

復旦大學

本科毕业论文



论文题目: 基于 Mozilla 的安全性漏洞再修复经验研究

院 系: 软件学院
专 业: 软件工程
姓 名: 张凯 学 号: 12302010061
指导教师: 赵文耘 职 称: 教授
单 位: 复旦大学
日 期: 2016 年 5 月 23 日

目 录

摘 要.....	1
ABSTRACT.....	2
第一章 引 言.....	4
1.1 研究的意义及必要性.....	4
1.2 现状研究.....	5
1.3 本文工作.....	5
第二章 背景与相关工作.....	6
2.1 漏洞生命周期.....	6
2.2 安全性漏洞分类.....	6
2.3 安全性漏洞修改模式.....	7
2.3.1 数据预处理阶段.....	8
2.3.2 数据处理阶段.....	8
2.3.3 数据确保阶段.....	9
2.4 安全性漏洞修复过程.....	10
2.5 相关工作.....	11
2.6 本章小结.....	12
第三章 研究设计.....	13
3.1 数据来源.....	13
3.1.1 安全性漏洞基本信息.....	13
3.1.2 安全性漏洞代码提交信息.....	16
3.2 研究内容.....	17
3.3 分类方式.....	18
3.4 本章小结.....	19
第四章 数据分析.....	20
4.1 基础数据统计.....	20
4.2 安全性漏洞再修复原因.....	20
4.2.1 加强漏洞的填补.....	20
4.2.2 减弱漏洞的填补.....	21
4.2.3 填补原来没想到的漏洞.....	21
4.2.4 对修复进行登记或测试.....	22
4.2.5 以前修复无效或错误而重新修复.....	23
4.2.6 怀疑修复造成了错误，后来发现不是.....	23

4.2.7 本节小结.....	23
4.3 安全性漏洞类型与再修复原因.....	24
4.4 修复基础信息统计.....	26
4.5 初始修复与再修复对比.....	27
4.5.1 修复过程的对比.....	27
4.5.2 修改时间和修改者的对比.....	27
4.5.3 修改模式的对比.....	28
4.5.4 文件修改情况对比.....	29
4.6 本章小结.....	30
第五章 再修复预测.....	31
5.1 线性关系再修复预测.....	31
5.2 增强漏洞填补范围的预测.....	33
5.3 本章小结.....	35
第六章 总 结.....	36
参考文献.....	37
致 谢.....	40

摘 要

安全性漏洞是指能被攻击者利用在计算机系统中获取未经授权的访问或进行超出权限的行为的一种软件漏洞，安全性漏洞会导致安全性缺陷，使计算机系统陷入危险之中。安全性漏洞不一定会导致程序崩溃或停止运行，但一旦安全性漏洞被攻击者利用，则可能会造成不可挽回的损失。

已有研究表明，安全性漏洞相较于其他类型的漏洞更容易发生再修复[1]，这使得安全性漏洞需要更多的开发资源，且增加了在程序员都不知道的情况下造成巨大损失的可能性，因此减少安全性漏洞再修复的发生的重要性不言而喻。

对安全性漏洞再修复的经验研究有助于减少再修复的发生，提高再修复的效率，减少开发资源的消耗，减轻安全性漏洞造成的损失。我们通过对 Mozilla 工程 2005~2011 年 48 个发生再修复的安全性漏洞的安全性漏洞类型、发生再修复的原因、再修复的次数、修改的 git 提交数、修改的文件数、修改的代码行数的增减、初始修复和再修复的对比（使用的修改模式、使用的修复过程、修改的时间、修改者、修改文件内容）等数据进行统计、分析、对比，了解了各种类型的安全性漏洞修复时对代码的影响，总结了各种类型的再修复的修改模式使用、修复过程使用、修改文件及内容等的规律。

根据总结规律，我们设计了两种安全性漏洞再修复的预测方法，有助于减少该类型安全性漏洞再修复的发生，一种是通过数据流分析和控制流分析来对线性关系的再修复进行预测，找到发生安全性问题的数据源头；另一种是通过克隆代码的分析找到具有相似功能且有很大可能性带有相同安全性问题的代码的增强漏洞填补范围的再修复预测；另外，还提供了一些对应每种再修复类型在修改模式、修改文件和代码、修复方式等方面的修复建议，使修复人员能够更快找到修复核心及修改方式，减少对开发资源的损耗和安全性漏洞可能造成的损失，还能对未来开发安全性漏洞再修复预测及自动化修复的工具提供基础知识及经验。

关键词 安全性漏洞，再修复，经验研究，Mozilla 工程，修复建议，再修复预测

ABSTRACT

Security bug refers to the software bug which can be exploited by attackers to get unauthorized access or do some illegal activities beyond their privilege in the computer and cause security vulnerabilities which will cause security failure of the computer system. Security bug may not cause fail-stop failure, but once being exploited by attackers, irreparable loss may happen.

However, security bug reopens more often compared to other types of bugs[1] which needs more development resources to fix it, and increase the risk of huge loss. So decreasing the possibility of security bug reopens is urgent and important.

Hence, an empirical study of reopened security bugs is important, which can help reduce the occurrence of security bug reopening, improve the efficiency of reopened security bug fixing, reduce the consumption of development resources, and reduce the loss caused by security bugs. Our study is based on the Mozilla project and collects a total of 48 reopened security bugs to analyze the security bug types, the reasons of reopening, the times of reopening, the times of commits in the git, the number of files which are modified, the lines of adding and deleting code, and the comparison of the original fixing and reopened fixing (usage of fixing pattern, usage of fixing process, total time of fixing, fixing developer, files and code which are modified). The results show the modification to the code while fixing each type of security bugs, as well as the regular patterns of the usage of fixing pattern, the usage of fixing process, and the modification of files and code of all types of reopened security bug fixing.

Based on the findings, we design the methods to predict the occurrence of two specific types of reopened security bugs which can help reduce the occurrence of reopening. One is to predict the Linear Relation reopening using the data flow and control flow analysis to find the source of the variable of the original fixing. The other is to predict the Be Wider reopens by analyzing clone code to find the code which has similar function and the same security problem with a high probability. Moreover, we also propose some advice for reopened security bug fixing in fix patterns, fix

ideas as well as the files and code that should be modified, so that the developer can quickly find the core of fixing and fix pattern which helps to reduce the waste of development resources and the loss of security bug, providing the fundamental knowledge and experience for future development of tools to predict the occurrence of security bug reopening and automatically reopened security bug fixing.

Keywords Security bug, fixing of reopened bug, empirical study, Mozilla project, fixing advice, security bug reopen prediction

第一章 引言

1.1 研究的意义及必要性

在软件开发和维护过程中,程序员常常会遇到程序运行状态与预期不符的情况,如输出值不正确、程序死机、数据丢失等等,这种现象就是程序的漏洞。如果程序员在程序中发现了漏洞的存在,则必定会停下开发工作修复漏洞,或者投入专门的人力物力进行漏洞修复工作。不论是哪种应对方案,都会不可避免地开发资源进行消耗,使开发成本变得更加昂贵,直接导致开发软件的利润变得更小,软件竞争力下降。有研究表明,软件维护和更新行为占用了超过 90% 的软件开发资源[2, 3],而漏洞的修复工作则是软件维护与更新的主要工作。因此,减少软件漏洞的出现是保证程序稳定、使软件客户满意、增强软件市场竞争力的重要手段之一。

在众多漏洞类型中,安全性漏洞则是软件开发者在软件维护与更新的过程中重要的关注点之一[4, 5, 6]。安全性漏洞是指能被攻击者利用来在计算机系统中获取未经授权的访问或进行超出权限的行为的一种软件漏洞,安全性漏洞会导致安全性缺陷,使计算机系统陷入危险之中。安全性漏洞不一定会导致程序崩溃或停止运行,但一旦安全性漏洞被攻击者利用,则可能会造成不可挽回的损失。

例如,2014 年 3 月的携程数据泄露事件就是因为安全性漏洞而导致用户个人信息、银行卡信息等泄露,这次事件给信息遭到泄露的用户造成了极大的困扰。数据的泄露不仅使用户私人信息被公开,还可能会因银行卡被盗刷而造成经济损失。这种事件虽然并未使携程的任何软件停止运行,但其中造成的损失却是不可估量的。首先,因信息泄露造成经济损失的用户需提供赔偿;其次,用户对携程的信赖度下降,使携程的总用户减少,给企业的信誉造成了不良影响。不只是携程,国内曾连续爆出一系列数据泄密事件:中国人寿 80 万保单信息泄露、搜狗输入法漏洞、12306 用户信息泄露等等。

这些安全性漏洞造成重大损失的教训无不在告诉我们安全性漏洞预防以及及时修复的重要性。可是,漏洞的修复有时候不能一蹴而就,程序员也不可避免地会发生修复漏洞错误的情况。程序员在对漏洞修复完成之后会关闭这个漏洞,表示漏洞修复完成可后来如果发现这个漏洞修复未完全使之正确,则需要重新打开这个漏洞,对这个漏洞进行再修复,避免在程序员自认为修复正确的情况下,安全性漏洞却悄悄地使我们蒙受重大的损失。Zaman et al. 发现,安全性漏洞相比于其他漏洞来说要更常需要再修复[1],因此,对安全性漏洞的再修复进行研

究，找出再修复与初次修复的隐含关系，对安全性漏洞再修复的预测和提供修复意见十分重要。

1.2 现状研究

目前为止，对安全性漏洞的研究主要集中在识别安全性漏洞[1, 7, 8]，预测安全性漏洞[9, 10, 11]，对安全性漏洞进行分类[12, 13, 14]，对安全性漏洞报告[15]、性质[16]进行研究与比较，安全模式的应用[17, 18]，安全性检测[19, 20]等，却十分缺少对安全性漏洞修复的研究。

而对于再修复的研究也寥寥无几，基本上是通过漏洞的特征、漏洞报告等信息对漏洞的再修复进行预测[21, 22]，但是因为是针对的所有漏洞，安全性这个特点和其特殊性没有重点考虑，因此安全性漏洞再修复的预测不是那么有效。通过对一些现有工作的研究，可以作为我的研究的基础知识、参考方法和方向指引。

1.3 本文工作

本论文主要通过 48 个发生再修复的安全性漏洞的各项数据进行统计分析，找到安全性漏洞在再修复时存在的各种修复规律，得到针对安全性漏洞特殊性的再修复的预测方法，为开发安全性漏洞再修复预测和自动化修复的工具提供知识支持。

接下来的论文结构如下：第二章是进行经验研究的一些相关工作和背景知识介绍；第三章介绍了研究的设计，包括数据来源，研究的数据和数据分类方式；第四章介绍了对数据分析的结果以及得到的规律；而第五章通过第四章的实验结果提出了两种安全性漏洞再修复的预测方法和修复意见；第六章则对整篇论文做了总结。

第二章 背景与相关工作

2.1 漏洞生命周期

一个漏洞有一个复杂的生命周期，不论是安全性漏洞还是其它类型的漏洞基本上都遵循这个生命周期，如图 1。

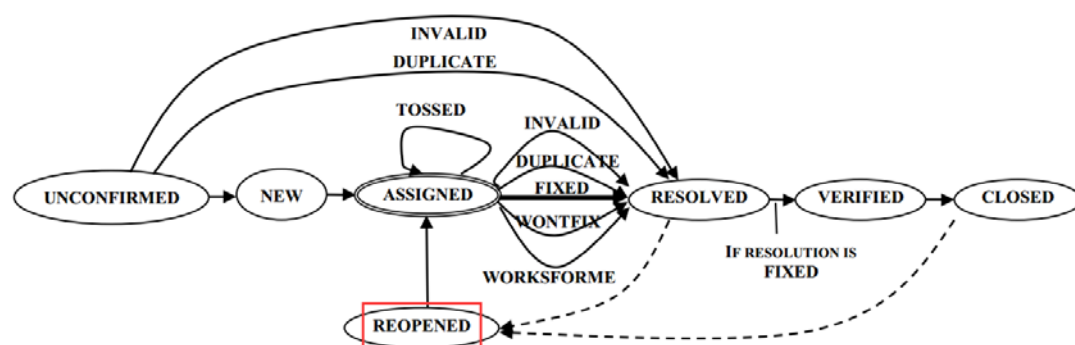


图 1 漏洞生命周期[23]

在一个漏洞被发现后会被发现者报告出来，此时只是知道这个漏洞会造成错误，但其它关于漏洞发生的根本原因、漏洞类型等信息都还不知道，这就是未确认 (UNCONFIRMED) 阶段。下个阶段就是通过错误信息对漏洞的发生原因进行分析，如果是新的漏洞，就进入这个新漏洞的各种信息获取阶段 (NEW and ASSIGNED)；如果是跟已有漏洞相同，就标注为重复 (DUPLICATE)。获取完漏洞信息之后，就要对漏洞进行修复，修复完成则进入解决 (RESOLVED) 阶段。解决阶段后面就是对修复进行验证 (VERIFIED) 阶段，验证修复是否正确，最后就关闭 (CLOSED) 漏洞。

在解决阶段和关闭阶段之后，都有可能因为修改不正确或不完整需要重新打开 (REOPENED，图中红框标注) 漏洞，重新对漏洞进行修复。这个重新打开阶段进行的工作就是再修复，这也是我们要研究的重点。

2.2 安全性漏洞分类

OSVDB (Open Sourced Vulnerability Database) 是一个为安全性漏洞提供准确、详细、及时、公正的信息的独立且开源的数据库。OSVDB 为安全性漏洞分类提供了参考，共分为六类。结合我们对数据的分析及 OSVDB 的参考分类，我们

将研究中的安全性漏洞分为七类[24]：信息泄露、拒绝服务攻击、跨站脚本攻击、缓存溢出、内存泄露、使用非法内存、其它。

信息泄露¹是指在程序运行过程中，一些数据应该被保护或加密，未经授权的用户不能得到这些数据，但这些数据在某些安全性漏洞存在的情况下，会被恶意用户获取从而给系统或正常用户造成损失。

拒绝服务攻击²是指使目标电脑的网络或资源过载或耗尽，从而使服务暂时中断或停止，导致其对客户不可用。

跨站脚本攻击³是代码注入的一种，指在网站应用程序中存在安全性漏洞，恶意用户通过这些漏洞在网页中插入恶意代码，而当用户进入、使用该网站时恶意代码会被执行，恶意代码包括 JavaScript, Java 等等。恶意代码执行之后，可能会获取更大的权限、窃取隐私信息等等，给正常用户造成影响和损失。

程序在运行时都会分配一定大小的缓冲区，而当我们写入的内容超过缓冲区大小时，就会造成缓存溢出⁴。若这种缓存溢出未被系统或程序察觉，则攻击者可能破坏程序的运行、获得程序甚至系统的控制权，进而运行恶意代码、进行一些破坏性的操作。

内存泄露⁵是指在程序中一些已经使用过且不再需要的内存未被释放，导致程序可用内存越来越少，最后会导致程序因缺少内存而部分功能被迫停止，甚至程序崩溃。这种内存的减少并不是物理上的内存减少，而是可分配使用的内存数量越来越少。

在程序中，我们经常使用指针来储存和操作数据，但如果这些指针是空指针、指向地址已经被释放或指向程序堆栈之外即不属于程序的内存地址的话，会使程序停止运行或崩溃，这就是使用非法内存⁶。

而除了以上那些安全性漏洞之外，还有一些比较特殊且数量比较少的安全性漏洞都放在其它类，比如 uri 处理问题等。

通过这些安全性漏洞分类，可针对每类安全性漏洞进行分析，研究每类安全性漏洞自身的特点。

2.3 安全性漏洞修改模式

在对安全性漏洞的分析中，我们发现安全性漏洞的修复大多数都不是很复

¹ https://en.wikipedia.org/wiki/Information_leakage

² https://en.wikipedia.org/wiki/Denial-of-service_attack

³ https://en.wikipedia.org/wiki/Cross-site_scripting

⁴ https://en.wikipedia.org/wiki/Buffer_overflow

⁵ https://en.wikipedia.org/wiki/Memory_leak

⁶ https://en.wikipedia.org/wiki/Memory_corruption

杂，且修复跟安全性非常相关，因此，对于安全性漏洞的修复，我们总结了一些修改模式。一个程序最主要的工作就是对数据进行处理，得到程序员想要的结果，而一个数据在程序中传递时，可分为三个阶段：数据进入程序，数据被处理，数据处理结束输出结果。针对数据在程序中的这三个阶段，可以在不同位置对数据做相应的保护措施。根据我们对安全性漏洞修复的分析，在三个阶段对数据的安全性修复有以下一些修改模式。

2.3.1 数据预处理阶段

在数据进入程序阶段，即数据预处理阶段，通常会对数据做一些检测和筛选，防止外部数据对程序的攻击。在数据预处理阶段总结出来的修改模式有以下这些：数据检查，数据筛选，加强数据限制，减弱数据限制。

数据检查就是对进入程序的外部数据进行检测，看是否符合程序限制、是否含有危险数据，若是危险数据，则报错或做相应处理。常见数据检查的修改模式为添加 IF 条件语句或使用 ASSERTION 语句进行判断。

数据筛选则是在一系列外部数据进入时，对这些外部数据进行检查，找出其中的非法或有危险性的数据并进行剔除，然后将合法的数据输入供程序使用。数据筛选与数据检查有一些类似，但是数据筛选需要去除非法数据，保留合法数据，而数据检查则不进行这项工作。常见的数据筛选修改模式为添加 IF 条件语句、添加循环模块，这两种修改模式可单独使用，也可能混合使用，特别是一些数组、数据集合一般都会使用循环模块进行筛选。

一些本来已经经过数据检查或数据筛选的，但是由于限制条件过弱，导致一些本来应该被剔除的数据却传入处理程序，从而使这些非法数据威胁到程序的正常运行。针对这种情况，我们要做的就是加强数据限制，使那些本来应该被剔除掉的数据得到处理。增强数据限制常用的修改模式为增强 IF 条件（如 `if(a>0)` 增强为 `if(a>0 && a<10)`），将某段程序移入 IF 模块中。

显而易见，减弱数据限制与增强数据限制是相反的过程，减弱数据限制是因为过强的限制使某些应该被程序接收到的合法数据被错误地剔除掉，使程序运行错误，需要使这些错误被剔除掉的合法数据被程序接收。减弱数据限制常用的修改模式是减弱 IF 条件，将某段代码移出 IF 模块。

2.3.2 数据处理阶段

在数据处理阶段，即程序对数据进行操作的阶段，我们需要保证数据被正确的、安全的进行处理，因此，以下这些修改模式是为了保证数据处理正确、安全完成：添加判断模块、将变量从子类变为父类、将变量从父类变为子类、添加

tryCatch 模块、将变量从弱引用变为强引用。

在数据处理时，程序需要通过添加判断模块来利用数据达到不同目的，且避免一些对程序造成影响的错误处理。常见的修改模式为添加 IF 或 SWITCH 模块。

假设在程序中两个变量 P、S，P 是 S 的父类，P 中某些数据被 S 继承。此时某些恶意代码可以通过 S 中的某些特有方法攻击到这些继承下来的数据，此时避免攻击的最好方法则是在保持功能正常的情况下将 S 变为 P（将子类变为父类），不但这些数据还存在，也避免了会被攻击的风险。

与之相反的，如果在程序中 S 通过某些方式保护了这些数据，而程序中使用的是 P，攻击者可以攻击 P。此时要做的就是将 P 变为 S（将父类变为子类），那么 P 的特性也得到了继承，那些数据也得到了安全的保护。

程序中总会存在一些存在危险性可能会出错但是现阶段还是正常的代码，针对这种代码无法进行什么有效的修改。那么，最好的办法就是将代码放入 tryCatch 模块中，这样即使代码出错也不会对程序造成太大的影响，还可以在出错时进行某种处理。

而在我们的研究中，有一种修改模式是比较特殊的，因为它只在使用非法内存类的安全性漏洞的修复中出现。使用非法内存大多数都是因为一些指针变量指向的内存地址因为是弱引用而在使用前就被垃圾收集器将那块内存收回并释放，导致后来再使用这些指针就会出错。针对这种情况，我们需要把那些弱引用变成强引用，这样可以防止这些被指向的内存地址被垃圾收集器释放，从而我们能正确使用并在不需要的时候再释放掉

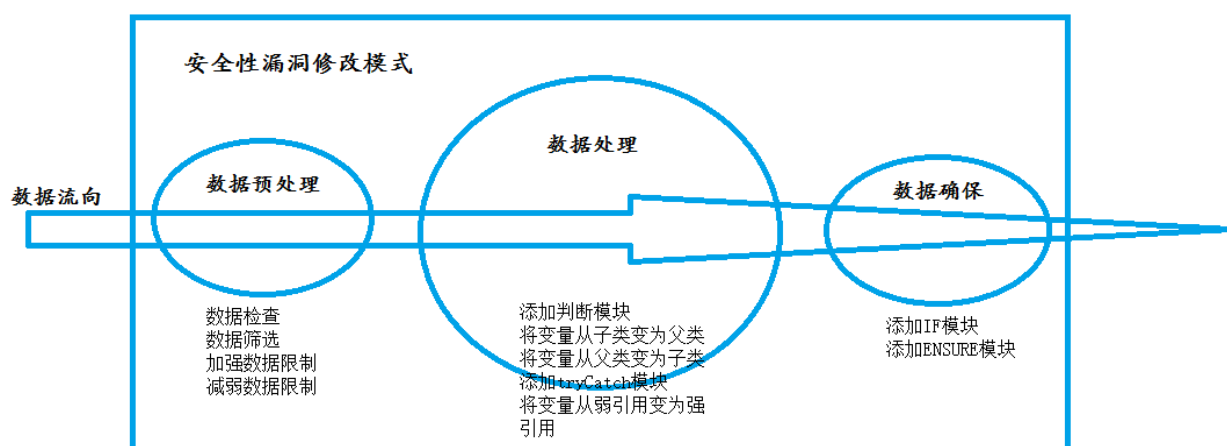


图 2 安全性漏洞修改模式

2.3.3 数据确保阶段

在数据被程序处理完成后，通常会根据这些数据输出一些我们想要的结果。

而这些结果也可能因为处理过程的不理想而出现一些错误，针对这种情况，就要进行数据确保工作，防止错误的处理结果对程序造成影响。

在数据确保阶段，最常用的修改模式就是添加 IF 模块，对结果进行判断，确认其为正确合法的结果。还有一种方法是添加 ENSURE 模块，但 ENSURE 从根本上来讲也是进行 IF 判断，只不过进行了一些包装，更方便调用。

针对以上这些修改模式（总结如图 2），可以观察在安全性漏洞再修复中与初始修复使用的修改模式有什么区别或联系。

2.4 安全性漏洞修复过程

安全性漏洞的修复有时候并不能一蹴而就，不同于功能性漏洞的修复，安全性修复可能会比较分散，再修复发生的可能性也更大。通过对安全性漏洞修复的研究，我们将安全性漏洞修复过程分为以下几类：修复—测试、修复—更好的修复、部分修复—部分修复、错误修复—正确修复、修复—重构。

最简单的一种修复过程就是修复—测试，常用于一些易于修复的安全性漏洞，且修复比较完整。

修复—更好的修复，则是因为安全性漏洞的一个特性而存在的修复过程。安全性漏洞虽然并不一定会使程序停止，但可能会在程序依旧运行时造成不可估量的损失，因此对于安全性漏洞一般都会及时快速地修复。而对于一些要完全修复难以在短时间之内完成的安全性漏洞，最常见的处理方法就是进行临时性修复，目的是使这个安全性漏洞暂时不会造成损失、或者只会造成一些较小的损失。而等有时间之后就会把这个临时性修复去掉，完全修复这个安全性漏洞，使之完全不会造成任何损失。还有一种情况就是原来的修复已经够好了，再次修复使原来的安全性防卫更加强大，提前预防了一些可能的攻击。

部分修复—部分修复，比较好理解，其实就是将一个完整的修复分成几部分，每次修复一部分，最后这些修复组成一个完整、正确的修复。分为几部分的方法可以根据修复所在文件、可以根据修复的功能，还可以是根据修复的方式等等。

错误修复—正确修复，是由于第一次修复其实并没有解决这个安全性问题，修复是错误的或者这个修复导致了其它错误，因此要重新进行正确的修复。

而修复—重构则是对设计进行了考虑，把以前设计不好的修复、或者设计不好的原始代码进行重构，使之更便于以后的维护和更新。

可以通过观察这些修复过程在每次再修复前后到底有什么联系，找出哪种修

复过程最容易发生再修复，或者哪种修复过程在再修复中最常用等规律。

2.5 相关工作

虽然并没有与本文相同的研究工作，但是可以通过一些相关工作获取本文研究需要的知识、方法，为研究打下基础。

Pamela Bhattacharya 等人通过对漏洞报告和漏洞修复的经验研究，识别出漏洞报告的优劣性，找出了漏洞生命周期对漏洞修复的影响 [25]。

Michael Gegick 等人则通过对漏洞报告自然语言描述进行自动化分析，能够找出其中的安全性漏洞报告，避免了浪费在寻找非安全性漏洞报告的时间 [11]。

丁羽等人通过对学术界关于漏洞分类学 (Taxonomy) 和漏洞分类 (Classification) 的调研，总结了安全性漏洞的分类的几个关键性问题 [14]。

Zhenmin Li 等人使用自然语言处理分类工具对两个大型开源项目约 29,000 个漏洞进行自动化分析，找到了漏洞的一些特性和规律，并发现了安全性漏洞数量正在增加且大部分会造成严重的损失 [12]。

管铭等人通过静态分析和动态分析两种程序分析技术研究了在软件开发周期中安全性漏洞检测的技术。文章利用软件分析技术检测安全性漏洞的方式对我安全性漏洞再修复的预测有一定启发性 [8]。

Yonghee Shin 等人通过九种复杂性度量标准找到安全性漏洞与非安全性漏洞之间的区别，并对安全性漏洞进行预测。文章中提供了安全性漏洞分类的参考，一个漏洞各修复过程的特点 [26]。

T Zimmermann 等人通过开发人员对再修复原因的分类、漏洞报告、修复人员的关系等方面利用静态分析模型对漏洞再修复性质与预测方式进行研究，其中的分类与预测方法给我研究提供了参考性建议 [21]。

Shahed Zaman 等人通过对 Firefox 工程中的漏洞，比较安全性漏洞和功能性漏洞在各方面的不同之处和它们各自的特点。研究发现，安全性漏洞修复的更快、修复时需要更多的开发人员、涉及到更多的文件，但是安全性漏洞再修复发生得更频繁 [1]。

通过对漏洞分类、安全性漏洞分类、安全性漏洞特点研究、安全性漏洞检测、漏洞再修复预测等工作进行学习和总结，我们借鉴了其中一些优秀的知识、规律和研究方法，设计了我们的安全性漏洞再修复的研究数据和研究方法，找到了通过对安全性漏洞的再修复的研究我们想要达成的目标。下面我们对本文用到的一些基础知识进行了介绍，这些基础知识被用于进行数据的分析工作。

2.6 本章小结

本章通过介绍漏洞的生命周期,了解漏洞整个修复流程有助于了解各阶段漏洞的特点。而根据收集到的数据分析及参考资料的推荐,我们对研究中涉及到的安全性漏洞进行了自己的分类研究,为后面根据不同类型的安全性漏洞研究打好基础。接下来介绍了安全性漏洞修复时常用的修改模式和修复过程,通过这些能为自动化安全性漏洞再修复提供理论支持。最后,对现有的相关工作进行了介绍,这些相关工作为研究提供了知识参考。

第三章 研究设计

3.1 数据来源

安全性漏洞是社会关注的热点，因此有很多组织机构对安全性漏洞进行关注并对发生的安全性漏洞的信息进行记录和维护，如 SecurityFocus (www.securityfocus.com/)、bugzilla (<https://bugzilla.redhat.com/>)、National Vulnerabilities Database (<https://nvd.nist.gov/>)、Mozilla (<https://www.mozilla.org/en-US/security/advisories/>) 等等。通过观察对比，bugzilla 和 National Vulnerabilities Database 中 bug 种类比较繁杂，产品种类也很多样化，不利于分类分析，而 SecurityFocus 则是缺少修复相关信息。因此，我们选定 Mozilla 作为研究安全性漏洞再修复的数据来源，它维护的全部都是 Mozilla 产品的安全性漏洞，且信息全面清晰，还能在 git 找到提交信息，使得研究数据丰富全面。

3.1.1 安全性漏洞基本信息

在对安全性漏洞的修复工作中，Mozilla 维护了一个安全性漏洞管理的社区，还为招聘安全性漏洞识别和修复的岗位投入了大量资金。在网站 (<https://www.mozilla.org/en-US/security/advisories/>) 中，管理了 2005 年至今为止 mozilla 发现的所有安全性漏洞的相关信息。

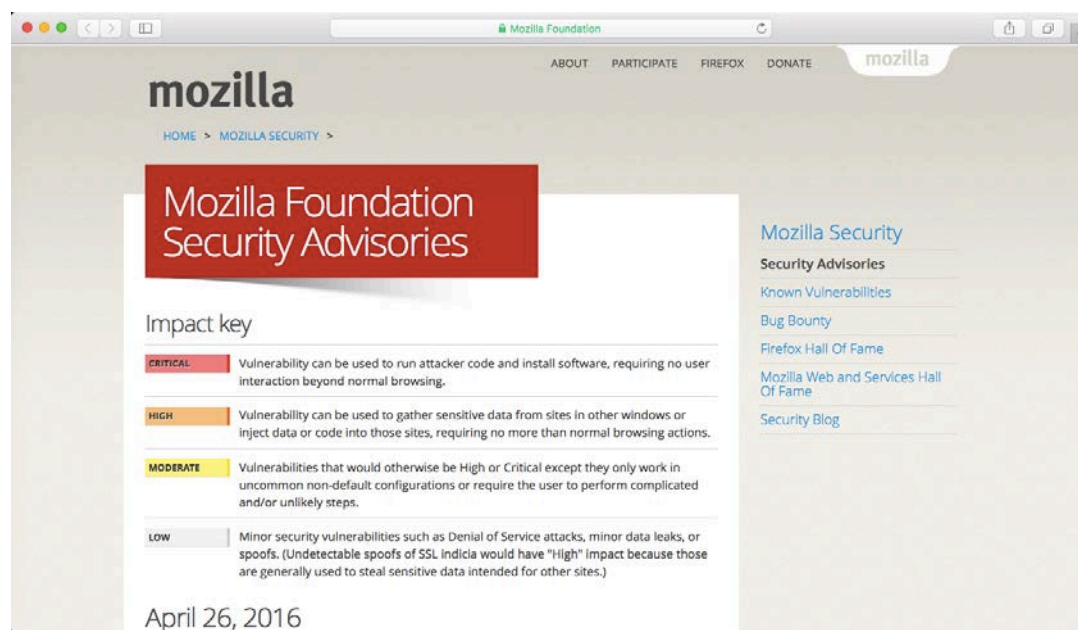


图 3 所有安全性漏洞管理网站

如图 3 所示, 页面一开始介绍了安全性漏洞的严重性级别; 图 4 中每一条安全性漏洞颜色代表严重性, 信息包含了该安全性漏洞发布日期和标题, 点击条目可跳转至具体信息页面 (图 5)。具体信息包括安全性漏洞标题、发布日期、发布人, 严重性、发生该漏洞的产品、修复涉及的产品、简要描述和漏洞报告的链接 (图 5 中红色方框处, 点击可跳转至该安全性漏洞报告)。虽然这里有一些漏洞相关信息, 但更详细、更多样化的信息还得通过漏洞报告来获取。

通过点击漏洞报告的链接, 跳转到相应漏洞报告页面, 如图 6 所示。在漏洞报告中有许多信息, 其中对研究有用的是以下几个信息: 漏洞 id、漏洞描述、漏洞修改历史记录 (点击可跳转至详细页面)、漏洞修复评论 (图 7)。

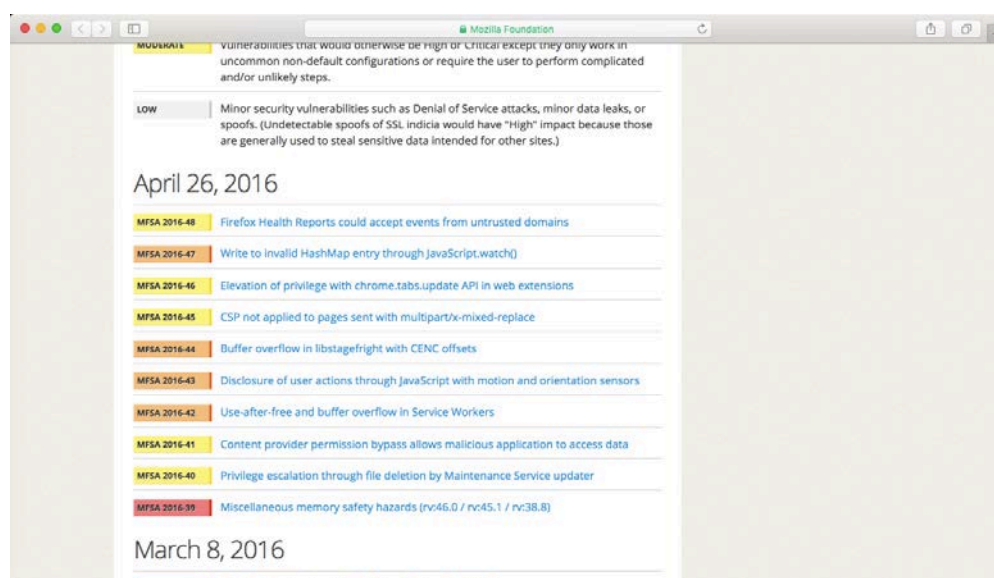


图 4 安全性漏洞条目

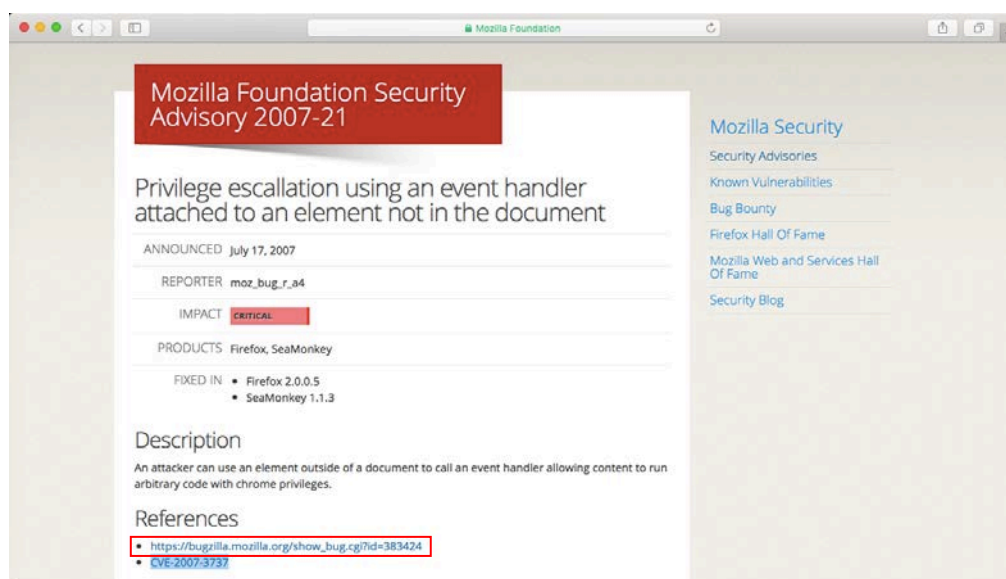


图 5 安全性漏洞具体信息

在漏洞修复评论中, 可以了解到修复人员的思考过程, 从中得出这个安全性

漏洞产生的原因、修复时对错误信息的分析、使用该种修复方式的原因、对漏洞再修复的原因、修复完成后进行了什么测试等等信息。通过对这些信息进行分析，一个安全性漏洞修复的全过程就清晰明了，从而找出安全性漏洞的类型、漏洞再修复的原因、漏洞每次修复使用的修复方式及原因、每次修复之间的关系等等。

漏洞修改历史记录有在每个时间点该漏洞处于何种状态的信息，如图 8，表中第一列表示做操作的人，第二列表示该操作进行的时间，第三列表示做了什么操作，第四列表示某状态被移除，第五列表示某状态（review, resolved, reopen, fixed 等）被添加。在第四列中出现“REOPEN”则表示该安全性漏洞被再修复，而在接下来的状态中第四列出现“FIXED”则表示该再修复完成。

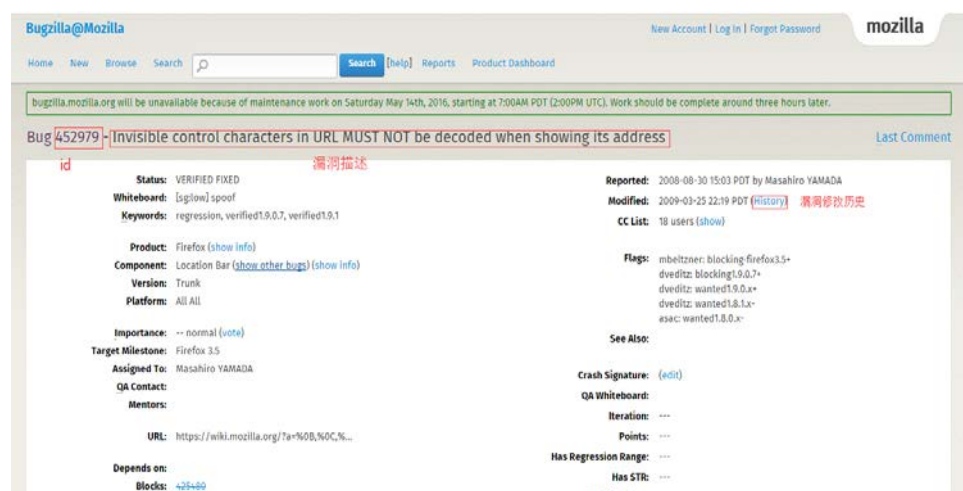


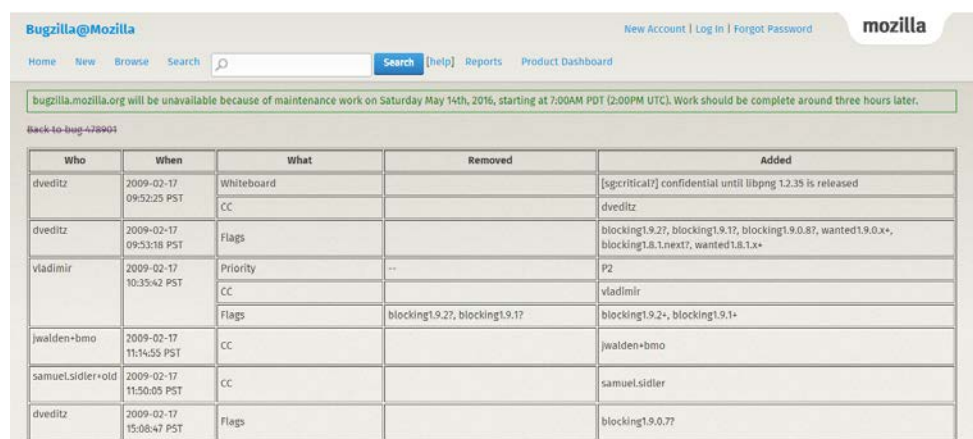
图 6 漏洞报告信息



图 7 漏洞修改评论信息

以上是进行安全性漏洞再修复分析需要的部分数据来源，但由于数据量过大，人工获取所有数据进度较慢，自动化数据获取是必须的。因此，我们使用 python 语言结合 Scrapy 框架编写了一个爬虫程序，爬虫程序首先对从图 4 页面

进入，获取每个安全性漏洞相应的图 5 页面 url，通过这个 url 跳转到该页面，然后用正则表达式匹配在图 5 页面的 html 中进行匹配，获取相应 id 和描述，找到该安全性漏洞的漏洞报告 url，再次通过这个 url 跳转到图 6 页面，在图 6 页面 html 中再次通过正则表达式匹配得到安全性漏洞修改时间，修改者等信息。同理，最后跳转到修改历史记录页面，若在第四列中发现“REOPEN”字样，则记录下该安全性漏洞我们所需要的信息，否则从头开始进行下一个安全性漏洞的数据获取。



Who	When	What	Removed	Added
dveditz	2009-02-17 09:52:25 PST	Whiteboard		[sgcritical?] confidential until libpng 1.2.35 is released
		CC		dveditz
dveditz	2009-02-17 09:53:18 PST	Flags		blocking1.9.27, blocking1.9.17, blocking1.9.0.87, wanted1.9.0.x+, blocking1.8.1.next17, wanted1.8.1.x+
vladimir	2009-02-17 10:35:42 PST	Priority	--	P2
		CC		vladimir
		Flags	blocking1.9.27, blocking1.9.17	blocking1.9.2+, blocking1.9.1+
jwalden+bmo	2009-02-17 11:14:55 PST	CC		jwalden+bmo
samuel.sidler+old	2009-02-17 11:50:05 PST	CC		samuel.sidler
dveditz	2009-02-17 15:06:47 PST	Flags		blocking1.9.0.77

图 8 修改历史记录表

下面是一小段爬虫代码的示例，第一行是类名，第二行是一开始爬虫要处理的页面，第三至七行则是进行跳转操作后通过漏洞报告页面通过正则表达式匹配获得相应数据（代码非完整，仅选择代表性代码展示）。

```

1 class MySpider(CrawlSpider):
2     start_urls = ['https://www.mozilla.org/en-US/security/advisories/',]
3     def parse_item(self, response):
4         hxs = HtmlXPathSelector(response)
5         products = hxs.select("//div[@id='bugzilla-body']")
6         for product in products:
7             bug_id = product.select("//div[@class='b
                z_alias_short_desc_containeredit_form']/a[contains(@href,'id=')]
                /@href").re(r'id=(\d*)')

```

3.1.2 安全性漏洞代码提交信息

要对安全性漏洞再修复进行分析，那么在每次修复时代码的提交信息是必不可少的。Mozilla 将他们的项目提交于 git 上，通过 git 这个方便的版本控制系统得到了 Mozilla 工程的每次提交的漏洞 id、代码、提交者、提交时间、修改

的文件、本次提交与上次提交的代码更改信息、修改相关信息等。但是，git 上的 Mozilla 工程只持续到 2011 年，因此能得到的提交信息年份为 2005~2011。

然后，我们通过 GitPython 接口连接到 Mozilla 工程 git 库，获得该工程的相关提交信息，再从数据库中获得发生了再修复的安全性漏洞 id，提取其在 git 中的提交信息，通过这些提交信息对安全性漏洞的修复进行有效的研究。下面是一小段示例代码，第一、二行是 GitPython 获得 Mozilla 工程 git 库的连接，第三行是一段对获得提交信息的操作示例。

```
1 repo = Repo("D:\mygit\mozilla-history")
2 commit = repo.commit('master~0')
3 diff = repo.git.diff(commit.parents[0], commit).encode()
```

以上就是分析安全性漏洞再修复用到的所有数据来源，根据这些数据做一些分析、总结，从而找出利于安全性漏洞再修复改善、预测的规律。

3.2 研究内容

通过 Mozilla 安全性漏洞社区和 git 项目数据，总共获取到 48 个可用的发生过再修复的安全性漏洞相关数据。这些安全性漏洞 id 范围从 312278 至 672485，时间跨度为 2005 年至 2011 年。因为 Mozilla 工程在 git 上的提交日期只到 2011 年，因此研究的再修复安全性漏洞截止到 2011 年，没能研究到最新的数据，这有可能会造成某种类型的安全性漏洞较少。分析数据包括三方面的数据(如图 9)：漏洞的整体情况、初始修复和再修复的对比数据、修复过程中用到的数据。

漏洞的整体情况数据包括漏洞的 id，漏洞的类型，发生再修复的原因，在 git 上这个漏洞的修复提交的总次数，所有修改的文件数，所有修改增加和减少的代码行数，发生再修复的次数。这些数据有助于分析安全性漏洞的基本信息中隐含的修复规律。

初始修复与再修复的对比数据包括：两次修复时间的长短对比，两次修复使用的修改模式的对比，修改者的对比，两次修复使用的修复过程的对比，两次修复修改的文件和文件中修改的内容的对比。通过这些修复的对比数据可以得到再修复和初始修复之间的关系，通过初始修复推断再修复的修改模式、修改文件与内容、修复时需要注意的地方，提供更方便的再修复方法。

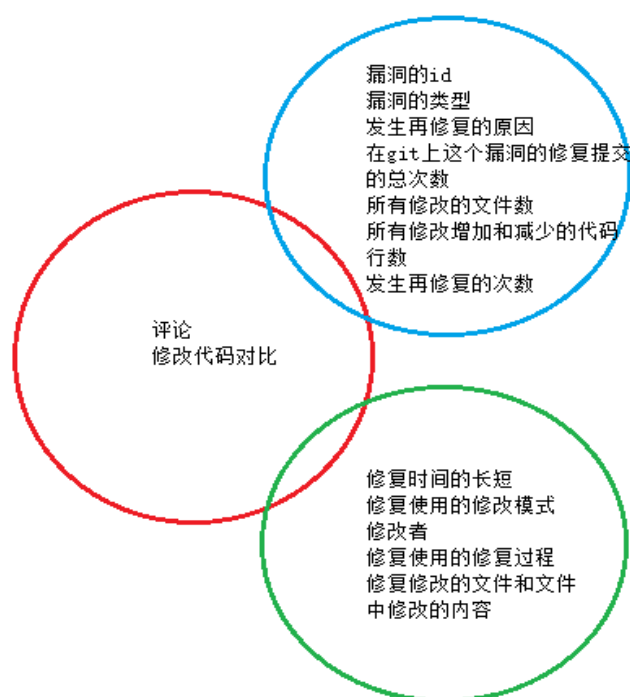


图 9 研究数据汇总

而修复过程的数据如评论、提交的代码对比，可以得到修复者的思路，从中找出安全性漏洞类型、再修复原因、修改模式、修复过程等等。这方面数据是得到上面两种数据的基础。上面两种数据有些就通过对修复过程中的数据的统计、分析获得。

研究的收集数据阶段首先获得修复过程的数据，然后统计一些网站上可以得到的数据，结合起来分析、总结出漏洞整体情况和两次修复对比的数据，完善这个表格中的所有数据，然后根据这些数据结合一些比较感兴趣的情况进行分析。

3.3 分类方式

我们的研究通过两种方式进行分类：安全性漏洞的类型和安全性漏洞再修复的原因（4.2 节）。通过这两种分类方式统计不同的数据，找到不同类型的规律。

通过安全性漏洞的分类方式，我们可以统计安全性漏洞再修复发生的次数，再修复的原因，这个安全性漏洞提交的总次数，总的文件修改数，增加的代码行数和删除的代码行数。在这些数据的支持下，我们想要对以下问题进行研究：安全性漏洞整体情况分析；不同类型的安全性漏洞较容易发生哪种再修复；不同类型安全性漏洞在修改时修改文件、修改的代码数量等等有什么规律；

通过安全性漏洞再修复的原因进行分类，我们收集初始修复与再修复两方面的数据：使用的修改模式，修复过程，修复所用的时间，修改者，两次修改对文件及代码的影响。通过收集到的数据研究的问题如下：安全性漏洞再修复的原因；再修复和初始修复使用的修复过程有什么联系；两次修复修改时间和修改者有什么联系；两次修复使用的修改模式有什么联系；两次修复修改的文件和内容有什么联系。

3.4 本章小结

本章通过数据的来源、要研究的数据和数据的分类三个方面对研究的设计进行了介绍，整体研究设计就是对收集到的数据进行分类研究，对每种分类的研究都能得到特有的结果和规律，这些规律为再修复预测的设计提供指引和支持。

第四章 数据分析

4.1 基础数据统计

我们总共收集到的安全性漏洞信息为 721 条，而发生再修复的安全性漏洞为 48 条，在 Mozilla 工程中，安全性漏洞发生再修复的频率约为 6.7%。在所有漏洞中，这已经算是一个比较高的发生再修复的频率。

安全性漏洞在所有 id 和时间段都有发生再修复，由此可见，安全性漏洞发生再修复是一个普遍存在且发生频率也比较高的问题。通过对安全性漏洞再修复的信息进行分析，找到减少再修复、预测再修复、帮助再修复的方法，有助于提高程序安全性，减少开发、维护资源消耗，使程序员在对安全性漏洞进行修复时更加全面、正确，减少对安全性漏洞再次修复的次数。

4.2 安全性漏洞再修复原因

安全性漏洞为什么会发生再修复，可以根据初始修复和重新打开后的再修复的对比找出其中的关系，总结其中规律。在对数据进行分析后，我们总结出以下几个发生再修复的原因：加强漏洞填补（范围加强、强度加强）；减弱漏洞的填补；填补原来没想到的漏洞（线性关系、并列关系）；对修复进行登记或测试；以前修复无效或错误而重新修复；怀疑修复造成了错误，后来发现不是。

4.2.1 加强漏洞的填补

在修复安全性漏洞时，有时候可能是临时性填补，安全性还不够高，还存在着一些安全隐患，后续需要对漏洞重新修复，提供更高的安全性防护；又或者可能是修复的范围不够广，同一个问题在不同的地方同样出现，此时只要将修复延伸至更大的范围即可。这些再修复就是对漏洞填补的加强。加强漏洞的填补可根据加强的再修复和原修复之间的关系分为：加强强度、加强范围。

可以对安全性漏洞进行临时性填补是安全性漏洞修复的一个特点，临时填补的目的是为了在没有时间和资源进行完全修复时，尽量减少安全性漏洞的作用力和危害，甚至用一些非常糟糕、后期难以识别维护的修复来完全修复漏洞。当然这样做只是暂时是软件能继续运行，不至于因为软件停运造成损失，在有时间和资源时就要对漏洞进行重新修复，保证修复的完整性与安全性，这就是加强漏洞

修复的强度。

例如安全性漏洞 329385⁷，攻击者可以利用这个安全性漏洞进行强制性鼠标操作。初始修复时对 UI 进行修改，使与这个漏洞相关的 UI 界面暂时停止相关功能。这样修复虽然是可以解决问题，但是一些功能无法使用，且真正能够强制操控鼠标操作的原因并没有找到。而在再修复中，找到了真正的原因，进行了修复，也使 UI 的相关功能得以正常运行。

另一种对漏洞修复的加强是范围的加强，对于某个安全性漏洞的修复方式是正确的，但是修复作用的范围未完全包含所有出问题的地方，再修复时只需要将之前的修复作用于剩下的地方即可成为完整的修复。一个贴切的比喻就是一扇窗户是打开的，想要使雨不能进来就完全关上窗户。窗户就是修复，漏雨就是安全性漏洞，通过把窗户完全关好来挡住雨。其实窗户已经完成并一直在那里，要做的只是把窗户关到所有开口的地方。

例如安全性漏洞 370127⁸，传递了某个不安全的方法作为新方法的父方法，而这个新方法如果被调用的话，会由于父方法的不安全产生安全性问题。一开始的判断是 js 代码会调用到这个新方法，因此初始修改是在 js 调用这个新方法是做一些安全性防护，保证调用时不会传入不安全的父方法。后来却发现不但是 js 代码会调用这个新方法，c++代码某功能也会调用这个新方法，因此再修复要做的事情就是将原来的防护方法作用于 c++的调用，使得不论是哪种调用都能安全的完成。

4.2.2 减弱漏洞的填补

与增强相反，减弱漏洞的填补也是安全性漏洞再修复的原因之一。在修复安全性漏洞时，如果对漏洞的防护太强，可能会导致一些功能被限制，一些合法数据被阻拦过滤，这样依旧会导致安全性问题。

例如安全性漏洞 419848⁹，js 通过导入一些脚本作为其内容来使用，可是可能会导入不安全的脚本，在系统权限下对程序造成危害。因此，初始修复就对 js 能导入的脚本进行了限制，某些 url 的脚本不能被导入。可是这种限制太强，导致有一些合法的脚本但是却因为含有相应 url 而不能被导入，导致程序不能正常运行。再修复就是使一些 url 含有“chrome:/" 导入后也不可能获得系统权限且被程序所需要一些脚本能够导入，保证程序安全地正常运行。

4.2.3 填补原来没想到的漏洞

一个安全性漏洞的发生可能是因为好几个地方有安全性问题，这些不同地方

⁷ https://bugzilla.mozilla.org/show_bug.cgi?id=329385

⁸ https://bugzilla.mozilla.org/show_bug.cgi?id=370127

⁹ https://bugzilla.mozilla.org/show_bug.cgi?id=419848

的安全性问题导致了同样的安全性漏洞。而在安全性漏洞修复过程中，只修复了其中一部分安全性问题，有一部分未发现或被忽略，那么这个安全性漏洞就是修复不完整，因此会发生再修复。

因为填补原来没想到的漏洞而发生的安全性漏洞可以根据那些安全性问题之间的关系分为两种情况，一种是原修复和再修复是线性关系。这就好比一条河流发生洪水，如果在中流进行抗洪，并不一定能达到最好的效果，而如果找到发生洪涝的源头，在源头治洪，效果会好上很多。例如安全性漏洞 312278¹⁰，这个漏洞发生的原因是某变量在未使用之前就被垃圾收集器释放掉了它所指向的那块内存地址，因此在后面使用该变量时就是使用非法内存。第一次修复是在发现内存被垃圾收集器释放掉的那个地方管理那个变量，使之不能被释放掉。但是后来发现这个变量被释放掉的源头不是这里，变量在被管理起来之前就有可能被垃圾收集器释放掉。因此，再修复就是找到变量的源头，在那里对变量进行管理，这样变量就不会被释放掉了。

而另一种情况就是初始修复与再修复是并列关系，初始修复和再修复解决的安全性问题不是相同的方法或功能，且修复方式可能不一样，但是都会导致这个安全性漏洞。这些并列的安全性问题可能是因为难以全部找到或者经常容易被忽略，导致需要再修复的发生。例如安全性漏洞 360529¹¹，这个漏洞是因为可以通过某些方法获取到浏览器的权限运行一些恶意代码，是跨站脚本攻击。那这个安全性漏洞的修复方式自然是找出这些方法，对他们进行保护，防止被攻击者利用。初始修复是对 `Array.prototype` 和 `document.write` 进行保护，然后以为成功修复了这个漏洞。但是后来发现 `appendChild()` 和 `removeAttribute()` 也会有这种问题，因此第二次修复就是对这些方法进行保护，但其实对每个方法进行保护的方式并不相同。

这种再修复原因与加强漏洞修复的范围（4.2.1 中）的区别在于用到的修复方式的不同。加强漏洞修复范围初始修复和再修复运用的修复方法是一样的，只是作用到的地方发生了变化。而填补原来没想到的漏洞（并联关系）则是初始修复和再修复运用的修复方式可能不同，针对的安全性问题也不同，但是这些安全性问题都会导致同一个安全性漏洞。

4.2.4 对修复进行登记或测试

有一些重新打开安全性漏洞并不是因为原来的修复有什么问题，而是对完成的修复进行登记，比如验证、加入真正项目中；又或者是对其进行测试，保证修复的安全性 with 正确性。这种再修复不涉及到原修复的改动，而是对原修复做一些

¹⁰ https://bugzilla.mozilla.org/show_bug.cgi?id=312278

¹¹ https://bugzilla.mozilla.org/show_bug.cgi?id=360529

操作。

例如安全性漏洞 410156¹²，这个安全性漏洞初始修复就已经完全解决了安全隐患。漏洞重新打开之后添加了一些测试，使修复能在更多条件下被检验，看是否仍然安全有效。而安全性漏洞 441169¹³不仅在再修复时加入了一些测试，还将这个修复添加了已验证状态并加入到真正项目中，这些就是对修复的一些操作、运用。

4.2.5 以前修复无效或错误而重新修复

上面发现的一些再修复原因都是初始修复有一定正确性，能起到一定作用，且再修复时不会完全推翻初始修复的修复方法。可是有时候对安全性漏洞的修复并不是那么简单的，只要发生安全性漏洞的原因分析不正确或者修复方式错误，都有可能看上去虽然问题解决了，但其实只是暂时的假象，问题不久又会再次发生或者还可能引起其它错误，因此要推翻以前修复进行重新修复。这种安全性漏洞的再修复一般会伴随着对初始修复的回退，然后将正确的修复替换掉初始修复。

例如安全性漏洞 349527¹⁴，初始修复完成之后发现会有编译错误，还可能产生数据泄露的危险，因此对初始修复进行了改正，使那些因修复而产生的安全性问题都得到解决。

4.2.6 怀疑修复造成了错误，后来发现不是

最后一种安全性漏洞再修复发生的原因其实算是一种误诊，因为安全性漏洞的发生原因复杂且难以找寻，因此如果初始修复后程序产生了某些问题，程序员有很大可能会对这个修复产生怀疑，觉得是这个修复造成了这些问题。于是，程序员就会对这个修复进行回退，回退之后如果发现问题依旧存在，说明并不是修复带来的问题。因此，这种原因产生的再修复一般是先对修复进行回退，然后发现不是修复存在问题后又把原来的修复恢复。

4.2.7 本节小结

我们发现的一些安全性漏洞再修复的原因，可以用一个实际生活中的比喻来更清晰的说明初始修复和再修复之间的关系、这些原因之间的关系。假设羊圈里面保护着羊，防止狼来把羊吃掉。那么，羊圈就是安全性保护，里面的羊就是受到保护的程序，外面的狼就是攻击者，羊圈上的洞就是安全性漏洞。首先，如果

¹² https://bugzilla.mozilla.org/show_bug.cgi?id=410156

¹³ https://bugzilla.mozilla.org/show_bug.cgi?id=441169

¹⁴ https://bugzilla.mozilla.org/show_bug.cgi?id=349527

羊圈上现在有一个洞，为了防止狼晚上来攻击羊，只能在有限的时间中做一些临时性的填补，比如用麻袋挡一下，等第二天时间充足再糊上水泥，这就是增强漏洞填补的强度。而也有可能暂时用水泥糊上一部分漏洞，减小狼可以通过的大小说不定能挡住体型较大的狼，后面再把整个漏洞填完，这就是增强漏洞填补的范围。如果这个羊圈有很多层墙，在某一层墙上有多个漏洞，但是我们只注意到其中一部分并进行了填补，那么剩下的漏洞还会使羊遭到攻击，只有填补完这层墙上的所有漏洞才能保护好羊，这就是填补原来没想到的漏洞中的并列关系。而只填补好了最外层墙的这个漏洞，那狼可能绕过最外层墙通过里层墙的漏洞攻击羊，因此找到最里层墙填上漏洞才能真正防止羊被攻击，这就是填补原来没想到的漏洞中的线性关系。而如果在填补漏洞的过程中，使用了过多的水泥，可能使整个羊圈不堪重负而倒塌，或者影响到羊圈某个部件的功能，因此要减少填补的水泥到适量，这就是减弱漏洞的填补。有时候修补某个洞却发现这个洞根本不是漏洞，也不能被攻击到，可能是用来喂食或排水的，因此要把堵住的洞重新疏通，找到真正的漏洞，这就是原来的修复无效或错误，重新修复。

我们统计的共 48 个发生了再修复的安全性漏洞中，加强漏洞的填补共 11 个，其中强度加强 7 个，范围加强 4 个；减弱漏洞的填补共 3 个；填补原来没想到的漏洞共 14 个，其中并列关系 13 个，线性关系 1 个；对修复进行登记或测试共 7 个；以前修复无效或错误而重新修复共 11 个；怀疑修复造成了错误，后来发现不是共 9 个。

从统计数据可知，除了减弱漏洞的修复数量较少以外，其他再修复的原因个数都比较平均，再修复时一般会修复的比较适度或未修复完全，因此修复过强的情况比较少。修复过强则表示安全性漏洞发生的原因分析正确，且修复方式也是正确的，只是限制过于强烈，避免这种再修复最好的方法就是衡量好限制的强度与影响。而在填补原来没想到的漏洞中，线性关系的安全性漏洞也比较少，这种再修复的发生是对数据源头判断的不准确或未考虑数据的来源，修改也比较简单，只要找出数据的源头并添加保护机制即可。上面的数据中有些类型再修复数据量较少，对发现普遍性规律有一定的影响，偶然性大大增加。

4.3 安全性漏洞类型与再修复原因

通过统计安全性漏洞类型与再修复的原因的数据，即每种类型对应的再修复原因有多少个，可以得到哪种类型的安全性漏洞更容易发生哪种再修复。

通过统计结果（表格 1），我们可以得到一些基础规律。在七种安全性漏洞中，数据泄露和拒绝服务攻击没有发生再修复的情况，首先拒绝服务攻击在安全性漏

洞中也是比较少的，所以没发生再修复可能也有这方面原因。另一方面，这也说明这两种安全性漏洞要么是比较简单、易于修复；要么就是修复方式是比较固定的，能够轻易找出所有要修复的问题并进行正确的修复，因此很少发生再修复。由此可见，在修复数据泄露和拒绝服务攻击的安全性漏洞时，如果能找到一个通用的发生漏洞的原因、修复漏洞的方式，就是使得这两种安全性漏洞基本上不会发生再修复。

表格 1 类别与再修复原因统计

安全性漏洞类别	类别数量	范围加强	强度加强	减弱漏洞填补	并列关系	线性关系	登记或测试	无效或错误	怀疑但不是
缓存溢出	12	1	2		2		2	2	4
内存泄露	3		1					2	
使用非法内存	10	2	2		4	1	1	1	3
跨站脚本攻击	21		2	3	6		4	6	1
其他	2	1			1				1
总计	48	4	7	3	13	1	7	11	9

跨站脚本攻击发生再修复的次数最多，内存泄露发生的次数最少，其他的都比较平均。跨站脚本攻击发生再修复的原因中，剔除掉 4 个登记或测试和 1 个误诊，基本上每种再修复的原因都有涉及，说明跨站脚本攻击的安全性漏洞发生原因比较复杂，修复的时候很容易出现遗漏或错误，修复方式也难以做到全面保护，在修复这类安全性漏洞时需要投入更多的精力进行这方面分析。特别的，仅有的三个减弱漏洞的填补的再修复都是跨站脚本攻击的漏洞，首先说明减弱漏洞的填补这种再修复本身就很少，其次说明了跨站脚本攻击安全性漏洞的修复方式较其他类型的安全性漏洞更容易发生修复过强的情况。在我们以后对跨站脚本攻击漏洞进行修复时，注意修复的强度与范围，防止其影响到其他功能，有助于减少这种再修复的发生。

内存泄露一般是因为内存使用和管理不当而产生的，因此在内存使用时记得进行分配与释放，内存泄露的安全性漏洞就不会发生，所以内存泄露的安全性漏洞发生原因并不复杂，修改方式也比较简单，数量较少。

可以看出，安全性漏洞发生再修复的可能性与两个因素是有关系的，第一就是发生原因寻找的难度，找到发生的是哪种类型的安全性漏洞，为什么会发生这

种安全性漏洞，发生错误的代码在什么文件、什么位置，这些信息越复杂，越容易忽略或遗漏一些信息而导致再修复；第二就是修复方式的复杂程度，比如修改的文件的多少，这些修改文件之间的联系，修改的代码行数的多少，要分多少次进行修复，修复方式越复杂、修复时越草率，发生再修复的可能性就越大。

4.4 修复基础信息统计

如表格 2 所示，在总共 48 个发生再修复的安全性漏洞中，37 个发生了一次再修复，10 个发生了两次再修复，只有 1 个发生了三次再修复，没有超过三次的再修复。在修改安全性漏洞时，发生一次在修复可能是对漏洞的理解有遗漏，发生两次就需要程序员特别注意是不是自己的修复根本就不对，或者对漏洞的理解错误。唯一一次发生了三次再修复的漏洞是对漏洞进行登记和测试，因此基本上不会因为修改原因而发生超过两次的再修复。多次发生再修复不仅对资源造成了浪费，还会提升系统危险性，给攻击者更多机会。

表格 2 安全性漏洞修复基础数据统计

安全性漏洞类别	类别数量	再修复 1 次	再修复 2 次	再修复 3 次	git 提交总数	修改文件数	减少代码行	增加代码行
缓存溢出	12	8	4		3	3.7	45	208
内存泄露	3	2	1		3.7	2.7	59	111
使用非法内存	10	8	2		3.4	6.7	56.2	119.7
跨站脚本攻击	21	18	2	1	3.3	4.7	57	118
其他	2	1	1		5	1.5	6.5	17
总计	48	37	10	1				

发生两次再修复较多的安全性漏洞是缓存溢出，发生一次再修复比较多的是跨站脚本攻击，其它的比例都比较接近。而 git 提交的平均次数几种类型都比较接近，修改的平均文件数使用非法内存最多，内存泄露最少。在减少和增加的代码行数方面，内存泄露和缓存溢出分别为减少和增加代码行最多的，缓存溢出和内存泄露分别为减少和增加代码行最少的。其它安全性漏洞（两个都是 uri 问题）git 提交数较多，但修改文件数、减少和增加的代码行数都是偏少的。

缓存溢出和内存泄露虽然涉及的文件数较少，但是修改的代码行数却是较多

的,说明这两类安全性漏洞问题比较集中,修复时涉及的功能关联性比较强,不会很分散。而使用非法内存和跨站脚本攻击涉及的文件较多但是修改的代码行数较少,说明这两种安全性漏洞涉及的功能较为分散,修改的文件范围较广,修复时就比较容易有遗漏,考虑容易不够全面。

4.5 初始修复与再修复对比

4.5.1 修复过程的对比

根据统计的数据,基本上所有发生再修复的安全性漏洞初始修复时都是一次修改和提交,使用的修复过程是修复一测试,而在怀疑修复造成问题但发现不是的初始修复中,使用了部分修复一部分修复、修复一更好的修复两种修复过程。这说明仅仅使用修复一测试这种修复过程难以准确全面的进行安全性防护,导致了再修复的发生。

在对漏洞进行登记或测试的再修复中,基本上只有一次修改和提交,因为再修复并不涉及代码的修改。再修复中使用除了修复一测试外其它修复过程比较多的是以前修改错误或无效而重新修改,因为这种再修复是推翻以前的修复进行全新的修复,因此就相当于修一个新的安全性漏洞,通过使用这些修复过程来提高修复的安全性,避免二次再修复发生。

使用更复杂、更有效的修复过程有助于全面找到安全性漏洞发生的原因和进行更有效、更正确的修复,减少再修复发生的可能性。而再修复相较于初始修复更为简单,是对初始修复的查缺补漏。

4.5.2 修改时间和修改者的对比

根据统计的数据(表格 3)可知,除了以前修复无效或错误而重新修复的安全性漏洞,其他类型的再修复时间都相对于初始修复时减少的情况比较多。这说明只要不是以前修复完全无用,初始修复对再修复有着参考性的作用,可以帮助寻找被遗漏的那部分安全性问题、提供修复方式的参考。

强度加强和并列关系都有一定数量的再修复时间更长的情况。并列关系其实是修复一些并列的安全性问题,且修复方式可能不同,安全性问题的导致原因也可能不同,因此需要更多的时间。而强度加强有可能是将原来的修复去除加入新的安全性更高的修复,所以修复时间比原修复时间长是有可能的。

在修改者对比方面,减弱漏洞的填补、对修复进行登记或测试、怀疑修复造成错误但不是这三种再修复基本上都是非同一个修复者,其他的再修复同一修复

者占大部分。其中测试和登记漏洞一般都专门的人员，修复人员不负责测试与登记的工作，而其他两种则一般是别的修复人员发现了错误，然后找到这个安全性漏洞，对这个安全性漏洞进行再修复。剩下的再修复类型则是因为如果同一个修复者再来进行再修复就不用重新熟悉这个安全性漏洞，对漏洞发生的原因、原来的修复方式都有一定的了解，可以有更高的再修复的效率，节省时间与资源。

表格 3 修复时间和修复者对比

再修复原因	再修复时间少	再修复时间多	修复者同一人	修复者非同一人
范围加强	4	0	4	0
强度加强	5	2	4	3
减弱漏洞的填补	2	1	0	3
线性关系	1	0	1	0
并列关系	8	5	11	2
对修复进行登记或测试	7	0	2	5
以前修复无效或错误而重新修复	5	6	7	4
怀疑修复造成了错误，后来发现不是	9	0	2	7

4.5.3 修改模式的对比

针对初始修复和再修复使用的修改模式进行的统计分析，可以通过得出的规律对再修复进行修改模式的推荐。

对漏洞进行登记或测试和怀疑修复造成错误但发现不是这两种再修复基本上不涉及安全性修改，因此再修复无修改模式的使用。

加强漏洞的填补中范围加强基本上再修复使用的修改模式包含初始修复使用了的修改模式，因为再修复只是将初始修复的使用范围变大，但修复方式不变。而强度加强则是既有初始修复的修改模式包含了再修复的修改模式的情况也有被包含的情况，因为再修复是对安全性的加强，但是修复方式不一定一样。

在填补原来没想到的漏洞中，线性关系基本上两次修改模式相同，因为只是作用于不同的地方，但是修复方式完全相同

最后，减弱漏洞的强度、并列关系、以前修复无效或错误而重新修改这三种再修复使用的修改模式与初始修复使用的可能相同也可能不同，因为他们再修复的修复方式与以前的是不同的，自然修改模式也可能是不同的。

针对以上规律，范围加强、线性关系可以进行再修复修改模式的推荐，推荐

初始修复使用过的修改模式，因为这些修改模式一般在再修复工作中都能用到。

4.5.4 文件修改情况对比

安全性漏洞的修复体现在文件、代码的修改上，通过对比初始修复与再修复修改的文件及其内容之间的区别和关系来找出各种再修复修改时的特点。

通过数据（表格 4）的分析，主要有五种文件修改的情况：相同文件，不同内容；相同文件，相似内容；不同文件，不同内容；不同文件，相似内容；无修改。无修改说明再修复没有对相关代码进行任何改动。相同文件，相似内容说明初始修复与再修复修改的文件是相同的，对这些文件内容的修改也是相似的。相同文件，不同内容则是修改文件相同，但是对文件的修改是不同的内容，对修复的作用也不同。同理，剩下两种都修改的是不同的文件，但修改的内容和作用的区别跟上面定义相似。

表格 4 文件修改情况比较

再修复原因	相同文件相似内容	不同文件不同内容	同样文件不同内容	不同文件相似内容	无修改
范围加强	4				
强度加强	3	2	1	1	
减弱漏洞的填补	2	1			
线性关系	1				
并列关系	1	6	5	1	
对修复进行登记或测试					7
以前修复无效或错误而重新修复	7	4			
怀疑修复造成了错误，后来发现不是					9

对修复进行登记或测试、怀疑修复造成错误却发现不是两种再修复不涉及代码修改，范围加强则全是两次修复修改相同文件，相似内容。范围加强前后修复方式相同，且安全性问题比较集中，所以出现这种情况。

强度加强和以前修复无效或错误都集中在前两种情况，说明修改方式不尽相同但也可能相同。而并列关系主要集中在不同的修复内容两种情况，说明修复方式基本上是不同的。线性关系集中在相同文件和相似内容的情况，说明修复方式相似。

4.6 本章小结

本章根据收集到的数据进行统计、分析、总结，4.1 节对漏洞基本信息进行了统计，如再修复发生的频率等，体现了再修复发生的频繁性。同时，4.2 节对再修复的原因进行了分类，针对不同种类的再修复，可以找到其特点，便于再修复预测和自动化再修复。接着，4.3 节对比了不同类型的安全性漏洞发生不同类型再修复的数据统计，得到了不同类型安全性漏洞发生再修复的特点。4.4 节则对安全性漏洞修复的基础信息进行了分析，如再修复次数、修改的代码数等等，得到了不同类型安全性漏洞在修复时的特点。最后，4.5 节将初始修复与再修复从修复过程、修改时间、修改者、修改模式、文件修改情况进行了对比，找到了初始修复与再修复之间存在的联系与规律，为自动化再修复提供知识储备。

第五章 再修复预测

在第四章中，我们对收集到的安全性漏洞再修复数据进行了统计、分析、总结规律，其中一些规律可以运用到对再修复修改的推荐，还可以对再修复的发生进行预测。因此，如果我们在对安全性漏洞进行修复时，通过一些方法预测可能发生的某种类型的再修复，在没有发生的时候就做好安全性防护，减少再修复发生的次数及可能性。但是如果不可避免地发生了再修复，我们可以通过对再修复的涉及文件、修改模式进行推荐来减小再修复的难度。

而根据现在得到的数据和规律，我们设计了两种再修复的预测方法：填补原来没想到的漏洞——线性关系；加强漏洞的修改范围。

5.1 线性关系再修复预测

线性关系这类再修复发生的原因是因为对数据的保护没有找到源头，因此安全性防护的位置不对，针对的比较多的是对某个变量的保护措施，而对变量做保护比较多的就是对变量进行管理，防止变量被垃圾收集器释放的情况。所以，在我们每次修复使用非法内存这类安全性漏洞时，对被保护的那个变量进行数据流分析，找到这个变量进入程序或者被声明赋值的源头，分析在源头是否还有安全隐患。如果有安全隐患就在那里进行保护，这样就可以防止这类再修复的发生。这就是对线性关系再修复发生的预测方法。

根据这个理论流程，我们对这种预测方法进行了验证，看在实际程序中这种预测方法是否可行。首先，使用 java 的 dexlib2 包将源代码转为固定格式的一串执行语句，通过这种执行语句编写程序得出源代码整体的控制流图（如图 10）和数据流图（如图 11）。然后，选定某句安全性问题发生的代码，其中包含某变量，那安全性修复就是对这个变量进行保护。修复完成后根据控制流图找到该代码对应的控制块及执行语句，通过这个在数据流图中找到相应语句，在数据流图中回溯该变量的传递，找到该变量最早出现的地方，然后根据最早出现的执行语句对应控制流图找到其在源代码中的位置，发现这句代码也有同样的安全性问题。但是预测并不是百分之百可以成功的，如果还是发生了再修复，那我们就可以对再修复提供修复意见，帮助程序员更快、更准确地进行再修复。线性关系的再修复一般与初始修复的使用的修改模式相同，且修改的文件相同，内容也相似。

因此,在进行推荐时,可以推荐初始修复的修改模式,应用到找到的数据的源头处,同时通过推荐上次修改的文件,轻易定位到该文件进行再修复。

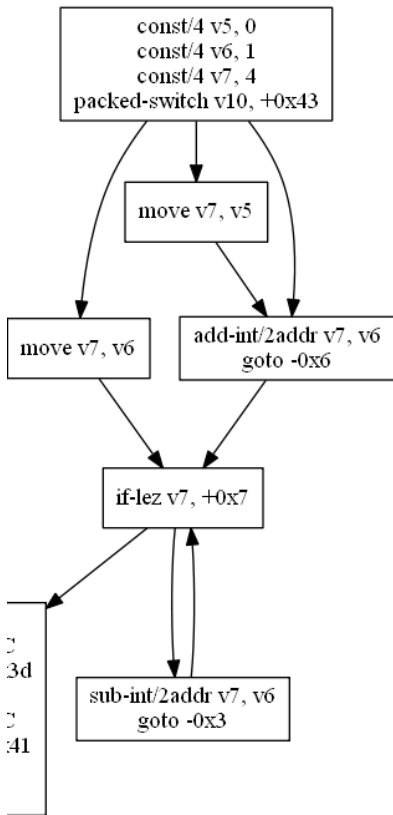


图 10 控制流图 (部分)

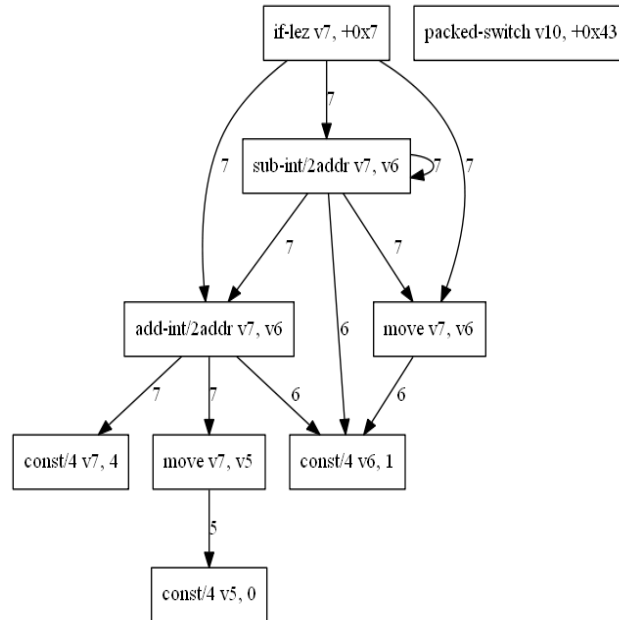


图 11 数据流图 (部分)

使用以上预测理论,我们对安全性漏洞 31227815 的再修复进行了预测 (如图 12 所示,黑色方框内为预测和修复的流程),这个安全性漏洞是因为一个 object 变量被垃圾收集器释放了指向的内存而导致错误。首先我们假设找到了在错误信息中发生问题的那句代码,并通过对该变量进行管理使其不会再被垃圾收集器释放。此时,通过对这个变量进行控制流与数据流分析,找到这个会出错的变量的声明处,查看从声明处到出错处的过程中变量还会不会被垃圾收集器释放,然后发现在这个过程中某处的确会被释放。于是找到那个会被释放的最接近声明的那一处进行管理,这样该变量就不会被垃圾收集器释放了。

对于这些在程序中传递范围较广的变量,一旦发生这种安全性漏洞要判断其错误发生的源头十分困难,因为变量可能从一个类传递到另一个类,传递关系比较复杂,且发生错误的地方与变量源头可能相距较远。而通过控制流域与数据流的分析,我们可以清楚地得到该变量源头所在,快速的对变量源头到变量出错处

¹⁵ https://bugzilla.mozilla.org/show_bug.cgi?id=312278

之间的数据操作进行分析，找到真正需要修复的地方，避免了在复杂的数据传递中漫无目的的搜寻，为修复工作节省了时间，提高了修复的准确度，避免了再修复的发生。

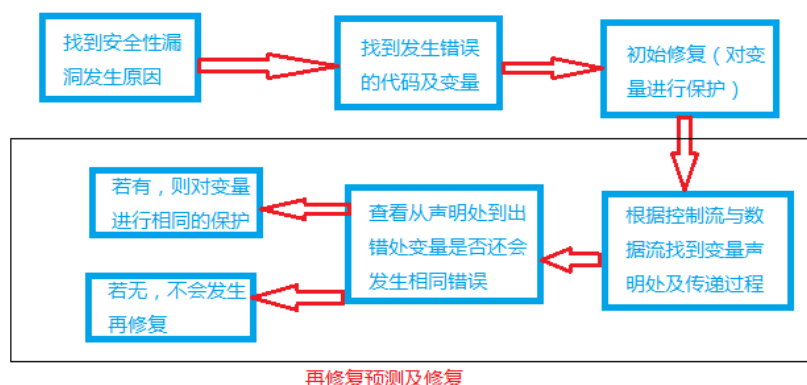


图 12 线性关系再修复预测及修复

5.2 增强漏洞填补范围的预测

克隆代码是指两个代码段之间根据一些给定的相似性定义他们是相似的，而代码之间的相似性主要有两种：一种是文本之间的相似，一种是基于函数性的相似。文本之间的相似一般是将一段代码复制到另一个地方，且只是在布局、空格、注释等地方进行简单修改。基于函数性的相似则是两段代码执行的是相同的运算，实现的是同样的功能，但是语法结构不一定相同。

图 13 展示的是一段函数性相似的代码，f1 与 f2 所做的操作都是打印数组 a 中的所有元素，打印结果均为：1 2 3 4 5 6。f1 与 f2 所用的循环方式不同，导致文字之间相似性低，但是函数性相似度高。而对于增强漏洞填补范围的再修复预测，我们需要进行文本相似性和函数性相似性相结合的检测方式。

```

1 public void f1()
2 {
3     int[] a = {1,2,3,4,5,6};
4     for(int i=0;i<6;i++){
5         System.out.print(a[i]+" ");
6     }
7 }

1 public void f2()
2 {
3     int[] a = {1,2,3,4,5,6};
4     int i = 6;
5     while(i>0){
6         System.out.print(a[6-i]+" "+a[6-i+1]+" ");
7         i -= 2;
8     }
9 }
    
```

图 13 函数性代码相似

增强漏洞填补范围再修复发生的安全性问题与初始修复发生的问题是一样

的，且使用的修复方式也是一样的。对这种再修复最简单的预测方式是进行克隆代码的检测，即找到相似性高的两段代码，一般这种代码有同种安全性问题的可能性很大。

对于这种预测方式我们同样进行了简单的验证，首先我们自己编写了一段有安全性问题的代码，然后在程序另一个地方将这段代码复制过来。我们在修复好原始有问题的代码后使用 CCFinder[27]进行克隆代码的检测，于是找到了那段同样有问题的克隆代码并参照刚刚的修复方式进行了修复。由此这个预测理论的可行性得到初步证明。

与 4.1 一样，我们在再修复发生时也可以提供一些修复意见，因为增强漏洞修复范围再修复的修改模式是包含初始修复用到的修改模式且修改文件、修改方式相同。因此可以将初始修复使用的修改模式、修改的文件进行推荐，提高再修复的效率。

在实际的安全性漏洞 452979¹⁶中，我们对这个理论进行了应用（如图 14，方框内为预测及修改过程），这个安全性漏洞的问题在于 url 中的一些字符必须进行加密，否则有时无法处理或者被窃取利用。第一次修复是将那些字符在进行处理时进行加密。然后，通过 CCFinder[27]克隆代码的检测发现了其它同样在进行处理的字符，其中也含有一些字符应该需要加密的。这些处理字符的代码分布在程序的各个地方，在寻找时十分容易遗漏，因此第一次修复时没能找到全部需要修复的代码，而通过克隆检测能够很全面的找到这些功能相似的代码，于是在这些遗漏的代码中对字符处理时也进行了加密，避免了再修复和损失的发生。

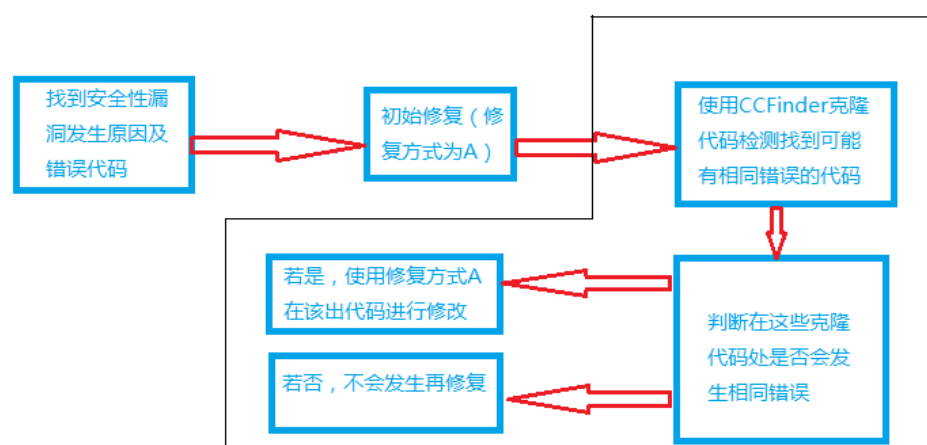


图 14 增强范围再修复预测及修复

¹⁶ https://bugzilla.mozilla.org/show_bug.cgi?id=452979

5.3 本章小结

对安全性漏洞再修复预测的类型目前还只找到两种，只能给开发自动化工具提供一定参考，暂时不具备开发覆盖所有类型安全性漏洞再修复预测和自动化修复工具的能力。这些安全性漏洞再修复的预测及再修复意见推荐都是根据统计数据及规律产生的，而其它种类的安全性漏洞再修复比较复杂或规律还不明显，因此无法提供再修复的预测，而修复意见的推荐可以根据统计出来的规律进行简单的参考，还是有助于再修复的进行和再修复预测与修复工具的开发的。

而且，对发现的两种再修复预测方法进行了简单测试，在实际代码中通过预测方法可以很好的找到再修复可能发生的地方，并且如果发生再修复可以用原始修复的修复方法进行修复，确定了两种预测方法的可行性，且应该具有一定的准确性。不过未经过大量数据、复杂程序的测试，准确性未定量分析，且未开发生成工具，预测效率较低，只能作为开发工具的参考。

第六章 总 结

本文通过对 Mozilla 工程中 2005 年至 2011 年共 48 个发生了再修复的安全性漏洞的经验研究, 分析安全性漏洞类型、发生再修复的原因、再修复的次数、修改的提交数、修改的文件数、修改的代码行数的增减、初始修复和再修复的对比(使用的修改模式、使用的修复过程、修改的时间、修改者、修改文件内容)等数据, 总结了两种再修复预测的方式, 有助于减少再修复的发生, 还提供了一些再修复的规律, 帮助开发人员再修复时找到着重点, 提供参考信息, 也为安全性漏洞再修复预测和自动修复的工具提供基础知识和尝试经验。

本次研究仍然有一些未解决的问题: 安全性漏洞再修复与初始修复之间修复方式的区分还未能很好的识别, 对自动化再修复的研究有一定的限制; 除了已有的两种再修复预测方式, 其他的再修复预测还有待数据分析与总结; 未把预测和自动修复的理论运用到实际工具开发中。

在未来的研究中, 我们可以分析更新、更多的发生了再修复的安全性漏洞的数据, 把规律还不明确的几种再修复着重研究, 尝试找出他们的预测方式, 并通过再修复的历史数据找到前后两次修复修复方式上存在的联系与区别, 来得到自动修复它们的方法, 并开发出再修复预测与自动化修复的工具。

参考文献

- [1] S. Zaman, B. Adams, and A. E. Hassan. Security versus performance bugs: A case study on firefox. In Proceedings of the 8th Working Conference on Mining Software Repositories. MSR ' 11, pages 93 - 102, New York, NY, USA. 2011. ACM.
- [2] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. Proceedings of the 17th Working Conference on Reverse Engineering, WCRE ' 10. pages 249 - 258, Washington, DC, USA. 2010. IEEE Computer Society.
- [3] L. Erlikh. Leveraging legacy system dollars for e-business. IT Professional. 2(3):17 - 23. 2000.
- [4] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. Empirical Software Engineering. 19(6):1665 - 1705. 2014.
- [5] C. B. Haley, R. Laney, J. D. Moffett, and B. Nuseibeh. Security requirements engineering: A framework for representation and analysis. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. 34(1):133 - 153. 2008.
- [6] J. Viega and G. McGraw. Building Secure Software: How to Avoid Security Problems the Right Way. Addison-Wesley, London. 1st edition. 2011.
- [7] 张林, 曾庆凯. 软件安全漏洞的静态检测技术[J]. 计算机工程. 34(12):157-159. 2008.
- [8] 管铭. 基于程序分析的软件安全漏洞检测技术研究[D]. 西北工业大学. 2007.
- [9] J. Thomé, L. K. Shar, and L. C. Briand. Security slicing for auditing xml, xpath, and SQL injection vulnerabilities. In 26th IEEE International Symposium on Software Reliability Engineering. ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015, pages 553 - 564. 2015.
- [10] L. K. Shar, H. B. K. Tan, and L. C. Briand. Mining SQL injection and cross site scripting vulnerabilities using hybrid program

- analysis. In 35th International Conference on Software Engineering. ICSE '13, San Francisco, CA, USA, May 18–26, 2013, pages 642 – 651. 2013.
- [11] M. Gegick, P. Rotella, and T. Xie. Identifying security bug reports via text mining: An industrial case study. In Proceedings of the 7th International Working Conference on Mining Software Repositories. MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2–3, 2010, Proceedings, pages 11 – 20. 2010.
- [12] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. The Workshop on Architectural & System Support for Improving Software Dependability. pp.25—33. ACM 2006.
- [13] 吕维梅, 刘坚. C/C++程序安全漏洞的分类与分析[J]. 计算机工程与应用. 41(5):123–125. 2005.
- [14] 丁羽, 邹维, 韦韬. 软件安全漏洞分类研究综述[C] 信息安全漏洞分析与风险评估大会. 2012.
- [15] Gegick, M., P. Rotella and T. Xie. Identifying security bug reports via text mining: An industrial case study, in IEEE International Working Conference on Mining Software Repositories (MSR 2010). p. 10. 2010: Cape Town, South Africa.
- [16] 马海涛. 计算机软件安全漏洞原理及防范方法[J]. 科协论坛 (6):49–49. 2009.
- [17] P. H. Nguyen, K. Yskout, T. Heyman, J. Klein, R. Scandariato, and Y. L. Traon. Sospa: A system of security design patterns for systematically engineering secure systems. In 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MoDELS 2015, Ottawa, ON, Canada, September 30 – October 2, 2015, pages 246 – 255. 2015.
- [18] K. Yskout, R. Scandariato, and W. Joosen. Do security patterns really help designers? In 37th IEEE/ACM International Conference on Software Engineering. ICSE 2015, Florence, Italy, May 16–24, 2015, Volume 1, pages 292 – 302. 2015.
- [19] M. Felderer, P. Zech, R. Breu, M. Büchler, and A. Pretschner. Model-based security testing: a taxonomy and systematic

- classification. *Softw. Test., Verif. Reliab.*, 26(2):119 – 148. 2016.
- [20] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner. Chapter one – security testing: A survey. *Advances in Computers*. 101:1 – 51. 2016.
- [21] T. Zimmermann, N. Nagappan, P. J. Guo and B. Murphy. Characterizing and predicting which bugs get reopened. *Software Engineering (ICSE), 2012 34th International Conference*. Vol.8543, pp.1074–1083. 2012. IEEE.
- [22] X. Xia, D. Lo, E. Shihab, X. Wang and B. Zhou. Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering*, 22(1), 75–109. 2015.
- [23] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc.. San Francisco, CA, USA. 2005.
- [24] McGraw G. *Software Security: Building Security In[J]*. *IEEE Security & Privacy*, 2006, 2(3):6.
- [25] P. Bhattacharya, L. Ulanova, I. Neamtiu and S. Koduru. An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps. *European Conference on Software Maintenance & Reengineering*. 133–143. 2013.
- [26] Shin, Yonghee, and L. Williams. An Empirical Model to Predict Security Vulnerabilities using Code Complexity Metrics. *International Symposium on Empirical Software Engineering and Measurement, ESEM 2008*. Kaiserslautern, Germany 2008:315–317. October 9–10, 2008.
- [27] Kamiya, Toshihiro, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans Softw Eng. IEEE Transactions on Software Engineering*. 28(7):654–670. 2002.

致 谢

首先感谢实验室导师彭鑫教授给我提供了这个研究的方向与研究的思路，通过时常的讨论逐渐使我找到研究方法，为论文完成起到了指向性的作用。其次，感谢孙小兵老师整个研究中起到的领导性作用，不断将我们的研究方向细化，使我们的研究思路不断清晰，给我们提供了很多相关研究和学习资料，带领着我们通过完成一个个小目标逐渐达成整个研究目标。最后，还要感谢杨辉学长和我一起进行了整个安全性漏洞的研究，通过丰富的研究经验和渊博的知识带领着我攻克了一个又一个的难关，既是我科研的伙伴又是我学习的榜样。