摘要

软件崩溃是一种表现较为严重的软件缺陷，它会给开发人员和使用者带来较为严重的不良后果。如今，软件崩溃报告收集系统被广泛的部署以帮助开发人员去发现软件缺陷。但开发人员仍需花费大量时间去在代码版本控制系统中数以万计的版本里寻找引入软件缺陷的版本。在本论文中，我们将提出一种算法，通过分析从软件崩溃报告和代码版本控制系统里面收集到的信息，向软件开发人员提供一个可疑崩溃缺陷代码引入版本列表，去帮助开发人员通过版本变更情况，快速的定位并修复崩溃缺陷。最后，我们使用了来自 Mozilla 的真实产品数据去评估我们的算法。评估结果显示，我们的算法有效。通过检查前 1、前 5 和前 10 个结果，分别能成功定位 53%、59%和 77%的缺陷引入版本。

关键词 缺陷定位 崩溃定位 引入版本 崩溃报告

Abstract

Software crashes are among the most severe manifestations of software faults and may cause serious consequences. Nowadays, crashing report systems are widely deployed to help developers fix bugs. However, it takes huge efforts to find the crash inducing revision among a huge amount of revisions in the control version system. In this paper, we propose RevisionLocator, which uses information from crash reports and control version system to recommend developers a ranked list of suspicious bug inducing revisions, facilitating debugging with the context of development progress. We use real world data from Mozilla to evaluate our approach. The result shows that our approach is effective. RevisionLocator successfully locates 53%, 59% and 77% of crashing faults by examining top 1, 5, and 10 revisions.

Keywords bug localization, crash localization, bug-inducing revision, crash reports

目录

摘要 I
Abstract II
第 1 章 绪论 1
1.1 课题背景 1
1.2 本文研究目标和内容 3
1.3 本文结构安排 3
第 2 章 文献综述 5
2.1 领域背景 5
2.2 领域现状 5
2.2.1 崩溃报告采集 6
2.2.2 收集分类优化 7
2.2.3 函数级别定位 7
2.3 本章小结 8
第 3 章 研究方案 9
3.1 扩展崩溃堆栈 10
3.2 分析候选语句 12
3.3 计算并排序可疑度 15
3.3.1 特征 1： 函数的代码行数 (FLOC) 15
3.3.2 特征 2： 到函数崩溃点的距离 (IAD) 15
3.3.3 特征 3： 候选语句与崩溃点关联度 (CRD) 16
3.3.4 特征 4： 函数出现频率 (FF) 17
3.3.5 特征 5： 版本出现频率 (RF) 17
3.3.6 整体合并 18
第 4 章 实验结果 19
4.1 实验数据集 19
4.2 度量函数 20

4.2.1 Recall@N 20
4.2.2 MRR 20
4.3 实验结果 20
有多少错误可以被成功定位 20
第 5 章 分析与讨论 22
5.1 算法工作原理 22
5.2 影响实验的因素 23
5.2.1 实验样本选择 23
5.2.2 程序调用图的局限性 24
5.3 优势和应用 24
第 6 章 本文总结 26
参考文献 28
致谢 31

第1章 绪论

1.1 课题背景

依据最近几年发表在软件工程领域相关国际会议上的部分实证研究所论述，程序崩溃（Software Crash）当下已成为一种严重的软件故障[5]。它会大幅度的影响软件的可用性和稳定性，不仅仅会给用户造成不友好的软件使用体验，降低用户的工作效率，而且还会造成用户当前尚未完全保存至非易挥发存储介质当中的工作数据丢失，给用户造成经济上的损失。伴随着人们日常生活对于计算机软件依赖程度的逐渐增长，软件的重要性越发被体现，这也使得软件开发维护人员需要提供更高质量的软件后续支持。在遇到程序崩溃情况时，能够快速的定位引起程序崩溃的相关代码，并及时修复相关问题代码，以减少对软件使用者造成的不便，从而提升自身软件的市场竞争力。

程序崩溃是一种较难查找到故障代码且又较难重现的软件缺陷。鉴于软件开发人员在提交至软件代码仓库（Source Code Repository）之前，会设计足够的测试数据对新增加的软件功能或者修改过的软件模块进行较为充分的测试，大部分代码都会被测试数据覆盖到。所以，发生在用户生产环境中的程序崩溃一般会在较为深层的执行分支中（Branch），且一般会依赖用户的运行环境、输入数据等作为崩溃的触发条件，上述原因也可以作为解释程序崩溃较难重现的主要原因。目前，在软件行业的主导公司或开源软件组织，例如苹果、微软[13]、Mozilla Foundation，均已经开始建设并部署自己的程序崩溃报告收集系统（Crash Reporting System）。这些收集系统主要用于从本公司运行在用户计算机中的产品上自动化收集程序崩溃报告，并将这些报告根据崩溃的特征进行归类，将相同崩溃原因的程序崩溃报告放置在同一个软件崩溃原因集合(Bucket)当中。以 Mozilla Foundation 的程序崩溃报告收集系统为例，他们主要收集的崩溃信息包括软件在崩溃时的各个线程的堆栈信息、崩溃点的信息、运行的产品的发行版本及开发版本信息和运行的系统环境信息等。我们在图 1 中给出了一份真实的来自于 Mozilla Foundation 的软件崩溃报告收集系统上的程序崩溃报告的崩溃堆栈信息，以此帮助更多的读者理解。产品开发者可以通过这些被收集到的程序崩溃报告信息，凭借经验来修复软件漏洞。

层 (Frame)	函数、文件及行号
Frame 0	mozilla::layers::CompositorD3D11::HandleError(long, mozilla::layers::CompositorD3D11::Severity) gfx/layers/d3d11/CompositorD3D11.cpp:1105
Frame 1	mozilla::layers::CompositorD3D11::Failed(long, mozilla::layers::CompositorD3D11::Severity) gfx/layers/d3d11/CompositorD3D11.cpp:1117
Frame 2	mozilla::layers::CompositorD3D11::UpdateRenderTarget() gfx/layers/d3d11/CompositorD3D11.cpp:940
…	
Frame 16	base::Thread::ThreadMain() ipc/chromium/src/base/thread.cc:170
Frame 17	`anonymous namespace'::ThreadFunc(void*) ipc/chromium/src/base/platform_thread_win.cc:26

图 1 软件崩溃报告的崩溃堆栈示意图

根据我们对于开源软件社区的实证研究，发现仅仅只有收集到的程序崩溃报告的信息，对于软件开发人员或者软件维护人员定位引起程序崩溃的缺陷代码而言是帮助性不充足的。换而言之，对于软件开发人员需要花费较多的时间对软件缺陷代码引入版本进行定位。目前，比较常见的确定软件缺陷代码引入版本的方法是通过二分的方法来进行手工检测。具体而言来说，开发维护人员知道一个不存在某程序崩溃缺陷的版本和一个存在该程序崩溃缺陷的版本，至此，开发维护人员便有了一个版本区间，显然易见的，某一个落在这个版本区间的版本引入了该程序崩溃缺陷。开发维护人员只需要不断通过二分的方法，选取中间版本，并从软件代码仓库中提取出此版本的源代码，对源代码进行编译生成可执行文件，再按照相应步骤去检测软件是否会崩溃，依此不断缩小区间并最终确定缺陷引入版本，然后对代码进行修复。在我们的实证研究过程当中，我们也有发现这种方法目前正在被 Mozilla Foundation 所使用，他们通过提供 mozregression 工具，来帮助开发维护人员进行二分查找缺陷代码的引入版本。这也可以从侧面来

说明：1）软件开发维护人员会根据在程序崩溃报告收集系统中所收集到的数据，对引起程序崩溃代码的引入版本进行定位，反映出开发维护人员对基于程序崩溃报告来定位软件缺陷引入的版本有一定需求 2）目前软件开发维护人员正在使用二分编译并运行测试这一种较为费时费力的方法进行缺陷版本定位。

在上述课题背景之下，我们提出设计一套基于程序崩溃报告的针对缺陷代码引入版本的定位算法，并通过使用真实世界中较为流行的软件产品作为我们的实验对象，对我们所设计的定位算法进行有效性验证。我们希望借助此算法去帮助软件开发维护人员快速查找缺陷代码引入的版本及故障重现[7][8]，并为其修复软件缺陷提供软件开发的上下文信息。

1.2 本文研究目标和内容

本课题的主要研究目标是通过分析由软件崩溃报告收集系统所收集到的软蹦崩溃信息和软件源代码仓库中的代码修改记录，来定位引起软件崩溃缺陷的代码具体引入版本，帮助软件开发人员找到缺陷代码引入的开发上下文关系，并促进软件缺陷的修复进程。

本课题的主要研究任务是设计并实现相应的软件崩溃版本定位算法，设计合理的验证实验，证明所提出的算法的可行性，并从真实世界的软件产品中收集数据实验数据，来保证实验的客观性。

1.3 本文结构安排

本篇论文共分为六个章节，并将按照下述组织结构来陈述相关研究内容：

我们将在第一章节里阐述课题背景、研究目标和内容。

第二个章节则为文献综述，将阐述领域的相关背景、领域的现状以及对文献内容的总结。被综述的文献，均是选自与软件工程领域相关的国际会议和期刊所刊登的文章。

第三个章节为研究方案，本篇论文将在该章节内阐述所提出的崩溃引入版本定位算法的框架原型。整体算法被分为三个部分，分别为扩展崩溃堆栈、分析候选语句以及计算并排序可疑度。

第四个章节为实验结果，其中将阐述我们的实验数据集详细信息及相关的标准答案的获取依据。然后，我们将阐述用于度量实验结果的两个度量函数 Recall@N 和 MRR。最后一个子章节将呈现本论文所提出的算法的实验结果，说明本论文所提出的算法的崩溃引入版本定位的有效性。

第五个章节为分析与讨论，将分别阐述算法工作原理、影响实验的因素和本论文所提出的算法的优势和应用三个子问题，并对它们分别进行讨论。

最后一个章节即第六章为本文总结，将对本文进行概括性总结，回顾本章的要点内容。

第2章 文献综述

2.1 领域背景

程序崩溃（Software Crash）相关的研究是软件工程领域一个重要的子课题。依据最近几年发表在软件工程领域相关国际会议上的部分实证研究所论述的现状，程序崩溃已经成为当下一种较为严重的软件故障[5]。这是由于程序崩溃对于用户在程序稳定性、程序用户体验等各方面有着较大的负面影响，对于大部分软件开发维护人员而言是一种较为严重和紧急需要处理的软件缺陷。伴随着人们日常生活对于计算机软件依赖程度的逐渐增长，软件变得越来越重要，这也使得软件开发维护人员需要提供更高质量的软件后续支持。在遇到程序崩溃情况时，能够迅速的定位引起程序崩溃的相关代码，并及时修复相关缺陷代码，以减少对软件使用者造成的不便，是对提升自身软件的市场竞争力的有力保证。目前，软件开发维护人员在遇到程序崩溃的软件缺陷的情况下，一般会首先着手于查找相关缺陷代码并修复相关缺陷代码，但若在于无法及时修复的情况下，一般会将软件当前版本会滚至缺陷代码引入之前的开发版本，避免给用户的使用造成不便和产生不良的后果。正因为程序崩溃的严重性，学术界和工业界对程序崩溃一直保持着高度的重视，并有众多学者和工程师对该问题进行不断的探索。

2.2 领域现状

在最近几年当中，有越来越多的关于程序崩溃论文发表在软件工程相关国际会议如 FSE、ISSTA、ICSE、ASE 当中。而在工业界，有越来越多的软件开发组织（例如苹果、微软[13]、Mozilla Foundation）去开发并部署自己的程序崩溃报告收集系统（Crash Report System），以帮助软件开发维护人员能够更加迅速、便捷的了解并修复产品的崩溃缺陷。我们可以看出，不仅仅是学术界对程序崩溃的重视程序越来越高，而且在软件工业界也有愈来越多的产品组织开始重视程序崩溃这一问题。

减少程序崩溃的方式有多种多样，我们既可以从预防的角度去减少软件崩溃出现，又可以从修复的角度去帮助软件开发维护人员去了解崩溃缺陷的存在，并协助软件开发维护人员去修复响应的缺陷。目前，程序崩溃报告被视为一种有

效可行的帮助开发者去了解程序崩溃缺陷的存在及提供修复参考依据的方法。我们对近几年发表在软件工程国际会议上关于程序崩溃报告的论文进行了调研和综述，从中选取了三篇具有代表性的论文，对目前关于程序崩溃报告的研究问题现状、研究方法、局限性等进行对比、分析和综述，并在本综述的最后一部分提出我们所希望研究的问题。

2.2.1 崩溃报告采集

程序崩溃报告是在软件崩溃的一瞬间，通过抛出软件异常等方式，调用预先设计开发并内置在软件产品中的程序崩溃收集系统采集并在适当的时间通过安全的方式上传至软件开发商的程序崩溃报告收集系统当中。程序崩溃报告采集系统对崩溃数据的收集的质量，对于软件开发维护人员能够有效的知晓程序崩溃的造成原因、程序崩溃的影响范围、崩溃引入代码的定位和相应缺陷代码的修复有着非常强的相关联度。我们希望程序崩溃报告收集的能够准确、有效。但是程序崩溃报告并非收集越多的数据越好，对于软件开发维护人员实际有用的数据是一定的，多余收集到的数据或者是无用，或者是重复，这均会增加用户的数据上传流量以及造成程序崩溃报告收集系统的存储负担。除此之外，部分数据还会涉及到用户的隐私数据，这些数据如果被无意采集，可能会给软件开发商造成信誉和法律上的风险。

故此，如何能在保护用户隐私、减少采集低效或无用数据的基础上，又能保证软件开发维护人员可以根据这些报告对程序崩溃进行无障碍的分析、定位甚至于重现程序崩溃成为了一个值得研究的问题。在发表于 ASE 2014 的论文中，作者们提出了一种名为 SymCon 的解决方案[4]。这是一种基于动态 Symbolic 执行的技术，它通过分析哪些条件分支中的约束变量无法在开发者测试程序时通过 Symbolic 执行技术进行线性求解，并在程序崩溃的时候，将这些变量在执行阶段的具体真实值进行记录的方式，实现了软件开发维护人员在调试阶段可以对产生于用户终端上的程序崩溃进行重现，从而帮助软件开发维护人员解决程序的崩溃问题。

在借助研究人员在崩溃报告采集子课题上的现有研究成果，可以帮助我们更加有效的了解和定位程序的崩溃。

2.2.2 收集分类优化

随着众多软件开发组织部署程序崩溃报告收集系统之后，有海量的数据被收集系统所收集。但是对于大批量的程序崩溃报告，软件开发维护人员已经越来越难纯凭手工逐一审阅。软件崩溃原因集合 (Bucket) 的出现在一定程度上解决了这一个问题，它是通过在程序崩溃报告收集系统中设立一个个木桶，将归属于同一崩溃引发因素的程序崩溃报告收集到同一个软件崩溃原因集合 (Bucket) 里面，方便软件开发维护人员查看。但是，目前系统对于程序崩溃报告的崩溃因素的提取方法还局限于以程序崩溃报告中崩溃线程的内存堆栈中最后一个被调用函数的类名加函数名的方式确定。这并不是一个完美的解决方案，因为在这一种归类条件下面，会出现假阳性和真阴性两种错误。在 ICSE 2012 的 ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity 论文，提出了一种基于程序崩溃线程堆栈信息相似度的聚类算法[5]，可以较好的解决程序崩溃报告崩溃因素分类的方法，从而保证在程序崩溃报告收集系统中，同一个崩溃引入因素的程序崩溃报告可以被分类至同一软件崩溃原因集合 (Bucket) 内，为后续的崩溃因素自动化分析提供了一定的数据降噪。

2.2.3 函数级别定位

在 ISSTA 2014 会议中，已经浮现出关于程序崩溃的自动化定位相关的论文——CrashLocator: Locating Crashing Faults Based on Crash Stacks。这一篇论文提出了一个新的观点和实证研究：导致程序崩溃的缺陷代码所在的函数，不一定会存在于程序崩溃的那一瞬间的内存堆栈当中[9]。为了解决这一种情况对缺陷代码的定位造成不准确影响，这一篇文章提出了通过扩展程序崩溃报告中的堆栈函数列表的方式，来将可能造成程序崩溃的函数均纳入到扩展后的可疑函数列表。最后，使用一些估价函数来对可疑函数进行估价和排序。这是一种较为新颖的基于程序崩溃报告的缺陷代码定位算法，但是这一种算法仅仅停留在函数级别，没有考虑提供软件开发进程的上下文关系，帮助开发维护人员理解，是在改动哪些特性的时候造成了程序崩溃。同时，对于较难及时处理且程序崩溃又会给用户造成较大的使用影响的软件缺陷，CrashLocator 这一种方法亦会显得苍白无力。如果能够为软件开发维护人员提供具体的程序崩溃缺陷引入版本，在出现这一种情况时，软件开发维护人员可以临时回滚部分故障代码到先前稳定版本，保障用户使用不会受到较大影响。除此之外，软件开发维护人员还可以借助在软件

代码仓库中的代码改动差异和提交日志，来回想起软件开发的上下文关系，更好的修复程序崩溃问题。

2.3 本章小结

在论文综述的第二章节中，我们阐述了目前软件工程领域对于利用程序崩溃报告去帮助软件开发人员的三种子方向和前人在这些方向上的努力。这一些子方向的最终宗旨皆在为帮助软件开发维护人员更好的参考程序崩溃报告上的有用信息去发现程序崩溃的存在、定位程序崩溃具体的位置和修复程序崩溃的缺陷代码。我们可以看到目前已经有很多研究人员在崩溃数据采集（在 2.1 章节中）和崩溃数据噪声处理（在 2.2 章节中）方面有了一定的贡献。同时我们也看到在崩溃缺陷定位（在 2.3 章节中）方面，已经有前人开辟先河，但提出的解决方法只是局限性的解决了一部分问题。我们希望通过总结前人的经验教训，去设计一种基于程序崩溃报告的缺陷代码引入版本的定位算法，帮助软件开发维护人员 1）可以根据引入版本号在软件代码仓库中查找对应的代码修改记录和日志，找到当时在软件开发过程中的一些上下文关联，更好的理解程序崩溃产生的原因，从而可以更加快捷的定位和修复程序崩溃；2）可以在程序崩溃较难短期内修复的情况下，让软件开发维护人员回滚部分模块代码至故障前，尽可能保障软件使用者不会受到较大的使用不便。

第3章 研究方案

在这一章节中，本文将详细叙述我们通过运用蕴含于软件崩溃报告系统和软件代码版本控制系统中的数据，对造成软件崩溃的代码的具体引入版本进行定位的算法。在图 2 中，给出了我们所研究和设计的算法的框架图。我们算法的输入数据包括：1) 一个来自软件崩溃报告收集系统中一个指定桶(Bucket)的软件崩溃报告集合，一个指定桶内的软件崩溃报告都是由相同的原因造成的软件崩溃； 2) 在软件代码版本控制系统里，源代码的具体变更记录和历史。我们的算法共分为三个步骤。在第一个步骤中，我们的算法将在函数级别进行相关的定位和分析。算法会先从软件崩溃报告中提取软件崩溃时的堆栈信息，并借助从软件源代码中提取出来的函数调用图对崩溃堆栈进行恢复和扩展。鉴于在部分崩溃情况下，引起软件崩溃的缺陷函数已经不再残留于软件崩溃时的堆栈当中，通过对崩溃堆栈的恢复和扩展，可以让我们尽可能的将故障函数覆盖。在第二个步骤中，我们的算法将在语句级别进行相关的分析。算法将会通过运用从软件源代码中所提取出来的数据流图和控制流图，使用回向原处切片算法(backward slicing algorithm)[6]从软件崩溃点对语句进行分析，去分析每一条语句和软件崩溃点的关系。需要注意的是，我们只对从第一个步骤中所筛选出的函数的语句、数据流图、控制流图进行分析，以节约成本并提高分析的有效性。在第三个步骤中，本论文所提出的算法将在函数级别和语句级别分别计算可疑度，并将两个级别的可疑度结合起来，最终将可疑度映射到版本级别。本论文所提出的算法最终将会按照版本的缺陷可疑度呈现出一个版本列表，作为我们的修复建议。

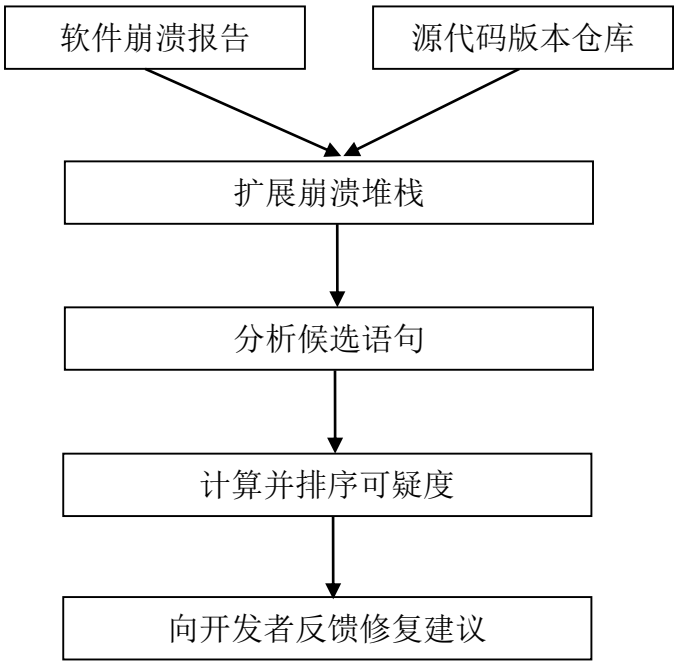

图 2 算法整体框架图

3.1 扩展崩溃堆栈

根据发表于 FSE 2014 会议中的论文[9]所述，引起软件崩溃的缺陷函数不一定总是存留在软件崩溃时的程序堆栈当中。这意味着存在引以软件崩溃的缺陷函数或许已经从程序堆栈堆栈当中弹出，但是缺陷函数执行时所产生的变量通过直接或者间接传播到了软件崩溃点，最终造成了软件的崩溃。为了便于立即，举一个简单的例子：对于一个 C 或 C++的程序在调用 memset 函数时崩溃，我们并不能说是 memset 函数含有代码缺陷，引起了软件崩溃，而更可能的是传递给 memset 的内存地址参数为无效内存地址。若该参数由先前的 A 函数计算得出，则我们认为是 A 函数的代码引起了软件崩溃。在这一步骤中，本论文所提出的方法将需要通过扩展并恢复软件崩溃堆栈的方法，去覆盖造成软件崩溃的缺陷函数，从而确保在接下来的步骤中，本论文所提出的方法在分析和计算可疑度的时候，能够覆盖到造成软件缺陷的代码。

为了扩展软件崩溃堆栈，本论文所提出的方法将首先检查软件崩溃报告收集系统中同类崩溃原因的崩溃报告中出现最早的代码版本，并对它的源代码进行分析，从中提取到函数调用图。当代的软件开发绝大多数都属于递增式软件开发，即软件功能会被不断增加，随之相伴的是软件代码越发复杂，函数愈多。本

方法选取同类崩溃原因的崩溃报告中最早的软件版本进行分析主要是为了减少软件不断开发引入的无关函数，减少分析的噪声，提高算法的准确度。

在完成对待分析软件源代码版本选取之后，本论文通过 Understand 从软件源代码当中提取程序函数调用图。程序函数调用图的英文名称为 Call Graph，是一种基于程序静态分析的程序的函数间调用关系图分析技术。函数调用图是由函数和调用边所组成的有向图，程序中的每一个函数作为图的节点，函数之间的调用关系作为函数调用图的单向边。它可以通过对源代码进行分析的方式，提取出函数之间的调用关系。

我们选取软件崩溃报告的崩溃堆栈中的函数作为起点函数，假设这些函数的深度为 1，每进行一次扩展，新扩展到的函数的深度则在父亲节点的深度的基础上加一。之后，本算法使用广度优先算法的思路，使用优先队列作为数据结构来维护当前函数扩展情况，逐深度使用程序函数调用图对函数调用关系进行恢复。具体而言，若当前本算法在对函数 A 进行恢复，则本算法先从程序函数调用图中调取函数 A 对其它函数的调用数据。之后，本算法对这些可能的被调用函数进行逻辑判断，检查被调用函数在当前软件崩溃报告情况下是否能够被调用，若能够被调用，则本算法会将该函数作为被扩展函数加入到优先队列当中，等待进行扩展。需要注意的是，一个函数只会被扩展一次，而且只扩展深度最浅的那一次调用。在完成进行优先队列有限深度扩展后，我们便获取到扩展后的程序崩溃堆栈。为了便于读者理解本步骤，我们在图 5 中提供了本步骤的伪代码

```
candidateFunctions = empty queue
for each function α in crash stacks in crash report
    candidateFunctions ← add pair < α, depth = 1>
end for
point ← the start position of candidateFunctions
while (point ≠ the end position of candidateFunctions)
    for each call edge β in candidateFunctions[point].first via call graph
        if β_callee not in candidateFunctions and β is logical callable
            candidateFunctions ← add pair < β_callee, depth = depth(β_caller) + 1>
        end if
    end if
```

```
    if out of depth
        break
    end if
end while
return candidateFunctions
```

图 3 扩展崩溃堆栈伪代码示意图

通过本步骤，本论文所提出的方法只选择性的保留软件源代码当中的一小部分可能实际造成软件崩溃的函数。否则，如果对软件源代码当中的全部函数均进行处理和分析，将会降低算法的效率以及精确度。这是因为，软件崩溃报告当中的崩溃堆栈蕴含着大量的有用信息，通过这些信息，可以使得本方法可以将缺陷锁定在局部。

3.2 分析候选语句

为了定位软件的缺陷具体引入版本，首先需要从软件源代码当中获取信息，分析软件崩溃点与每条语句之间的关系。在步骤一当中，本方法通过扩展软件崩溃堆栈，已经将大部分与软件崩溃无关的函数排除掉。为了避免我们的方法对软件整体源代码进行分析同时提高本方法修复建议的精确率，我们将在步骤一当中从软件崩溃堆栈所扩展得到的函数是为候选函数。因为只有候选函数当中的语句会引起软件崩溃，所以本论文所提出的方法仅会对候选函数当中的语句进行分析。

本步骤基于软件静态分析技术。我们假设从软件崩溃报告收集系统当中最早产生该崩溃原因的软件版本为Rev_{first}，本步骤将会从软件源代码的Rev_{first}版本当中提取数据流图和控制流图。在本步骤当中，我们选取分析Rev_{first}版本而非其它后续版本，是出于两方面原因：1) 尽管软件崩溃报告收集系统不一定能在缺陷引入之后的第一时间内收集到崩溃报告，但我们只能确保源代码的Rev_{first}版本中包含我们所寻找的软件缺陷，而Rev_{first}之前的版本，不一定会包含所寻找的软件缺陷，直接贸然对Rev_{first}之前的版本进行分析，会使得算法不精确；2) 分析源代码Rev_{first}之后的版本过于属于过度工作。因为软件增量开发的缘故，在该版本接下来的版本中会引入与软件崩溃无关的代码，额外去分析这

些代码并不能为版本定位工作带来任何帮助，反而会造成不必要的负担。综上所述，本步骤仅对源代码Rev_{first}版本进行分析是充分且必要的。

在候选函数当中只有部分语句与软件崩溃有关系，而在候选函数当中的其它语句则与软件崩溃没有关系。为了便于文字陈述，本论文将简称这些可能会造成软件崩溃的语句为候选语句。在这里，我们给出三种类型的候选语句：类 1) 软件崩溃点自身，即软件发生崩溃的时候所执行的最后一条有效语句； 类 2) 语句中包含至少一个变量，同时，这一个变量可以通过至少一次赋值传递到软件的崩溃点，或，这一个变量可以影响到从该语句到软件崩溃点的分支语句的条件表达式值；类 3) 控制程序到达软件崩溃语句的执行路径上的分支条件语句。换而言之，如果一条分支条件语句能够对软件崩溃语句是否执行产生直接影响，则我们认为这一条分支条件为第 3 类候选语句。为了便于理解，我们在图 4 中给出一个代码样例并对样例进行解释。图 3 中的代码在执行到行 5 的时候因访问无效内存而崩溃，故行 5 为程序崩溃点，根据候选语句定义 1，属于第 1 类候选语句。此时，本算法将继续向上分析。由于行 4 属于条件分支语句，并且行 4 能够影响崩溃语句行 5 的执行，依据候选语句定义 3，行 4 属于第 3 类候选语句。行 3 为一个定义语句，定义了 condition 变量的值，且 condition 的值影响到候选语句行 4，依据候选语句定义 2 的第二个条件，行 3 属于第 2 类候选语句。行 2 定义了 varAddress 变量，并且此变量值直接的传递至程序崩溃点行 5，依据候选语句定义 2 的第一个条件，行 2 属于第二类候选语句。除此之外，其它行均与程序崩溃点无关，故此不属于任何候选语句。

行	源代码	候选类型
1	int var = 512;	
2	int* varAddress = 1024;	类 2)
3	condition = 1;	类 2)
4	if (condition)	类 3)
5	**memset(varAddress, 0, sizeof(int));**	类 1)
6	else	
7	var+= 2048;	

图 4 三类候选语句示意

接下来，我们将介绍分析获得这三类候选语句的方法。对于第 1 类候选语句，我们可以非常直接的从软件崩溃报告的堆栈信息当中获取得到，在此不再赘述。对于获取其它两类候选语句，本论文所提出的方法是用了回向原处切片算法[6]，从软件的崩溃点开始反向分析之前所执行到的语句，去获取变量语句到软件崩溃点的传播情况。在这里，本论文所提出的方法选用类似 Use-Definition Chain[2][3]来帮助分析。UD Chain 是一种描述一个变量的使用和之前定义情况的数据结构，通过这一种数据结构，我们可以较好的反向描述变量到软件崩溃点的传播关系。对于在 UD Chain 当中的每一个变量，本论文所提出的算法将把最后一次对该语句执行定义操作的语句视为第 2 类候选语句。对于第 3 类候选语句，本算法将检查控制流图，将所有会影响到程序执行到软件崩溃点的条件分支语句作为第 3 类候选语句。为了便于读者理解本步骤，我们在图 5 中提供了本步骤的伪代码。

通过上述方法，本论文所提出的方法将对所有在步骤一中所保留下来的函数的语句进行分析，从中再次筛选并保留出可能造成软件崩溃的候选语句。并最终结合软件工程会议当中前人所发表的实证研究中的经验，在步骤三种计算出每一条候选语句的可疑度。

```
1   candidateStmt = empty set;
2   influentVariables = empty set;
3   candidateStmt ← add stmt(crash point)
4   for each variable α used in crash point via data flow
5       influentVariables ← add variable α
6   for each stmt β executed before crash point via source code
7       addMark ← false
8       for each variable α defined in stmt β via data flow
9           If α in influentVariables
10              addMark ← true
11          end if
12      end for
```

```
13      if β impacts the execution path of crash point
14          addMark ← true
15      end if
16      if addMark = true
17          for each variable α used in stmt β via data flow
18              influentVariables ← add variable α
19          end for
20          candidateStmt ← add stmt β
21      end if
22  end for
23  return candidateStmt
```

图 5 分析候选语句伪代码示意图

3.3 计算并排序可疑度

结合相关观察和在软件缺陷分析领域中的先前研究结果，本论文所提出的算法将使用五种特征来计算软件代码仓库中的每个版本引入软件崩溃缺陷的可疑度，并将排序后的版本列表作为我们的修复推荐提供给软件开发人员，帮助软件开发人员更好的找到软件开发时的上下文关系，更快的修复软件缺陷。

3.3.1 特征 1： 函数的代码行数 (FLOC)

一篇发表于 ICSM 2009 的实证研究[10]说明，一个函数含有越多的代码行数，则这个函数越可能包含有软件缺陷，并会造成软件崩溃。因此，我们选择使用 FLOC 函数来量化这一特征，具体的公式如下：

$$FLOC(func) = log(LOC(func) + 1)$$

3.3.2 特征 2： 到函数崩溃点的距离 (IAD)

若一个函数需要函数需要通过较多次堆栈扩展才能够被覆盖到或者这个函

数在堆栈中距离崩溃点较远的位置中，则这个函数很小可能会包含引起我们正在定位的软件崩溃缺陷。我们通过引入$callDepth(func)$函数来表示从原始崩溃堆栈覆盖到目标函数所需要的堆栈扩展次数。这个函数用于量化某一特定函数距离软件崩溃点的扩展距离，使得需要越多扩展次数的函数的可疑度权重越低。同时，我们通过引入$posInStack(func)$函数来表示从函数映射在崩溃堆栈上的位置到崩溃点的距离。在实验阶段，我们发现$callDepth(func)$比$posInStack(func)$更为重要，故此我们通过引入$log(posInStack(func))$来降低后者在评分系统当中的权重。特征 2 的具体公式如下：

$$IAD(func) = \frac{1}{log(posInStack(func)) + callDepth(func)}$$

3.3.3 特征 3： 候选语句与崩溃点关联度 (CRD)

对于在步骤二当中所筛选出来的候选语句而言，并不是每一条语句都拥有着相等的崩溃引发可疑度。为了对此进行量化，我们为每一条候选语句引入候选语句与崩溃点关联度即 CRD 函数来衡量它们的可疑度。若一个候选语句的 CRD 值越高，则意味着这个候选语句越有可能造成该软件崩溃。

候选语句的可疑度影响要素之一便是该语句是否有对可以传递到软件崩溃点的变量进行定义操作以及对多少这样的变量进行操作。然而，这些变量对于造成该软件崩溃并不是起到相同的影响作用。经过实证研究，发现变量通过越少次赋值便可传递到软件崩溃点的变量，越有可能造成相应的软件崩溃。故此，我们引入变量距离函数，即$varDist(var)$函数来对不同变量可能造成软件崩溃的可能性进行量化。 $varDist(var)$的具体定义如下：变量 var 当中的值通过至少$varDist(var)$次可以传递到软件的崩溃点。对于$varDist(var)$越小的变量，越有可能造成该软件崩溃。借此，我们给出 CRD 特征的具体公式：

$$CRD(Stmt) = \sum_{var_i \in Being-defined\ in\ Stmt} \frac{1}{varDist(var_i) + 1}$$

3.3.4 特征 4： 函数出现频率 (FF)

若一个函数频繁的在同一个崩溃原因的软件崩溃报告集合当中的不同软件崩溃报告中频繁出现，则这个函数越有可能造成该软件崩溃。在这里，我们通过$\#FFB(func)$来表示某一函数在同一崩溃原因的软件崩溃报告集合当中出现的频率。同时，若一个函数越频繁的出现在不同的软件崩溃报告集合当中，则这个函数越低可能造成该软件崩溃。试想，在 C++、Java 等语言当中的 main 函数，几乎会出现在所有的软件崩溃报告当中，但却恰恰很少存在缺陷。同时，在自动化测试颇为流行的当今软件开发模式当中，若一个函数越容易被到达，则有越多的测试数据，可以保障这个函数的健壮性，这也从另一方面说明了这个函数越低可能造成该软件崩溃的原因。在这里，我们通过$\#BCF(func)$来表示某一函数在不同崩溃原因的软件崩溃报告当中出现的频率。我们给出特征 4 的具体公式：

$$FF(func) = \frac{\#FFB(func)}{\#\ BCF(func)}$$

3.3.5 特征 5： 版本出现频率 (RF)

在通常情况下，在软件代码版本控制系统当中每一次提交的代码变动量是较少且有限的，并且会进行相应的自动化测试和单元测试，单次代码提交所引入的代码缺陷数量亦是少量且有限的。如果一个版本频繁的出现在不同的崩溃原因的报告集合当中，则这个版本越小可能包含所搜寻的软件崩溃缺陷，在量化该版本的可疑度的时候，应该适当降低该版本的权重。我们使用$\#BCR(rev)$来表示版本 rev 在其它不同崩溃原因的集合当中的不经特征 5 量化的前一百个修复建议版本中出现的频率。若一个版本的$\#BCR(rev)$越高，则这个版本越小可能性造成该缺陷，故版本出现频率的公式如下：

$$RF(rev) = \frac{1}{\#\ BCR(rev)}$$

3.3.6 整体合并

在给出本论文提出的算法所使用的具体量化特征后，本小节将介绍我们所提出的算法如何具体将这些量化特征进行结合，并将可疑度分数映射回具体的提交版本。

首先，本算法将使用特征 1 至特征 4 来计算每一个在步骤 2 中筛选出来的候选语句的可疑度分数，在这里，我们定义$\#Score(Stmt)$为每一条候选语句可疑度分数。同时，为了便于文字表达，我们定义函数$func(Stmt)$，该函数表示在给定候选语句 Stmt 的情况下，返回该候选语句所在的函数名称。具体公式如下：

$$Score(Stmt) = CRD(Stmt) * FLOC\big(func(Stmt)\big) * IAD\big(func(Stmt)\big) * FF(func(Stmt))$$

接下来，本论文所提出的算法将会将每一条候选语句的可疑度分数映射回具体的开发版本，即最后一次对该候选语句进行修改的版本或者是引入该候选语句的版本。同时，特征 5 也会在此步被引入到可疑度分数评估计算当中。本论文所提出的算法具体计算某一特定 rev 的分数公式如下：

$$Score(rev) = RF(rev) * \sum_{Stmt \in rev} Score(Stmt)$$

最后，本论文所提出的方法将依据Score(rev)对各个候选版本进行排序，并按照排序结果向软件开发人员提供修复建议。需要注意的是，我们将同一个软件崩溃原因集合当中每一份软件崩溃报告对每一个 rev 的分数贡献进行累计，从全集合的角度上对各个候选版本按照分数进行排序。软件开发人员可以通过参考我们所提供的建议列表，对相应版本进行检查，并调取软件开发上下文关系。

第4章 实验结果

4.1 实验数据集

在本篇论文当中，我们选择了来自 Mozilla Foundation 的真实世界的产品来对本文所提出的算法的有效性进行评估。我们选择 Mozilla Firefox 产品的原因在于：1) Mozilla Firefox 作为世界前三大的浏览器，在世界范围内均被广泛的使用，有着较高的流行度，便于保证实验样本的充分性；2) Mozilla Foundation 属于开源组织，且是为数不多的同时将软件源代码仓库、软件崩溃报告和缺陷报告开放的软件组织。上述两点原因，使得本论文最终选取了 Mozilla Firefox 作为我们的实验数据的测试对象。Mozilla Firefox 使用够了 Mercurial 作为软件代码版本控制系统，在其中有超过 243,712 个开发版本。同时我们从软件崩溃报告收集系统当中收集到 4,250 个软件崩溃报告，这些软件崩溃报告分别属于 17 种不同的软件崩溃原因集合。我们将实验数据集的详细信息放置于表 1 中。

软件名称	Mozilla Firefox
软件发行版	36.0a1
源代码数量	68,313
开发版本数量	243,712
崩溃原因集合数	17
崩溃报告数量	4,250

表 1 实验数据集详细信息

本论文所选用的软件崩溃报告均是后来在软件缺陷追踪系统中被标记为已修复的软件崩溃报告。这确保我们所收集到的软件崩溃报告均是可以根据在软件缺陷追踪系统和软件代码版本控制系统中的修改记录，来获取到实验数据集的答案的，这也使我们的实验数据集有了依据。我们使用了 MSR 2005 会议中所提出的方法 [12] 将软件追踪系统中的修改记录和在代码版本控制系统中的提交记录相连接起来，根据代码的改动差异和改动语句在改动前的所属版本信息，来获取到具体的版本信息。

4.2 度量函数

4.2.1 Recall@N

通过检查我们论文所提出的算法给出的前 N 个修复建议，便能够得到正确软件崩溃缺陷代码引入版本的百分比概率。这个值越接近百分之百，则表明本论文所提出的算法在软件崩溃缺陷代码引入版本定位上的表现越有效。换而言之，软件开发人员便可以更加容易的借助我们的算法去找到缺陷代码引入版本。

4.2.2 MRR

MRR 英文全称为 Mean Reciprocal Rank，中文名称为平均倒数排名。平均倒数排名是统计学当中一种依据排序的正确性对查询请求响应结果进行评估的测量方法。查询响应结果的倒数排名是第一个正确答案的倒数积，而平均倒数排名是多个查询结果的平均值[11][24]。平均倒数排名从另一个角度来评测了本论文所提出的算法在定位软件崩溃情况的表现。平均倒数排名的值越大，则说明软件崩溃定位算法越有效。具体的 MRR 公式如下：

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{rank_q}$$

4.3 实验结果

在本小节当中，我们将会提出对本篇论文所提出的算法的研究问题和相应的实验结果，通过这个问题，来更好的阐述本文所研究的课题，并用实验数据来证实本论文所提出的算法的效果。

有多少错误可以被成功定位

这个问题可以最直接的反馈软件崩溃定位的有效性。我们通过测量本篇论文所提出的算法的缺陷代码引入版本定位成功率百分比(Recall)和平均倒数排名(MRR)来说明本问题。这两个参数指标，均是数值越大，算法效果越好。在 4.1

所给出的实验数据集下的实验结果，在下述表 2 中具体给出。

产品名称	Mozilla Firefox		
产品发行版	36.0a1		
度量函数	结果百分比	命中桶数目	命中报告数目
Recall@1	53%	9	2250
Recall@5	59%	10	2500
Recall@10	77%	13	3250
MRR	57.79%	-	-

表 2 算法评估结果

通过表 2 的实验结果可以看到，软件开发人员通过检查本论文所提出的算法推荐的第一个可疑版本，可以成功的定位 53%的软件崩溃报告集合；如果软件开发人员检查本算法推荐的前是个可疑版本，可以成功的定位 77%的软件崩溃报告集合。软件开发人员仅需要对相应检查对应版本的软件代码管理系统的代码差异和改动日志，借助领域知识，就可以做出初步判断。相对于传统的通过二分的方式，我们减少了源代码编译的大量时间以及避免了很难生成测试用例的尴尬。除此之外，平均倒数排名(MRR)为 57.79%。总体来说，实验结果说明本论文所提出的算法是有效的。

第5章 分析与讨论

5.1 算法工作原理

在本子章节中，我们将阐述本论文所提出的算法的工作原理。本算法在接收到软件崩溃报告集合之后，会对集合中的每一个软件崩溃报告进行分析处理。在软件崩溃报告当中，程序崩溃堆栈包含有软件在崩溃时在堆栈中的函数调用关系，该信息已经被多篇论文论述[7][8][9]为可以用于崩溃追踪及崩溃缺陷场景重现。在本篇论文的算法中，使用程序崩溃堆栈作为起点，通过扩展崩溃堆栈的方法，尽可能的将含有崩溃缺陷的函数覆盖进候选函数列表。已有论文说明，扩展崩溃堆栈的方法在覆盖缺陷函数方面是有效的[9]。从而，本论文的算法在完成步骤一之后的候选函数列表是可行有效的。通过分析筛选候选函数，可以将软件源代码当中大量与该软件崩溃无关的代码排除掉，而仅保留相关了存在造成该软件崩溃可能性的代码。从某一种角度上来讲，在步骤一当中，本算法利用现有的软件崩溃报告的堆栈信息，并结合扩展崩溃堆栈算法对整体源代码进行了排除，去除了不可能造成软件崩溃的部分。

在完成对函数的筛选之后，本论文所提出的算法再次借助软件崩溃报告的堆栈信息当中所记录的代码语句执行位置信息，从语句层面对代码进行筛选。若软件崩溃不是由于最后一条有效语句的编写错误直接造成的，则是由于不恰当的变量或参数造成了软件崩溃。在这里借助回向原处切片算法[6]的语句处理顺序方法，本算法从程序的崩溃点即最后一条执行的有效语句开始，从该语句开始建立 UD Chain，来描述变量的赋值传递关系。并在进行逆向对代码扫描简历 UD Chain 的过程中，记录距离崩溃点执行距离最短的在 UD Chain 上进行定义操作的语句。因为这些语句对变量的值进行了相关操作，并且这些值最终将会影响崩溃语句的执行效果。故此，通过步骤二的操作，我们可以筛选出可疑的候选语句，排除更多的无关语句。在这里，并没有使用崩溃报告中的寄存器值信息，这是因为这些寄存器值在现有的方法下，无法被还原至具体程序变量当中。

最后，本算法通过五个软件工程当中的特征对剩余的语句进行评分，每一条语句都被计算并赋予一个可疑度，再通过在软件代码版本控制系统当中的最后语句修改记录情况，将每一条语句的可疑度映射回对应的开发版本。对于同一个

软件崩溃原因集合当中的不同软件崩溃报告，本算法会把不同软件崩溃报告对于版本的可疑度分数贡献累计，实验结果证实了这一种方法有效。软件工程领域相关会议也已有对于不同软件崩溃报告按照崩溃原因合并为相应集合的算法[5]。累计相同崩溃原因集合的不同软件崩溃报告这一种方法有效的原因在于尽管集合当中软件崩溃报告有着相同的崩溃原因，但因为用户系统环境的差异，造成程序在崩溃时的执行路径不相同，即有不同的崩溃堆栈。例如，我们在 Mozilla Firefox 的源代码当中，发现通过使用 C++预处理宏定义(如表 3)来在不同的操作系统下执行不同的语句分支。

系统平台	宏定义方式
Mac OS	#ifdef __APPLE__
MingW (32 bits)	#ifdef __MINGW32__
Linux (GCC)	#ifdef __linux__
Unix	#ifdef __unix__

表 3 不同平台宏定义方式举例

5.2 影响实验的因素

5.2.1 实验样本选择

因为需要对本论文说提出的算法进行客观评价，我们选用来自第三方的真实产品数据。在本实验当中，实验数据集仅选自来自 Mozilla Foundation 的真实世界产品数据。这主要局限于目前仅仅只有 Mozilla Foundation 将自己的产品代码版本控制系统、软件缺陷报告追踪系统和软件崩溃报告收集系统的数据公开，允许我们通过应用爬虫技术从这些系统中收集到我们实验所需要的实验数据。同时，我们也注意到，在依据软件崩溃报告来进行缺陷定位的论文中[9]，也受到与本论文相同的实验样本选择局限性。如果更多的开源软件基金会能够将相关数据在不暴露用户隐私的前提下开放出来，将会更加有利于我们通过实验来评估本论文所提出的算法的有效性，同时促进学术界对相关问题的研究。

5.2.2 程序调用图的局限性

在具体进行验证实验的过程中，我们发现软件程序遇到了程序调用图的局限性。在提取待实验软件的程序调用图时，是采用基于程序静态分析的技术，但这种技术仅仅是对源代码在静态的情况下进行分析，其中一个缺点便是无法获取程序运行时的值。但我们选择的实验对象 Mozilla Firefox 中存在一些函数指针形式的函数调用，这些函数指针的具体指向函数只有在程序运行时才能够被具体确定，并且有些函数指针是依赖于软件运行环境。这导致我们在从源代码中提取程序调用图的时候，无法获取到通过函数指针进行的函数调用，使得函数调用图缺失相应的调用边，并会影响在进行 3.1 扩展崩溃堆栈时候的准确程度。除此之外，Mozilla Firefox 最新版本在进行项目编译的时候，会通过部分 Python 脚本生成部分 C++文件，并使得这些文件参与到项目的编译。对于这一部分通过 Python 脚本生成的 C++文件，因为没法预先从代码版本控制系统中直接获取，亦无法在代码版本控制系统中进行追溯，所以对于存在于这些 Python 当中的缺陷函数，因为无法被包含进函数调用图当中，也不能被覆盖。这也是当前采用软件静态分析技术时遇到的局限性。

5.3 优势和应用

本论文所提出的方法，相对于基于二分查找的软件崩溃缺陷代码引入版本修复工具或者技术如 mozregression 等来说，这些工具和技术有两大比较明显的劣势。劣势之一是，在每一次进行二分测试是否该中间版本含有软件缺陷的时候，工具或者技术均需要对从代码版本管理系统中取回的代码进行重新编译，生成可执行文件才能进行具体的运行测试，来判断该中间版本是否含有目标的软件崩溃缺陷。但是，每一次切换版本进行重新编译的时候都需要耗费大量的时间，给软件开发人员带来极大的不便。例如，对于像 Mozilla Firefox 这种规格的软件，编译时间已达到小时级别。劣势之二是，二分测试的核心是在于通过运行程序来判断软件是否含有崩溃，而整个过程依赖于人力和测试用例。软件开发人员需要按照测试用例通过手工操作的方式去测试相关功能，并尽可能的触发相应的软件缺陷。但是，软件崩溃一般隐藏在较深的软件执行路径分支当中，人为的去触发相对困难。况且，因为 GUI、事件和网络等因算，导致生成测试用例的难度增

加，甚至很多情况下无法生成测试用例。相比之下，本文所提出的算法的优势在于不需要对软件源代码进行编译运行，仅需要对源代码进行静态分析，从而节省了时间。同时，本论文所提出的算法不依赖于测试用例，避免了生成测试用例上的困难。

本论文所提出的方法，相对于现有基于软件崩溃报告的缺陷函数定位算法相比，则从另一个角度为软件开发人员提供软件缺陷定位信息，帮助软件开发人员来对软件进行日常维护。相对于仅仅反馈单一函数信息的定位算法，我们提供软件崩溃代码引入版本的算法一来可以帮助软件开发人员找到软件开发时的上下文关系，查看软件代码控制系统的代码修改差异和提交说明，可以了解具体是在进行什么操作的时候引入了缺陷代码。从而使得软件开发人员更加便利的修复软件缺陷。二来，在遇到相对较难短时间内修复的软件崩溃，并且该软件崩溃又对用户使用造成较大负面影响，则软件开发人员可以将相应模块，回滚至缺陷代码引入之前的版本，重新部署给客户，从而保证客户的正常使用。上述这些特性，都是反馈单一函数信息的定位算法所无法做到的。

本论文所提出的算法可以被软件开发商部署在软件崩溃报告收集系统后端，当软件崩溃报告收集系统收集某一类相同崩溃原因的软件崩溃报告样本达到一定数量后，便可以触发本论文所提出的算法对软件崩溃报告进行自动化分析，并将分析结果发送给对应的软件开发人员进行修复。

第6章 本文总结

在本篇论文当中，我们首先给出了关于软件崩溃相关的背景介绍和当前软件工程工业界所遇到的问题。然后，我们通过查阅相关文献资料，了解当前学界对于软件崩溃定位所做的相关工作，并在本论文中以文献综述的形式进行呈现。软件崩溃是一种较为严重的软件缺陷，不仅会对软件开发人员带来维护上的困难，同时也会给使用者带来经济上的损失，能够敏捷修复软件崩溃的缺陷是软件开发的一种核心竞争力。

本篇论文在第三章当中阐述了一种通过分析软件崩溃报告和软件代码版本控制系统，来向软件开发人员提供缺陷代码引入版本定位的算法。算法主要使用了软件分析技术当中的程序静态分析技术，通过从软件崩溃报告当中提取崩溃时堆栈信息，并借助程序函数调用图来对崩溃堆栈进行扩展，尽可能的对包含缺陷代码的函数进行。并对这些候选函数的语句通过借助数据流图和控制流图进行逆向分析，筛选出可疑的候选语句。并最终通过五个特征对这些候选语句的可疑度进行计算，并按照递减序向软件开发人员提供修复建议。

我们通过使用来自 Mozilla Foundation 的真实世界中的产品及相关数据，来对本论文所提出的算法的有效性进行评估，评估结果说明我们的算法可行有效。软件开发人员通过检查本论文所提出的算法所提供的前 1 个可疑版本建议，可以达到 53%的定位成功率；通过检查前 5 个可疑版本建议，可以达到 59%的定位成功率；而通过检查前 10 个可疑版本建议，则可以达到 77%的定位成功率。相比在数十万的版本中大海捞针的寻找，通过采用本论文所提出的算法，可以较大的减少软件开发人员的工作量

参考文献

[1] H. Seo, S. Kim. Predicting recurring crash stacks. in Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. 2012: Pp 180-189

[2] Mark N. Wegman, F. Kenneth Zadeck. Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems (TOPLAS). 1991: pp 181-210

[3] AV Ao, R SETHI, JD ULLMAN. Compders: Prmctples, Techruques, and Tools. Addison-Wesley, Reading, Mass., 1986

[4] Yu Cao, Hongyu Zhang, Sun Ding. SymCrash: selective recording for reproducing crashes. Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. 2014: pp 791-802

[5] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, Peter Nobel. ReBucket: a method for clustering duplicate crash reports based on call stack similarity. Proceedings of the 34th International Conference on Software Engineering. 2012: pp 1084-1093

[6] G. A. Venkatesh. The semantic approach to program slicing. Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. 1991: pp 107-119

[7] Shay Artzi, Sunghun Kim, Michael D. Ernst. ReCrashJ: a tool for capturing and reproducing program crashes in deployed applications. Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. 2009: pp 295-296

[8] Wei Jin, Alessandro Orso. BugRedux: reproducing field failures for in-house debugging. Proceedings of the 34th International Conference on Software Engineering. 2012: pp 474-484

[9] R. Wu, H. Zhang, S.C. Cheung, S. Kum. CrashLocator: locating crashing faults based on crash stacks. Proceedings of the 2014 International Symposium on Software Testing and Analysis. Pp. 204-214

[10] Hongyu Zhang. An investigation of the relationships between lines of code and defects. Software Maintenance, 2009. ICSM 2009. IEEE International Conference on. pp 274-283

[11] E.M. Voorhees. The TREC-8 Question Answering Track Report. Proceedings of the 8th Text Retrieval Conference. 1999: pp. 77–82

[12] J. Śliwerski, T. Zimmermann, A. Zeller. When Do Changes Induce Fixes?. Proceedings of the 2005 international workshop on Mining Software Repositories, 1-5

[13] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, G. Hunt. Debugging in the (very) large: ten years of implementation and experience. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009: pp 103-116

[14] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang and H. Mei. A survey on bug-report analysis. Science China Information Sciences. Volume 58, Issue 2, Pages 1-24, 2015

[15] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schutze. Introduction to Information Retrieval. Cambridge University Press. 2008

[16] Hao Hu, Hongyu Zhang, Jifeng Xuan and Weigang Sun. Effective Bug Triage based on Historical Bug-Fix Information. in The 25th IEEE International Symposium on Software Reliability Engineering. 2014: Pages 122-132

[17] K. Liu, H. B. K. Tan and H. Zhang. Has this Bug Been Reported?. in Reverse Engineering (WCRE), 2013 20th Working Conference on. 2013: Pages 82-91

[18] Jian Zhou and Hongyu Zhang. Learning to Rank Duplicate Bug Reports. In Proceedings of the 21st ACM international conference on Information and knowledge management. 2012: Pages 852-861

[19] R. Wu, H. Zhang, S. Kim and S.C. Cheung. ReLink: Recovering Links between Bugs and Changes. In Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. 2011

[20] Sunghun Kim, Kai Pan and E. E. James Whitehead, Jr. Memories of bug fixes. In Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering. 2006: Pages 35-45

[21] T. Zimmermann, A. Zeller, P. Weissgerber and S. Diehl. Mining version histories to guide software changes. In IEEE Transactions on Software Engineering. 2005: Volume: 31, Issue: 6, Pages 429-445

[22] Schroter, A., Bettenburg, N., Premraj, R. Do stack traces help developers fix bugs?. Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on. 2010: pp 118-121