溃时，首先根据栈踪迹信息 (图 1(a)) 得出：异常类型为 NumberFormatException，触发崩溃的输入是字符串 “0x80000000”，崩溃发生的位置在 createInteger() 函数的第 684 行，程序员于是提交了这个崩溃，并上传了重现方法，即 testLang747()。因此，重现这个崩溃需要执行 test-

Lang747() → createNumber() → createInteger(). 然后, 程序员通过分析相关代码 (图 1(b)) 和程序控制流图 (图 1(c)) 来判断出崩溃的可能的执行路径, 即图中的灰色节点的路径部分. 最后, 程序员分析得出崩溃的根源在 createNumber() 函数的第 472 行, 即当输入字符串为 “0x80000000” 时, 由于此处的 if 语句的判断条件不完善, 没能考虑到 “0x80000000” 这一种情况, 导致程序没能跳转到第 473 行调用 createLong() 函数将该字符串转化为长整型, 而是跳转到第 474 行调用 createInteger() 函数强行将该字符串转化为整型, 引发了数值转换异常的崩溃. 由上述过程知, 处理崩溃是一件极其耗时的工作. 即使我们知道崩溃触发的条件, 仍需要综合代码和栈踪迹的信息来分析出崩溃的位置进而修复崩溃. 在实际开发中, 崩溃的触发条件 (即崩溃重现场景) 是难以获知的, 这给崩溃处理带来极大的挑战.

2.2 崩溃报告系统

崩溃报告 (crash report) 是记录软件崩溃的信息文档, 包含有崩溃现场环境及崩溃栈踪迹等信息^[8]. 崩溃报告系统 (crash reporting system) 是一个自动化收集和管理崩溃报告的系统^[5,9]. 软件公司通过建立完善的崩溃报告系统来有效地处理和修复崩溃. 如今, 绝大多数现代软件都设有专属的崩溃报告系统. 当软件出现崩溃的时候, 该系统能迅速收集用户实时信息, 并自动形成崩溃报告上传至服务器. 崩溃报告系统能有效地应对和管理软件崩溃. 程序员通过对崩溃报告进行分析并完成修复补丁的提交, 及时挽回损失.

著名的崩溃报告系统有 Microsoft 公司的 Windows 错误报告系统 WER (Windows Error Reporting)^[10], Apple 公司的报错系统 Apple Crash Reporter^[11], Mozilla 公司的崩溃报错系统 Mozilla Crash Reporter^[12], 以及 NetBeans 的崩溃报系统^[13], 这些系统能快速准确地处理用户提交的崩溃信息, 且处理流程大体类似. 如图 2 所示, 当用户软件出现崩溃时, 崩溃报告系统将按照以下步骤收集、分类和指派崩溃报告.

崩溃收集. 此阶段由程序自动收集崩溃信息并发送给服务器. 当程序出现崩溃时, 首先会在后台自动收集用户的初始崩溃信息, 然后根据这些初始崩溃信息生成格式化的崩溃报告, 这些报告涵盖了崩溃位置、异常类型、函数调用等详细信息, 最后这些崩溃报告会被统一发送给服务器. 值得注意的是, 在发生崩溃后, 大多数软件会弹出一个崩溃提示窗口 (图 2 左下角), 在征求用户的同意后将这些崩溃信息上传.

崩溃分类. 此阶段由服务器将收集到的崩溃报告进行分类并生成相应的 bug 报告. 服务器接收到这些崩溃报告后会将其汇总到的总崩溃报告集 (图 2 右上角), 再按照崩溃位置将崩溃报告集进行分类, 也就是说相同崩溃报告类中的崩溃位置是相同的. 在进行分类后, 服务器会选出其中用户报错较多的类别, 生成对应的 bug 报告.

任务指派. 此阶段由软件管理者将 bug 报告指派给相关程序员来修复. 每个 bug 报告都是一组崩溃报告的合并, 管理者认为这一组崩溃的原因均是由这一个 bug 导致, 并指派给相关负责的程序员来修复 (图 2 底部). 因此当程序员修复了代码中的 bug, 就解决了这一组崩溃问题.

3 软件崩溃分析

鉴于软件崩溃的重要性和严重性, 近年来针对软件崩溃的分析研究不断涌现, 主要集中在崩溃报告的实证研究和崩溃的分类上. 崩溃报告的实证研究通过手工分析收集到的崩溃报告, 力图总结出触发崩溃的原因及修复崩溃的方法, 这一类工作以经验型研究为主; 崩溃的分类研究则希望提出更加合理的分类方法, 解决崩溃报告重复, 崩溃报告误分类等问题, 使得开发人员更加快速高效地处理这些崩

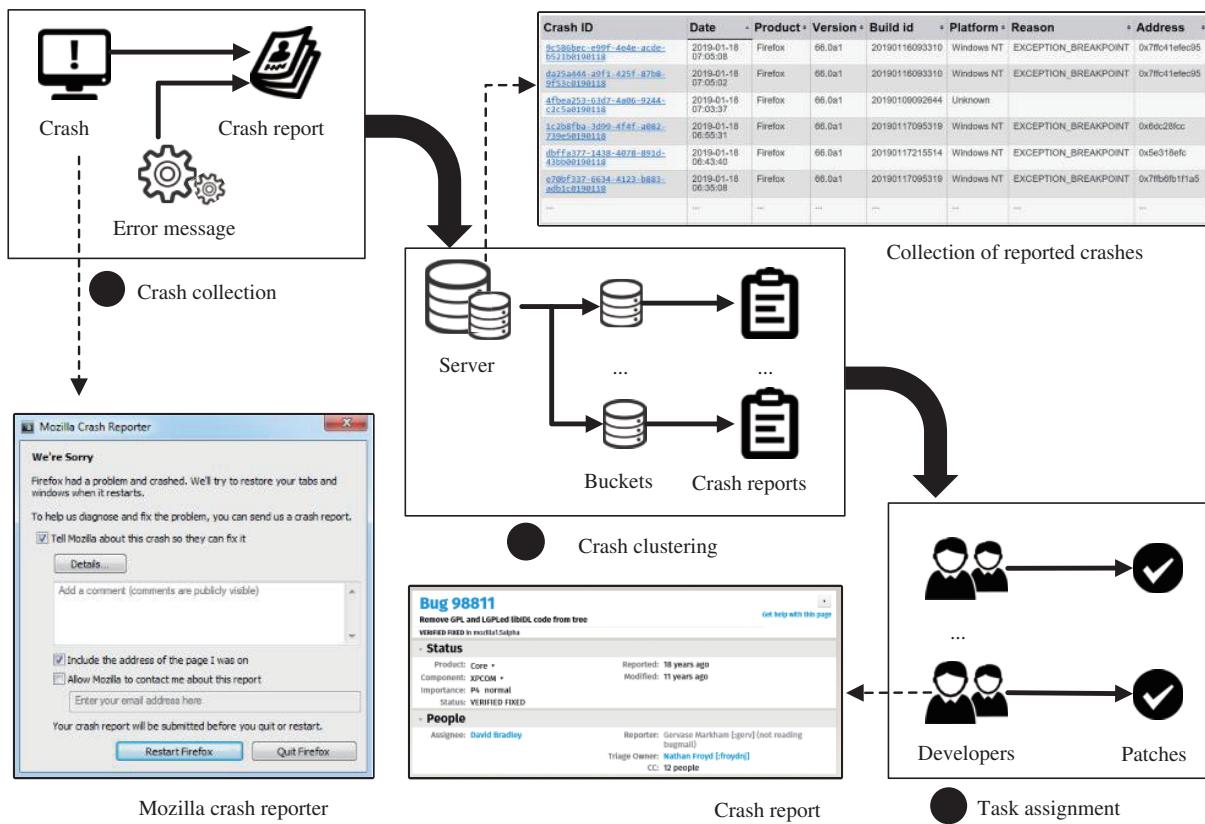


图 2 (网络版彩图) 崩溃报告系统工作流程

Figure 2 (Color online) Workflow of a crash reporting system

溃。崩溃分析面临的挑战有：

- 项目崩溃报告格式不一。** 虽然不同系统或产品提供的崩溃报告模板相似，但是细节部分仍然不统一，如报告的填写选项、报告关注的重点等，这些给崩溃分析方法的通用性造成了影响。
- 崩溃报告缺少关键信息。** 一些用户或程序员在提交崩溃报告时会漏写重要信息，如主机环境、崩溃的重现过程、崩溃的堆栈迹信息等，这为后期管理者对崩溃进行分类和管理造成了困难。
- 重复或无效报告盛行。** 用户在不知情的情况下反复提交重复的崩溃报告会浪费程序员的时间与精力，而由于填写错误导致的无效报告会增加管理者的负担，并且影响研究人员的分析结论。

3.1 崩溃报告的实证研究

Ganapathi 等^[14]对 Windows XP 操作系统的 2528 个系统崩溃进行了手工分析，他们发现：操作系统的崩溃绝大多数都是由于较差的硬件驱动代码造成的，而真正由于操作系统代码造成的崩溃只占总崩溃数的少数；提高这些硬件驱动代码的质量可以避免大约 75% 的系统崩溃。这一工作有助于提高开发人员对系统崩溃的理解。Kim 等^[15]对 Firefox 和 Thunderbird 的崩溃报告作了深入调研，他们发现大多数的崩溃报告中的崩溃都是由少数顶级崩溃 (top crash) 造成的，找到并修复这些顶级崩溃有助于提高程序员的工作效率。为此，他们提出了一种机器学习的方法，通过学习已发布的“顶级崩溃”的特征，从而预测出新版本的“顶级崩溃”。

Gu 等^[16]系统地分析了 Ant, AspectJ 和 Rhino 开源项目中的修复补丁引发潜在崩溃的现象, 他们提出了“覆盖”和“中断”两个度量标准来判断一个修复未来是否会引发潜在的软件崩溃。其中“覆盖”用来衡量修复后的程序对于非法输入的容忍程度, 而“中断”则用来衡量修复后的程序的实际行为与预期行为的偏差。他们基于此提出了 Fixation 方法来自动化的预测 Java 程序中较差的修复补丁, 从而避免为后续开发中的重复修改的工作。Seo 等^[17]分析研究了 Firefox 项目中被重新开启的崩溃 (recurring crashes), 提出一种自动化方法来预测崩溃报告在未来是否会被开启。具体地, 该方法首先将每个崩溃组中的崩溃报告按照其栈踪迹进行进一步的划分, 以保证划分后的每组崩溃报告都有相同的栈踪迹; 接着利用静态分析技术来合成程序完整的函数调用序列, 若修复的位置在调用序列中, 则认为该组崩溃报告“被覆盖”了, 若修复的位置在调用序列外, 则认为该组崩溃报告“未被覆盖”; 最后那些“未被覆盖”的崩溃报告会被预测为未来可能会被重新开启的崩溃。Fan 等^[6]分析并总结了 F-droid 数据集上 16245 个 Android 应用的崩溃的出错原因和修复模式。他们将这些崩溃按出错模块归纳为 17 个类, 进而总结出 11 个出错原因, 并利用归纳总结的信息对传统 Android 测试方法 Stoot^[18]进行了改进, 正确预测出了 Gmail 和 Google+ 项目上的 3 个潜在的真实错误。

3.2 崩溃的分类研究

Kim 等^[9]分析研究了微软公司的崩溃报告系统 WER 存在的两个问题: 即崩溃报告的重复性问题和崩溃修复不确定性问题, 他们提出了一种新的崩溃分类方法, 该方法通过构建崩溃图 (crash graph) 的方式来解决上述两个问题。崩溃图是栈踪迹构建出树状结构图, 其中每个结点代表栈踪迹中不同的函数, 每条边代表栈踪迹中的函数调用关系, 函数出现次数和函数调用次数被作为图中结点和边的权重。崩溃图方法首先根据崩溃报告构建出每组崩溃的崩溃图, 接着通过比较不同崩溃组的崩溃图来判断两组崩溃是否是重复, 并通过崩溃报告的特征来进一步预测该崩溃在未来是否可修复。

Dang 等^[19]着重分析了系统崩溃报告分类过程中存在的两类问题: (a) 第二桶问题 (the second bucket problem), 即相同原因引起的崩溃被误分类到不同的崩溃类别中; (b) 长尾问题 (the long tail problem), 即崩溃报告集中在少数类中, 其余多数桶中包含的崩溃报告数量较少。对此他们提出了基于栈踪迹相似度的分类方法 ReBucket 来对崩溃报告进行更加精准的分类, 从而节省程序员的宝贵精力。Tonder 等^[20]同样针对崩溃报告分类问题提出了新的分类方法: 语义崩溃桶 (semantic crash bucketing), 该方法利用轻量级的程序转换技术结合补丁模板来自动化生成可能的程序补丁, 并以此来进行崩溃报告的分类。

4 软件崩溃重现

崩溃重现 (crash reproduction) 指的是根据已有的报错信息生成崩溃的执行路径从而重现指定崩溃的过程^[21]。给定一段有缺陷的代码和对应的崩溃报错信息, 崩溃重现旨在找到触发给定崩溃的执行路径, 帮助程序员理解重现崩溃的具体步骤。

崩溃重现是关键且富有挑战的, 一方面, 重现任务是后续修复的基础。如果不能重现崩溃, 则程序员无法修复软件的缺陷。而只有重现了崩溃, 程序员才可能快速知晓崩溃的原因并及时进行修复。在某些开源项目上, 重现用户提交的崩溃往往要消耗数十天甚至几个月的时间, 而修复已被重现的崩溃只需要几天甚至几分钟^[21]。另一方面, 崩溃信息的不完整或缺失给程序员的重现工作带来巨大困难, 有些崩溃长年保持着无法重现的状态^[22]。具体来讲, 崩溃重现面临的挑战有:

- 程序中不确定因素增加重现的难度。程序中出现随机数生成、内存地址分配等因素时, 程序员

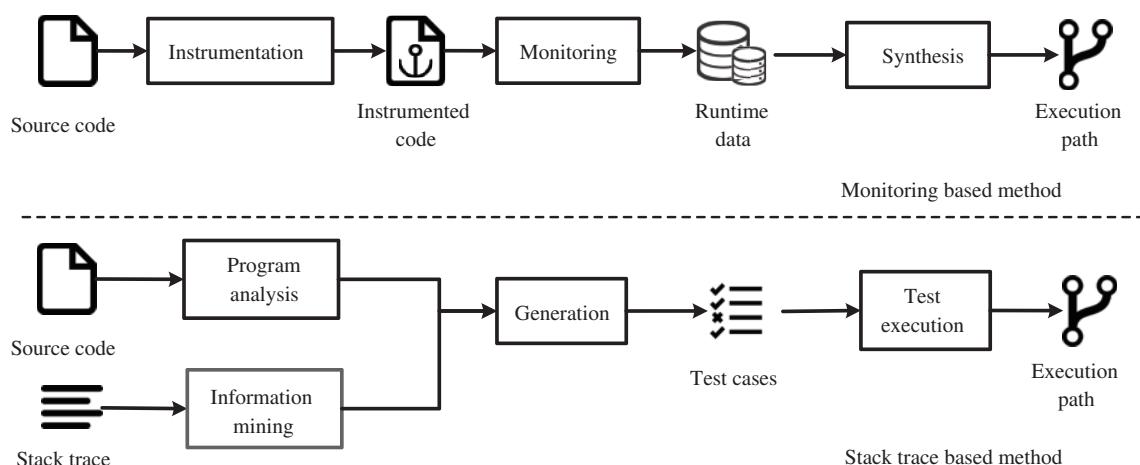


图3 崩溃重现的两类基本方法

Figure 3 Two categories of methods for crash reproduction

很难重现崩溃发生时的现场环境.

- 远程崩溃日志容易遗漏有效信息. 崩溃往往都是在客户端发生, 程序员无法获取触发该崩溃的确切操作步骤, 只能靠人工思考来假设崩溃的场景.

现有的崩溃重现的方法大体分为两种 (见图 3): 一种是基于监听的崩溃重现方法, 该方法依赖于程序执行时的执行信息; 另一种是基于栈踪迹的崩溃重现方法, 该方法依赖于崩溃发生后的栈踪迹信息. 两种方法各有优劣, 在实际开发过程中可根据目标和要求灵活运用.

4.1 基于监听的崩溃重现方法

基于监听的崩溃重现方法, 也叫“记录 – 回放”方法, 该类方法通过程序插桩的方式收集程序运行过程中的详细信息, 并利用此信息来合成崩溃的执行路径, 从而完成重现任务. 程序插桩指的是在几乎不影响程序功能的情况下, 向程序中植入代码以记录程序执行过程中的各种信息. 早期研究者记录的是系统内存或者编译器底层的执行信息, 导致方法的通用性不强且计算开销过大. 在后续研究中, 研究者常使用的记录信息包括变量值、参数值、函数调用序列等. 虽然基于监听的方法理论上能确定地重现任意崩溃, 但是在实际开发中需要对程序进行插桩并实时监听程序的各种行为. 这往往会产生巨大的资源消耗, 极大地影响了程序的性能, 有时甚至导致程序无法正常运行.

Steven 等^[23]提出了较早的基于监听的方法的框架 jRapture, 该方法仅针对于 Java 项目的崩溃进行监听和重现. jRapture 修改了 Java 开发工具包 (JDK), 因此在项目执行过程中, 程序员能够监听到 Java 程序与系统的交互信息, 并以此信息来对产生的崩溃进行重现. 但该方法不够通用且不适合处理发生在大型复杂的 Java 项目中的崩溃. Narayanasamy 等^[24]提出更加底层的方法 BugNet. BugNet 方法记录了系统的寄存器的状态改变信息和内存读写的数据信息, 因此它适用于更多编程语言. 但同时该方法也存在缺陷, 一旦程序与系统的交互变得复杂和频繁, BugNet 方法记录的信息呈爆炸式增长, 这种开销让 BugNet 和原程序都无法正常运行. Clause 等^[25]设计了崩溃重现的原型工具 ADDA, 该工具通过记录用户的操作过程以及操作环境来重现上报的软件崩溃. 作者随后在邮件通讯软件 PINE 上对方法进行了验证, 结果表明 ADDA 工具能够全部记录用户的各种信息并且重现了 PINE 上给定的所有崩溃. 但 ADDA 同样面临着计算开销过大的问题. 较近的且具有代表性的重现方法是 Artzi 等^[21]提出的 ReCrash, 该方法首先利用插桩技术记录程序运行时的函数调用顺序及对应参数等信息,

一旦程序发生崩溃, 则利用收集到的信息生成相应的测试用例来触发崩溃。同时, ReCrash 也使用了一系列手段来降低插桩对于程序性能的影响, 如监听时仅保存参数的引用地址, 不监听空的和简单的函数的执行信息等。实验证明这些手段确实能够降低系统内存的使用和开销。在后续的工作中, 他们基于 ReCrash 的思想, 设计并开发了针对 Java 代码的崩溃重现的原型工具 ReCrashJ^[26]。Jin 等^[27]提出了基于符号执行的崩溃重现方法 BugRedux。该方法同样首先收集程序的执行数据信息, 然后据此信息模拟出可能的执行路径来触发崩溃。在模拟执行路径中, BugRedux 使用符号执行技术模拟出可能的测试输入, 再结合 ad-hoc 搜索策略来生成测试输入来模拟崩溃的执行路径。区别于先前基于监听的方法, BugRedux 和 ReCrash 方法记录的都是程序运行过程中的状态信息而非系统底层或经编译得到的信息, 因此, BugRedux 和 ReCrash 更加轻量级且更容易实现。

4.2 基于栈踪迹的崩溃重现方法

基于栈踪迹的崩溃重现方法不再对程序进行实时监听, 转而分析崩溃时产生的栈踪迹信息。由于栈踪迹中包含了大量的崩溃相关信息, 如崩溃位置、崩溃异常类型、函数调用次序等, 因此基于栈踪迹的崩溃重现方法的主要思想是: 利用对栈踪迹的崩溃信息挖掘和对代码的静态程序分析^[28] 来生成测试用例来执行可能的崩溃执行路径, 从而重现崩溃。虽然这种方法能节省不少的资源开销, 但是由于栈踪迹信息本身的不完整性以及程序的复杂性, 该方法很难准确地模拟出真实的崩溃路径, 即该方法不能确定地重现软件崩溃。

Rößler 等^[29] 针对插桩技术的开销问题提出了新的崩溃重现方法 ReCore。由于 ReCore 不会对程序进行插桩操作, 因此在程序运行过程中实现了“零开销”。具体地, ReCore 首先通过操作系统收集程序崩溃时的内存中的堆栈信息 (core dump), 然后据此信息结合遗传测试工具 EvoSuite^[30] 来生成能够触发崩溃的测试集。ReCore 摆脱了重现工具对于插桩的依赖, 降低了运行开销, 并且首次将遗传测试用例生成的思想带入到崩溃重现领域来。Cao 等^[31] 提出了基于符号执行的方法 SymCrash, 该方法有选择地对函数进行监听并记录其状态信息, 解决了符号执行中部分函数无法求解的问题, 让重现概率大大提升。Chen 等^[32] 提出了基于栈踪迹和符号执行的崩溃重现方法 Star, 该方法通过栈踪迹预处理、初始条件推理、前置条件计算和测试用例生成这 4 个步骤来重现软件崩溃。由于 Star 能快速推演出空指针异常 (null pointer exception) 和数组越界异常 (array index out of bounds exception) 的前置条件, 因此能有效解决对应崩溃的路径爆炸问题。实验证明, Star 方法能够重现 52 个真实世界的崩溃中的 31 个, 重现率很高。Xuan 等^[33] 提出了基于测试用例变异的崩溃重现技术 MuCrash, 该方法通过测试用例筛选、测试断言去除和测试用例变异这 3 个步骤来重现软件崩溃。在测试用例筛选过程中, 通过执行已有测试集, 筛选出覆盖了栈踪迹中的类的测试用例, 并构成候选测试用例集。在测试断言去除过程中, 去除每个候选测试用例中的断言语句 (即 Assert 语句)。在测试用例变异过程中, 利用事先定义好的 7 个变异算子对候选测试用例进行变异操作, 并形成新的测试集, 最后使用新的测试集对程序进行测试, 从而触发给定的崩溃。Soltani 等^[34, 35] 提出的基于遗传算法的崩溃重现方法 EvoCrash 是较新的研究成果, 该方法将测试用例看作种群, 将栈踪迹的相似度看作适应度指标, 通过不断的迭代进化最终搜索到能够触发特定崩溃的测试用例。不同于之前的 Star 和 ReCore 方法, EvoCrash 将重现问题转化为搜索问题, 进而找到满足约束的最优解, 即触发崩溃的测试用例。

4.3 崩溃重现相关工作

研究者对软件崩溃重现的相关工作进行了探讨, 如测试用例生成、重现难易预测等。Csallner 等^[36] 提出了针对 Java 程序的随机测试工具 JCrasher, 该工具首先分析 Java 类中的类型信息并用生成随机

表1 典型的崩溃重现方法的比较

Table 1 Comparison among typical crash reproduction methods

Method	Input	Technique	Potential drawback	Category
ReCrash ^[21]	Instrumented code	Selective monitoring	High computation consumption	Monitoring based reproduction
BugRedux ^[27]	Instrumented code	Symbolic execution	Some complex constraints are hard to solve	
ReCore ^[29]	Source code, core dump	Genetic algorithm	Some complex crashes are hard to reproduce	
SymCrash ^[31]	Source code, stack trace	Selective monitoring, symbolic execution	Some complex constraints are hard to solve	
Star ^[32]	Source code, stack trace	Symbolic execution	Some complex constraints are hard to solve	Stack trace based reproduction
MuCrash ^[33]	Source code, stack trace, test cases	Test case mutation	Some complex crashes are hard to reproduce	
EvoCrash ^[35]	Source code, stack trace, test cases	Genetic algorithm	Some complex crashes are hard to reproduce	

输入来检测类中的公用方法,尝试让程序发生崩溃,然后使用启发式方法来判断崩溃造成的原因是软件内部缺陷还是非法输入。JCrasher 原型工具已经开源并嵌入在 Eclipse 之中。在后续工作中^[37],他们结合了静态检查方法和测试方法,提出了新的测试工具 CnC。CnC 先通过静态检查工具 ESC/Java 推理出违背需求的抽象前置条件,再用 JCrasher 将这些抽象前置条件转化为随机测试用例来检测程序的故障。虽然 JCrasher 和 CnC 均是自动化测试方法,但它们的本质还是通过生成测试用例来触发程序的崩溃,因此可以看做是崩溃重现的工作。Gu 等^[38]提出了重现的难易程度预测问题,并针对此问题开发了 Crash Detector 原型工具,该工具通过机器学习的方法学习真实开发过程中的崩溃的 23 个相关特征来预测崩溃重现的难易程度,预测结果对于程序员合理分配调试精力和时间大有裨益。该工具能在真实崩溃数据集 Defects4J^[39] 上进行验证且效果不错。Mao 等^[40]提出了针对 Android 应用的基于搜索的多目标测试方法 Sapienz,该方法通过 NSGA-II 进化算法最终选择出那些代码覆盖率高,发现崩溃数目多,测试序列长度短的测试集。实验证明 Sapienz 方法能够触发 1000 个开源 Android 项目中的 558 个崩溃,并正确预测出了 14 个潜在的真实错误。表 1 展示了不同种类具有代表性的重现方法的比较。

5 软件崩溃定位与修复

崩溃定位与修复是软件调试中最为关键的两项任务,定位任务旨在于找到错误的根源,而修复任务旨在于生成正确的程序补丁。由于这两项任务较崩溃重现更为困难,因此涉及的相关工作较少,但它们仍具有广阔的发展空间。崩溃定位与修复面临的挑战有:

- **软件代码设计复杂。**现代软件的代码量庞大且参数众多^[41],程序员需要花费很多时间来理解这些代码以找出崩溃位置,修复崩溃则会变得更难。
- **崩溃根源的位置不在函数调用序列中。**在程序执行过程中,某些栈踪迹之外的函数仍有可能影

响执行过程, 从而造成软件崩溃. 仅依靠栈踪迹的信息, 程序员很难判断出崩溃根源的位置.

- **第三方库的影响.** 当程序调用了第三方库时, 崩溃可能不是由软件本身造成的, 而是由第三方库中的缺陷或接口调用错误造成的^[42]. 这种情况会增大定位的搜索范围, 增加程序员的工作量.

5.1 崩溃定位的研究

软件崩溃定位 (crash localization) 指的是根据已有的报错信息推测出崩溃根源的大概位置的过程^[5, 43]. 更准确地讲, 给定一段引发崩溃的代码和相对应的崩溃报错信息, 崩溃定位旨在找到触发崩溃根源的位置. 在发生崩溃时, 程序员一般按照“重现、定位、修复”的顺序来修复崩溃, 因此在进行崩溃定位之后, 基本可以确定崩溃的原因, 有助于快速修复崩溃. 在软件维护过程中, 即使知道触发崩溃的步骤, 但要想准确地定位崩溃还是十分困难的.

为了有效地帮助程序员进行崩溃定位, 一系列自动化崩溃定位的方法应运而生. Wu 等^[5]提出了函数级的崩溃定位方法 CrashLocator, 该方法首先通过静态分析和栈踪迹拓展技术生成可能的触发崩溃的执行路径, 然后根据自定义的可疑值计算公式算出每个函数出错的概率, 并得到最终推荐排名. 该方法能有效地解决堆栈信息不完整问题, 使得执行路径趋于完整, 但是当程序变得复杂时, 栈踪迹拓展技术可能会导致路径爆炸问题. Wu 等^[44]对开源项目 NetBeans 的崩溃报告作经验性研究分析, 他们发现合理利用崩溃报告的信息能有效地分辨出质量较差的提交, 即引入崩溃的提交 (crash-inducing change). 他们提出了 ChangeLocator 方法来解决开发过程中的引入崩溃的提交的定位问题, 该方法将传统的代码特征结合崩溃报告的信息, 从而更加精准地预测哪些提交会引入崩溃, 在程序员提交代码的同时给出提示来辅助开发. Ren 等^[45]针对 Debian 操作系统的无法重现的构建 (unreproducible build) 的问题提出了错误定位方法 RepLoc, 该方法结合构建日志信息和启发式规则来找到导致构建无法重现的问题文件.

此外, 已有的基于频谱的故障定位技术 (spectrum-based fault localization) 也可以用于崩溃定位. 该技术基于统计的方法, 通过对测试用例的执行情况的计算 (suspicious formula, 可疑值计算公式), 推测出语句或函数出错的概率值, 从而定位故障根源的位置^[46]. 传统的方法包括 Tarantula^[47], Ochiai^[48], Jaccard^[49], Naish^[50] 等, 这些方法提出了不同的可疑值计算公式来进行计算. 为了进一步探究这些方法的性能好坏, Xie 等^[51]对传统定位技术的计算公式进行了理论上的推导和分析, 他们发现除了少数公式效果较优外, 大多数计算公式的效果其实是相同的, 这说明了不关注问题本质而盲目地设计新计算公式的做法对于故障定位没有太大意义.

由于准确的定位崩溃比较困难, Gu 等^[43]提出了一种崩溃分离方法 CraTer, 该方法通过机器学习的方法预测崩溃的根源是否在栈踪迹中. 方法框架如图 4 所示, 方法分为训练阶段和部署阶段. 在训练阶段, CraTer 对已有崩溃数据集进行建模, 通过挖掘发生崩溃的程序和崩溃栈踪迹中的 89 个特征, 并结合机器学习的方法建立预测模型 CraTer; 在部署阶段, CraTer 会对新来的崩溃进行预测, 判断其错误根源是否在栈踪迹中. 该工作能有效地帮助程序员分离崩溃的位置.

5.2 崩溃修复的研究

崩溃修复 (crash repair) 指的是利用自动化的方法生成补丁并修复程序崩溃的过程. 现有的基于测试的自动程序修复方法可以用于崩溃修复. 程序自动修复的主要流程框架如图 5 所示, 输入是一个出现崩溃的程序和一组规范程序正确行为的约束, 这类约束通常以测试用例集的形式给出, 输出则为能够满足该种约束的补丁^[52]. 自动程序修复方法大致分为 3 个阶段, 即故障定位阶段、补丁生成阶段和补丁评估阶段.

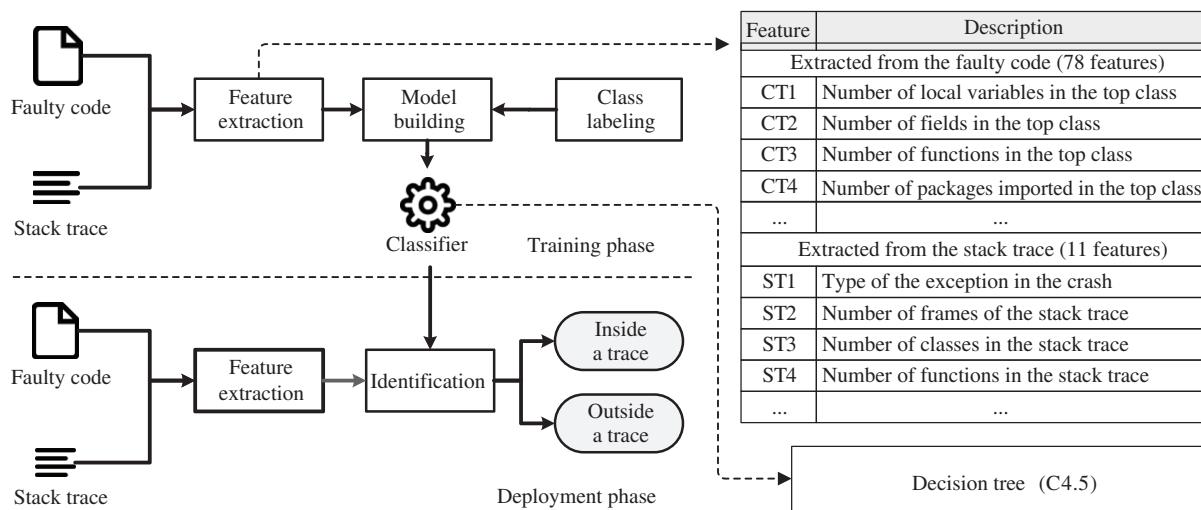


图4 崩溃分离方法 CraTer 的框架
Figure 4 Framework of a crash isolation method, CraTer

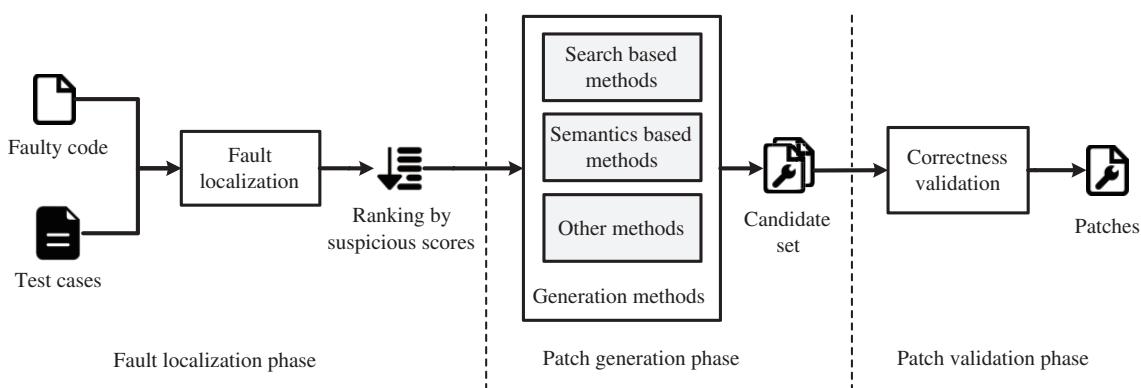


图5 自动程序修复的框架图
Figure 5 Framework of automatic program repair

目前自动生成补丁的方法可分为3种^[53],即基于搜索的方法、基于语义的方法和其他类型的方法。在补丁评估及选择阶段,根据预先定义好的评估指标之前生成的候选补丁进行评估,并选择出那些满足评估要求的补丁。

基于搜索的方法. 基于搜索的修复方法通过“生成–验证”模式在特定的搜索空间内通过启发式的算法生成补丁,并在配套的测试集上筛选出满足条件的补丁,其思想来源于基于搜索的软件工程(search based software engineering, SBSE)^[52]。具有代表性的方法包括Weimer等^[54]提出的GenProg,Kim等^[55]提出的PAR,Qi等^[56]提出的Kali,Xuan等^[57]提出的B-Refactoring,以及Le等^[58]提出的HDRepair。

基于语义的方法. 基于语义的修复方法通过分析程序的语义信息来合成补丁代码^[53]。具体地,该方法首先根据程序运行时信息提取出补丁要满足的约束,再利用约束求解器得到补丁,这些补丁由于在生成时已经满足了约束,故一般都会通过验证。具有代表性的方法包括Xuan等^[59]提出了Nopol

和 Xiong 等^[60] 提出的 ACS.

其他类型的方法. Gao 等^[61] 提出了一种全自动方法, 这种方法通过分析问答网站来解决重复出现的崩溃问题. 具体来说, 这种方法首先通过从栈踪迹中提取信息, 并根据这些信息检索问答网站得到一系列 Q&A 页面, 通过分析所得到的 Q&A 页面, 抽取这些页面问题贴中的错误代码和回答帖中的修复代码, 进而生成编辑脚本, 生成候选补丁. Tan 等^[62] 提出了一种针对 Android 应用的崩溃修复方法 Droix, 该方法利用总结出的 17 个错误原因提出了 8 个修复模式, 并结合基于搜索的方法逐步迭代进化, 直到产生出满足测试要求的补丁. 实验证明 Droix 能够找到 24 个真实崩溃中的 15 个, 并且生成的补丁质量较高. 其他的修复方法包括 Ke 等^[63] 提出的 SearchRepair, Tan 等^[64] 提出的 Relifix, 以及 Kaleeswaran 等^[65] 提出了 MintHint.

6 未来的机遇与挑战

软件崩溃研究作为软件开发及维护中的难点问题受到人们广泛关注. 为了有效地预防和应对突发的软件崩溃, 研究人员在崩溃分析、崩溃重现、崩溃定位与修复这 3 方面作了富有成果的研究. 在崩溃分析中, 研究人员通过对崩溃报告和崩溃分类的分析调研, 从而深入理解崩溃的成因及发生规律, 为程序员高效处理崩溃打下基础. 在崩溃重现中, 研究人员通过插桩或栈踪迹信息来生成崩溃的执行路径, 从而重现给定的崩溃. 人们将符号执行、静态分析、演化算法等方法与可用的崩溃信息综合分析, 力图在较少的性能开销上重现更多的崩溃. 在崩溃定位与修复中, 研究人员利用崩溃信息搜索出崩溃根源的可能的位置, 从而定位并修复给定的崩溃. 基于机器学习的、基于统计学的, 以及基于信息检索的新方法不断涌出, 为崩溃定位提供了更多的方向. 在未来, 此 3 方面工作机遇与挑战并存:

- 在崩溃分析的工作中, (a) 项目的崩溃报告格式不统一, (b) 人员提交的报告质量参差不齐, (c) 存在大量未经审核的重复或无用的报告, 调试人员很难掌握完整且准确的崩溃信息. 本文认为未来在开源生态环境中建立一个健全严格的崩溃管理准则是必要的, 如崩溃的提交和管理准则. 进一步, 应当在崩溃的管理和分类上尝试更多的自动化管理工具, 现阶段的手工管理不能有效地识别重复错误的崩溃报告, 引入智能化管理和分析软件崩溃是未来的一个可行化方案.

- 在崩溃重现的工作中, (a) 基于监听的重现方法受限于因插桩而造成的计算开销, (b) 基于堆栈迹的重现方法则受限于符号执行和演化算法的自身特点(如条件组合爆炸造成的搜索空间过大), 现有研究能够重现的真实的软件崩溃很少. 随着程序的复杂性和规模不断提高, 重现任务会变得更加艰难. 由于基于监听和基于堆栈迹的方法各有优缺点, 人们在设计重现方法时不应当全部依赖于监听信息或是全部依赖于堆栈迹信息. 有限度地监听程序状态信息是必要的, 高效的搜索算法同样也是必要的, 二者的结合是未来重现工作的发展方向.

- 在崩溃定位与修复的工作中, (a) 报错的位置可能不是引起崩溃的真正位置, (b) 崩溃可能由第三方库造成, 而现阶段的研究停留在恢复完整程序流图和分析故障代码特征的阶段. 这些技术并不能保证准确地找到崩溃的位置. 崩溃定位技术在未来有较好的发展潜力.

本文从软件崩溃分析、重现、定位与修复的角度, 系统地阐述了软件崩溃近期的研究进展, 并对其未来的机遇与挑战进行了展望. 软件崩溃研究十分重要, 现阶段成果较少. 未来发展中, 引入相关领域技术, 如深度学习、近似优化等算法, 可以为软件崩溃研究提供更为广阔探索空间.

参考文献

1 Mathur A. Foundations of Software Testing. Delhi: Pearson Education India, 2011

- 2 Pressman R S. Software Engineering: a Practitioner's Approach. New York: McGraw-Hill, 2010
- 3 Wikipedia. Crash (computing). 2013. [http://en.wikipedia.org/wiki/Crash_\(computing\)](http://en.wikipedia.org/wiki/Crash_(computing))
- 4 Kim J. Fatal Bugs: Disasters and Revelations from Software Defects. Seoul: Acornpub, 2014 [金钟河, 著. 叶蕾蕾, 译. 致命 Bug: 软件缺陷的灾难与启示. 北京: 人民邮电出版社, 2016]
- 5 Wu R X, Zhang H Y, Cheung S C, et al. CrashLocator: locating crashing faults based on crash stacks. In: Proceedings of International Symposium on Software Testing and Analysis, San Jose, 2014. 204–214
- 6 Fan L L, Su T, Chen S, et al. Large-scale analysis of framework-specific exceptions in android apps. In: Proceedings of the 40th International Conference on Software Engineering, Gothenburg, 2018. 408–419
- 7 Li Y H, Ying S, Jia X Y, et al. EH-recommender: recommending exception handling strategies based on program context. In: Proceedings of the 23rd International Conference on Engineering of Complex Computer Systems, Melbourne, 2018. 104–114
- 8 Mozilla. Understanding crash reports. 2017. http://developer.mozilla.org/en-US/docs/Mozilla/Projects/Crash-reporting/Understanding_crash_reports
- 9 Kim S, Zimmermann T, Nagappan N. Crash graphs: an aggregated view of multiple crashes to improve crash triage. In: Proceedings of International Conference on Dependable Systems and Networks, Hong Kong, 2011. 486–493
- 10 Glerum K, Kinshumann K, Greenberg S, et al. Debugging in the (very) large: ten years of implementation and experience. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles, Big Sky, 2009. 103–116
- 11 Apple. Technical note tn2123. 2010. <http://developer.apple.com/library/archive/technotes/tn2004/tn2123.html>
- 12 Mozilla. Mozilla crash reporters. 2016. <http://crash-stats.mozilla.com/>
- 13 NetBeans. Netbeans crash reporters. 2015. <http://netbeans.org/bugzilla/>
- 14 Ganapathi A, Ganapathi V, Patterson D A. Windows XP kernel crash analysis. In: Proceedings of the 20th Conference on Systems Administration, Washington, 2006. 149–159
- 15 Kim D, Wang X M, Kim S, et al. Which crashes should I fix first?: predicting top crashes at an early stage to prioritize debugging efforts. IEEE Trans Softw Eng, 2011, 37: 430–447
- 16 Gu Z, Barr E T, Hamilton D J, et al. Has the bug really been fixed? In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Cape Town, 2010. 55–64
- 17 Seo H, Kim S. Predicting recurring crash stacks. In: Proceedings of International Conference on Automated Software Engineering, Essen, 2012. 180–189
- 18 Su T, Meng G Z, Chen Y T, et al. Guided, stochastic model-based GUI testing of Android apps. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, Paderborn, 2017. 245–256
- 19 Dang Y N, Wu R X, Zhang H Y, et al. ReBucket: a method for clustering duplicate crash reports based on call stack similarity. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012. 1084–1093
- 20 van Tonder R, Kotheimer J, Le Goues C. Semantic crash bucketing. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, 2018. 612–622
- 21 Artzi S, Kim S, Ernst M D. ReCrash: making software failures reproducible by preserving object states. In: Proceedings of European Conference on Object-Oriented Programming, Paphos, 2008. 542–565
- 22 Joorabchi M E, MirzaAghaei M, Mesbah A. Works for me! characterizing non-reproducible bug reports. In: Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, 2014. 62–71
- 23 Steven J, Chandra P, Fleck B, et al. jRapture: a capture/replay tool for observation-based testing. In: Proceedings of International Symposium on Software Testing and Analysis, Portland, 2000. 158–167
- 24 Narayanasamy S, Pokam G, Calder B. BugNet: continuously recording program execution for deterministic replay debugging. In: Proceedings of the 32nd International Symposium on Computer Architecture, Madison, 2005. 284–295
- 25 Clause J, Orso A. A technique for enabling and supporting debugging of field failures. In: Proceedings of the 29th International Conference on Software Engineering, Minneapolis, 2007. 261–270
- 26 Artzi S, Kim S, Ernst M D. ReCrashJ: a tool for capturing and reproducing program crashes in deployed applications. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering, Amsterdam, 2009. 295–296
- 27 Jin W, Orso A. BugRedux: reproducing field failures for in-house debugging. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012. 474–484

- 28 Zhang J, Zhang C, Xuan J F, et al. Recent progress in program analysis. *J Softw*, 2019, 30: 80–109 [张健, 张超, 玄
跻峰, 等. 程序分析研究进展. *软件学报*, 2019, 30: 80–109]
- 29 Rößler J, Zeller A, Fraser G, et al. Reconstructing core dumps. In: Proceedings of the 6th IEEE International
Conference on Software Testing, Verification and Validation, Luxembourg, 2013. 114–123
- 30 Fraser G, Arcuri A. EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the
19th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Szeged, 2011. 416–419
- 31 Cao Y, Zhang H Y, Ding S. SymCrash: selective recording for reproducing crashes. In: Proceedings of the International
Conference on Automated Software Engineering, Vasteras, 2014. 791–802
- 32 Chen N, Kim S. STAR: stack trace based automatic crash reproduction via symbolic execution. *IEEE Trans Softw
Eng*, 2015, 41: 198–220
- 33 Xuan J F, Xie X Y, Monperrus M. Crash reproduction via test case mutation: let existing test cases help.
In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, Bergamo, 2015. 910–913
- 34 Soltani M, Panichella A, van Deursen A. Evolutionary testing for crash reproduction. In: Proceedings of the 9th
International Workshop on Search-Based Software Testing, Austin, 2016
- 35 Soltani M, Panichella A, van Deursen A. A guided genetic algorithm for automated crash reproduction. In: Proceedings
of the 39th International Conference on Software Engineering, Buenos Aires, 2017. 209–220
- 36 Csallner C, Smaragdakis Y. JCrasher: an automatic robustness tester for Java. *Softw-Pract Exper*, 2004, 34: 1025–1050
- 37 Csallner C, Smaragdakis Y. Check ‘n’ crash: combining static checking and testing. In: Proceedings of the 27th
International Conference on Software Engineering, St. Louis, 2005. 422–431
- 38 Gu Y F, Xuan J F, Qian T Y. Automatic reproducible crash detection. In: Proceedings of the International Conference
on Software Analysis, Testing and Evolution, Kunming, 2016. 48–53
- 39 Just R, Jalali D, Ernst M D. Defects4J: a database of existing faults to enable controlled testing studies for java
programs. In: Proceedings of International Symposium on Software Testing and Analysis, San Jose, 2014. 437–440
- 40 Mao K, Harman M, Jia Y. Sapienz: multi-objective automated testing for android applications. In: Proceedings of
the 25th International Symposium on Software Testing and Analysis, Saarbrücken, 2016. 94–105
- 41 Xuan J F, Gu Y F, Ren Z L, et al. Genetic configuration sampling: learning a sampling strategy for fault detection
of configurable systems. In: Proceedings of Genetic and Evolutionary Computation Conference Companion, Kyoto,
2018. 1624–1631
- 42 Kechagia M, Mitropoulos D, Spinellis D. Charting the API minefield using software telemetry data. *Empir Softw Eng*,
2015, 20: 1785–1830
- 43 Gu Y F, Xuan J F, Zhang H Y, et al. Does the fault reside in a stack trace? Assisting crash localization by predicting
crashing fault residence. *J Syst Softw*, 2019, 148: 88–104
- 44 Wu R X, Wen M, Cheung S C, et al. ChangeLocator: locate crash-inducing changes based on crash reports. *Empir
Softw Eng*, 2018, 23: 2866–2900
- 45 Ren Z L, Jiang H, Xuan J F, et al. Automated localization for unreproducible builds. In: Proceedings of the 40th
International Conference on Software Engineering, Gothenburg, 2018. 71–81
- 46 Xuan J F, Monperrus M. Test case purification for improving fault localization. In: Proceedings of the 22nd ACM
SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, 2014. 52–63
- 47 Jones J A, Harrold M J, Stasko J T. Visualization of test information to assist fault localization. In: Proceedings of
the 24th International Conference on Software Engineering, Orlando, 2002. 467–477
- 48 Abreu R, Zoeteweij P, van Gemund A J C. An evaluation of similarity coefficients for software fault localization.
In: Proceedings of the 12th IEEE Pacific Rim International Symposium on Dependable Computing, Riverside, 2006.
39–46
- 49 Abreu R, Zoeteweij P, van Gemund A J C. On the accuracy of spectrum-based fault localization. In: Proceedings of
Testing: Academic and Industrial Conference Practice and Research Techniques – MUTATION, Windsor, 2007. 89–98
- 50 Naish L, Lee H J, Ramamohanarao K. A model for spectra-based software diagnosis. *ACM Trans Softw Eng Methodol*,
2011, 20: 1–32
- 51 Xie X, Chen T Y, Kuo F C, et al. A theoretical analysis of the risk evaluation formulas for spectrum-based fault
localization. *ACM Trans Softw Eng Methodol*, 2013, 22: 1–40

- 52 Xuan J F, Ren Z L, Wang Z Y, et al. Progress on approaches to automatic program repair. *J Softw*, 2016, 27: 77–784
[玄跨峰, 任志磊, 王子元, 等. 自动程序修复方法研究进展. 软件学报, 2016, 27: 771–784]
- 53 Wang Z, Gao J, Chen X, et al. Automatic program repair techniques: a survey. *Chinese J Comput*, 2018, 41: 588–610
[王贊, 郭健, 陈翔, 等. 自动程序修复方法研究评述. 计算机学报, 2018, 41: 588–610]
- 54 Le Goues C, Nguyen T V, Forrest S, et al. GenProg: a generic method for automatic software repair. *IEEE Trans Softw Eng*, 2012, 38: 54–72
- 55 Kim D, Nam J, Song J, et al. Automatic patch generation learned from human-written patches. In: Proceedings of the 35th International Conference on Software Engineering, San Francisco, 2013. 802–811
- 56 Qi Z C, Long F, Achour S, et al. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings of International Symposium on Software Testing and Analysis, Baltimore, 2015. 24–36
- 57 Xuan J F, Cornu B, Martinez M, et al. B-refactoring: automatic test code refactoring to improve dynamic analysis. *Inf Softw Technol*, 2016, 76: 65–80
- 58 Le X D, Lo D, Le Goues C. History driven program repair. In: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering, Osaka, 2016. 213–224
- 59 Xuan J F, Martinez M, DeMarco F, et al. Nopol: automatic repair of conditional statement bugs in java programs. *IEEE Trans Softw Eng*, 2017, 43: 34–55
- 60 Xiong Y F, Wang J, Yan R F, et al. Precise condition synthesis for program repair. In: Proceedings of the 39th International Conference on Software Engineering, Buenos Aires, 2017. 416–426
- 61 Gao Q, Zhang H S, Wang J, et al. Fixing recurring crash bugs via analyzing Q&A sites. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, Lincoln, 2015. 307–318
- 62 Tan S H, Dong Z, Gao X, et al. Repairing crashes in Android apps. In: Proceedings of the 40th International Conference on Software Engineering, Gothenburg, 2018. 187–198
- 63 Ke Y, Stolee K T, Le Goues C, et al. Repairing programs with semantic code search. In: Proceedings of the 30th International Conference on Automated Software Engineering, Lincoln, 2015. 295–306
- 64 Tan S H, Roychoudhury A. Relifix: automated repair of software regressions. In: Proceedings of the 37th International Conference on Software Engineering, Florence, 2015. 471–482
- 65 Kaleeswaran S, Tulsian V, Kanade A, et al. Minthint: automated synthesis of repair hints. In: Proceedings of the 36th International Conference on Software Engineering, Hyderabad, 2014. 266–276

Progress on software crash research

Yongfeng GU¹, Ping MA¹, Xiangyang JIA¹, He JIANG² & Jifeng XUAN^{1*}

1. School of Computer Science, Wuhan University, Wuhan 430072, China;

2. School of Software, Dalian University of Technology, Dalian 116621, China

* Corresponding author. E-mail: jxuan@whu.edu.cn

Abstract Software crashes represent unexpected program interrupts that manifest as software faults and can be dangerous in that their frequent occurrence diminishes user experience, can damage the reputation of a company, and potentially cause significant losses to stakeholders. The increasing scale and complexity of modern software requires methods for preventing/handling software crashes. Here, we briefly review and summarize progress in three areas of the research field associated with handling software crashes: crash analysis, crash reproduction, and crash localization and repair.

Keywords software crashes, crash analysis, crash reproduction, crash localization, crash repair, program debugging, program exception handling



Yongfeng GU was born in 1992. He received a B.S. degree from Hubei University, Wuhan, China, in 2015. He is currently a Ph.D. candidate at the School of Computer Science, Wuhan University. His research interests include software testing and debugging, software crashes, and configuration analysis.



Ping MA was born in 1998. She received her B.S. degree from Wuhan University, Wuhan, China, in 2019. She is currently a graduate student at the School of Computer Science, Wuhan University. Her research interests include software testing and debugging and mining software repositories.



Xiangyang JIA was born in 1972. He received his Ph.D. degree in computer software and theory from Wuhan University, Wuhan, China, in 2008. He is currently a lecturer at Wuhan University. His research interests include program analysis, symbolic execution, and AI-driven software engineering.



Jifeng XUAN was born in 1984. He received his Ph.D. degree from Dalian University of Technology, Dalian, China, in 2013. He is currently a professor at Wuhan University. His research interests include software testing and debugging, software data analysis, and search-based software engineering.