西安交通大學

硕士学位论文

基于并行程序不确定时序交织的恶意代码隐藏方法研究

学位申请人：郝宇

指导教师：刘烃　副教授

学科名称：控制科学与工程

2018 年 5 月

Research on Malicious Code Hiding Methods Based on Uncertain Interleaving of Concurrent Programs

A thesis submitted to
Xi'an Jiaotong University
in partial fulfillment of the requirement
for the degree of
Master of Engineering

By
Yu Hao
Supervisor: Associate Prof. Ting Liu
(Control Science and Engineering)
May 2018

论文题目：基于并行程序不确定时序交织的恶意代码隐藏方法研究
学科专业：控制科学与工程
学位申请人：郝宇
指导教师：刘烃 副教授

摘 要

随着多核技术在处理器中的广泛使用，串行程序已难以发挥现有处理器的运算能力，并行程序成为当前提升效率的必然选择。由于线程调度的不确定性、执行过程难以复现和时序交织数量巨大等问题，并行程序的分析难度远超过串行程序。目前已经有专门针对并行程序的并行攻击出现，然而传统恶意代码检测方法因为没有考虑到并行程序独有的时序交织，所以在面对并行攻击时是无效的。本文的主要目的是为了发现并行程序与串行程序不同的安全漏洞，并且通过对漏洞的使用引起对于并行程序安全的重视。

本文分析了现有恶意代码检测方法在面对并行程序时的一个缺陷，无法检测到隐藏在并行程序中指数增长的时序交织中的恶意代码。并且基于并行程序不确定时序交织提出了一种恶意代码隐藏方法——并行逻辑炸弹。给出了在多线程程序中实现并行逻辑炸弹的完整步骤，设计了将恶意代码分段的恶意代码拆分算法，提出了序列模式理论对时序交织按照触发概率分类，并统计了不同时序交织在不同条件下的触发概率，以此为基础按照不同的触发策略生成了并行逻辑炸弹。将真实的恶意代码使用并行逻辑炸弹隐藏后，实验证明了在面对静态恶意代码检测方法和动态恶意代码检测方法时并行逻辑炸弹均有很好的隐蔽性。此外针对这类并行攻击，本文设想了一些切实可行的防御方法。

本文研究证实了现有的恶意代码检测方法不能检测到隐藏在并行程序时序交织中的恶意代码。传统的恶意代码检测方法在面对新型的并行攻击时是无效的，需要对并行程序的安全进行深入的研究，根据并行程序独有的特点提出完善的恶意代码检测方法。

关 键 词：并行程序；恶意代码检测；时序交织；并行逻辑炸弹
论文类型：理论研究

Title: Research on Malicious Code Hiding Methods Based on Uncertain Interleaving of Concurrent Programs

Discipline: Control Science and Engineering

Applicant: Yu Hao

Supervisor: Associate Prof. Ting Liu

ABSTRACT

With the widespread use of multi-core technology in processors, it has been difficult for serial programs to exert the computing power of existing processors, and concurrent programs have become an inevitable choice for improving efficiency. Due to the uncertainty of thread scheduling, the difficulty of recurrence of the execution process, and the large number of interleaving, it is more difficult to analyze concurrent programs than serial programs. At present, there have been concurrency attacks specifically targeting concurrent programs. However, traditional malicious code detection methods are malfunction when faced with concurrency attacks because they do not take into account the unique interleaving of concurrent programs. The main purpose of this paper is to discover the different security vulnerabilities of concurrent programs and serial programs, and to draw attention to the security of concurrent programs through the use of vulnerabilities.

This thesis analyzes a flaw in the existing malicious code detection methods in the face of concurrent programs, and can not detect the malicious code hidden in the exponentially increasing interleaving in concurrent programs. And based on the concurrent program uncertain interleaving proposed a malicious code hiding method - Concurrency Logic Bomb. The complete steps to implement concurrent logic bombs in a multithreaded program are presented. The malicious code segmentation algorithm is proposed to segments malicious code. The sequential pattern theory is proposed to classify the interleaving according to the trigger probability, and the trigger probability of different interleaving under different conditions is counted. On this basis, concurrent logic bombs are generated according to different triggering strategies. After the real malicious code is hidden using the Concurrency Logic Bomb, the experiment proves that the Concurrency Logic Bomb has good concealment in the face of the static malicious code detection method and the dynamic malicious code detection method. In addition to this type of concurrent attack, this paper proposes some practical defense methods.

This study confirms that existing malicious code detection methods cannot detect malicious code hidden in the interleaving of concurrent program timings. Traditional malicious code detection methods are malfunction when faced with a new type of concurrent attack. It is necessary to conduct in-depth research on the security of concurrent programs and propose a complete malicious code detection method based on the unique features of concurrent pro-

grams.

KEY WORDS: Concurrent program；Malicious code detection；Interleaving；Concurrency logic bomb

TYPE OF THESIS : Theoretical Research

目 录

CONTENTS

1 绪论

1.1 研究背景

随着科学技术的发展，海量计算的需求日益增多。然而支撑海量计算的处理器无法依靠单个处理器核心提高计算速度，需要多个处理器核心共同协作提高计算速度。目前不但处理器的核心数量逐渐增多，计算机的处理器数量也在逐渐增多。比如神威·太湖之光超级计算机拥有多达 40960 块处理器，每个处理器有 260 个核心[1]。并行程序成为了必然的选择，并行程序可以同时利用多个处理器核心，达到相同条件下串行程序无法达到的运算速度。但是，并行程序的编程和测试却是一个非常困难的问题。仅仅是保证并行程序正确的运行就已经非常困难，而保证并行程序正确而又安全的运行则更是难上加难。大部分程序员按照串行程序的思维方式编写并行程序，因此很容易发生各种并行程序的错误，而并行程序的执行不确定性又让并行程序的错误很难被复现。

目前并行程序主要面临如下挑战[2]：并行程序错误检测。目前许多并行程序错误检测的研究主要集中在检测数据竞争错误和死锁错误、原子性冲突错误。数据竞争错误是在没有合适的同步措施时对一个共享变量的两个相互竞争的访问。死锁错误是在两个或者更多操作循环的等待彼此释放需要的资源。原子性冲突是某段代码没有按照期望的原子性执行。

并行程序测试。在串行程序中测试本身就是一个难题，在并行程序中测试中除了串行程序的分支覆盖等问题，还有并行程序独有的交织覆盖等问题[3]。发现一个并行程序错误不仅仅需要一个可以触发错误的输入，还需要一个可以触发错误的交织[4]。而并行程序的交织随着程序规模的增大是指数级增长，覆盖每一个输入下的每一种交织在实际中是完全不可行的。

并行程序语言设计。优秀的并行程序语言可以帮助并行程序正确的完成原本的目标，避免一些特定类型的错误。比如 *transactional memory*[5] 可以提供给并行程序一种更合适的方法来指定需要原子性执行的代码。

就像串行程序的错误可能导致一系列安全问题，并行程序的错误同样能导致安全问题[6]。而且这些问题主要针对并行程序错误独有的特性，传统的安全防范措施并不能很好地发挥作用。并行安全问题并不是杞人忧天，事实上在 *CVE database*[7] 已经有很多并行安全漏洞。

图1-1是 *Linux* 内核中关于内存指针的一个并行错误的例子。这个漏洞严重到可以使一个本地的普通用户得到根权限或者在 *ring 0* 级别执行任意代码[8,9]。这个漏洞的根本原因是原子性冲突，忽视了在并行程序中其他线程可以在代码执行间隔中修改未被保护的数据。在 *Linux* 中加载一个 *ELF* 格式的共享库文件，一个进程会调用函数

```
1  load_elf_library(...){
2      down_write(&current->mm->mmap_sem);
3      error = do_map(...);
4      up_write(&current->mm->mmap_sem);
5      ...
6      if(bss > len)
7          do_brk(...);
8  }
9  do_brk(...){
10     struct mm_struct * mm = current->mm;
11     ...
12     vma = kmem_cache_alloc(...);
13     ...
14     vma_link(mm, vma, ...);
15 }
```

图 1-1　*Linux* 内核内存映射漏洞

uselib()，然后会调用图1-1中的函数 *load_elf_library()*。在第 2-4 行中这个函数正确的对内存空间进行了操作，因为其通过信号量 *mmap_sem* 保证了这段代码的原子性。但是在第 7 行调用函数 *do_brk()*，这个函数在对内存操作时没有使用信号量，从而导致其他线程可以在第 10-14 行执行这段时间内修改内存空间，最终导致这个严重的漏洞。

```
1  __nptl_setxid(struct xid_command *cmdp)
2  {
3      lll lock(stack cache lock);
4      ...
5      list_for_each(runp,&stack used)
6      {
7          struct pthread *t =
8              list_entry(runp,struct pthread,list);
9          if (t == self)
10             continue;
11         setxid_signal_thread(cmdp,t);
12     }
13     lll_unlock(stack_cache_lock);
14 }
15 allocate_stack(...) {
16     lll_lock(stack_cache_lock);
17     list_add(&pd->list,&stack_used);
18     lll_unlock(stack_cache_lock);
19 }
```

图 1-2　*Glibc setuid* 竞争漏洞

图1-2是 *Glibc* 中身份识别的一个并行错误的例子。这个漏洞可能导致错误的身份识别进而允许提升权限攻击[10]。这个漏洞的根本原因是由数据竞争导致，没有合适的同步措施保证这些操作按照正确的顺序执行。在 *Linux* 中，每个内核线程都有自己的身份信息。当一个线程调用函数 *setuid()* 时，*nptl* 需要确保当前进程内所有的线程调用函数 *setuid()*。图1-2中的函数 *__nptl_setxid()* 所示，在第 6-12 行逐个迭代每一个线程并且唤醒线程调用 *setuid()*。但是唤醒线程却没有等待线程完成全部操作，直接释放了锁 *stack_cache_lock*。在这个间隔可能创建新的线程，调用图1-2中的函数 *allocate_stack*，就会导致新的线程可能拥有旧的线程的身份，从而提升权限。

```
1  bool FastCopy(MonoArray *src, MonoArray* dest, int length){
2      // Checks that the type of dst[i] derive from src[i]
3      for (i = 0; i < length; ++i)
4          if(!safe_cast(type_of(src[i]), type_of(dest[i])))
5              return FALSE;
6      ...
7      for (i = 0; i < length; ++i)
8          memcpy(dest[i], src[i], size of(ObjPtr));
9      return TRUE;
10 }
```

图 1-3 *Moonlight* 快速复制竞争漏洞

图1-3是 *Moonlight* 中快速复制的一个并行错误的例子。*Moonlight* 是一个浏览器组件，这个漏洞允许攻击者违反 *Moonlight* 中的类型安全[11]。这个漏洞的根本原因是没有合适的同步措施保证这些操作可以满足原子性的执行。为了加速复制速度，图1-3中函数 *FastCopy()* 首先在第 3-5 行检查源数据和目标数据类型是否相容，然后在第 7-8 行调用函数 *mencpy()* 进行快速复制。但是这些操作并不是原子操作，在经过类型检查后第 6 行的间隔中攻击者可以修改目标数据类型，比如将私有属性修改为公共属性，进而进行窃取数据等行为。

通过这几个例子可以看出，从串行程序的角度考虑，上述例子中的代码并没有漏洞。但是在并行程序中，原本安全的代码就会有漏洞，造成各种各样的危害。而且由于触发的条件独特，一般在串行程序有效的检测方法无法检测出来这些只在并行程序中发生的漏洞。

1.2 研究现状

并行程序的分析在软件工程领域越来越重视，特别是 *Shan lu* 等人在 2008 年对实际并行软件中存在的漏洞进行分类和统计，为后续并行软件分析的工作提出了目前真实存在的最迫切的问题[2]。这些工作主要解决程序员在编写并行程序时可能导致的各种问题，比如数据竞争[12–14]、死锁[13,15]、原子性冲突[16–18]、模型检验[3,19] 等等。

然而在安全领域，尽管 2012 年 *Junfeng Yang* 等人提出并行攻击这一概念试图引起人们对并行程序安全的足够重视[6]，但是从安全角度对并行程序研究的工作依旧很少，无论是从攻击者的角度或者是从防御者的角度。

在 2010 年由 *Ramilli, Marco* 等人提出了多阶段攻击[20]。多阶段攻击将完整的恶意代码拆分成多个部分，由原本一次完成的恶意行为分为若干个阶段完成。一段完整的恶意代码会被静态检测方法检测到恶意特征，同样，一次性的完成恶意行为会被动态检测方法直接将所有恶意行为检测出来。该方法将恶意代码拆分成多个部分，按照顺序插入到程序中的不同地方。多阶段攻击可以有效的抵抗一些静态检测方法和动态检测方法，因为拆分后恶意代码的特征也分散到不同部分，单独对一个部分进行检测无法得到足够有效的恶意特征进行判断，前后多个部分有一定的时间跨度，而一般的检

测方法无法跨越时间跨度将特征整合到一起进行检测。

多阶段攻击本身并不是并行攻击，但是在 2011 年由 *Ramilli, Marco* 等人基于多阶段攻击提出了多进程攻击[21]，多进程攻击就是一种并行攻击。多进程攻击在多阶段攻击的基础上，将拆分的恶意代码按照顺序插入到不同的进程中，不仅仅在时间上有一定跨度，在空间上也有一定跨度。而传统的恶意代码检测方法只考虑单个进程，多进程攻击中单个进程中没有足够的恶意特征进行判断，所以面对各种恶意代码检测方法有很好的隐蔽效果。

然而多阶段攻击甚至多进程攻击检测方法显而易见，只需要将分散在不同进程的恶意代码的不同的部分识别出来按照整体进行检测，就可以像检测串行程序那样进行检测。

按照这个思路，2015 年 *Lejun Fan* 等人[22] 分析了恶意隐私窃取软件中的多进程协作并进行检测。

2017 年 *Seyyed Mojtaba Bidoki* 等人[23] 提出了针对一般多进程恶意软件的检测方法。主要通过监视系统上运行的所有进程，并通过查找按照通用执行策略运行的进程来发现协作进程，这些通用执行策略是事先已经通过使用强化算法学习的不同的执行策略。最后，通过分析已识别的协作进程的累积行为来决定是否为多进程恶意行为。

2017 年 *Gheorghe Hăjmăşan* 等人[24] 认为应当从以往将恶意软件视为单个组件的想法转变为更为精确的将恶意软件视为多个组件的想法。提出了一种动态行为检测解决方案，用于识别相关进程组，并使用行为进行启发式的分析这些进程组所执行的操作，评估其行为，从而可以检测到多进程恶意软件。

上述攻击方法或者防御方法尽管利用了并行程序的特点，但是远远没有达到 *Junfeng Yang* 等人提出的并行攻击的上限。仅仅是利用了并行程序多进程这一特点进行攻击，所以防御措施也非常直观，找出所有相关的进程统一分析，毕竟操作系统中进程的数目是有限的。然而，并行程序中各种复杂的特性比如时序不确定行、路径交织数量爆炸等等并没有得到其应有的发掘与重视。尽管利用并行程序这些特性非常困难，但是从理论上分析是可行的，并且已经有了许多实际发生的漏洞来证明其可行性，而且这些漏洞都是非常难以发现却能造成巨大的危害。所以，并行程序的安全问题应当得到更多的重视。

1.3 研究内容

尽管并行攻击可能发生的场景越来越多，可能造成的危害越来越严重，但是目前对于并行程序的安全问题还没有足够的认识与重视。特别是传统的恶意代码检测方法目前并不能适应并行程序的安全需求。一方面因为并行程序本身就存在着固有的复杂性，相对于串行程序更加难以分析。另一方面因为现有的恶意代码检测方法考虑的对

象均为串行程序，而串行程序在本质上和并行程序并不相同。

为此，本文找出了一个传统恶意代码检测方法在面对并行程序时的缺陷，基于此提出了一种隐藏恶意代码的方法——并行逻辑炸弹，并且通过实验证实了其危害的有效性。

希望可以引起对于并行程序安全问题的重视，提前防范并行逻辑炸弹，以及将来可能出现的更多更加复杂的并行攻击。

本工作依托于国家自然科学基金重点项目 (61632015)“基于符号执行的复杂软件系统分析与验证”，以及国家自然科学基金面上项目 (61472318)“多线程程序约束构建、优化求解及其智能测试方法研究”。

本文主要工作包括如下几个部分：

找出现有恶意代码检测方法在面对并行程序时的一种缺陷。本文总结了在软件工程领域中分析并行程序面临的难题，同时总结了目前的恶意代码检测方法。并且在二者之间找到了恶意代码检测在面对并行程序时的一个缺陷，现有的恶意代码检测方法没有对并行程序中指数级增长的路径交织进行检测，为并行逻辑炸弹的提出提供了理论基础。

提出并实现并行逻辑炸弹。根据恶意代码检测方法在面对并行程序时的缺陷，提出了一种基于并行程序时序交织的恶意代码隐藏方法——并行逻辑炸弹。并行逻辑炸弹是将恶意代码拆分为多个代码片段，插入到并行软件的不同位置。并行软件只有按照特定的线程交织执行，拆分好的代码段才会按照正确的顺序执行，实现了将恶意代码隐藏在几乎不可数的线程交织中。给出了并行逻辑炸弹完整的实现过程，包括恶意代码的拆分、插入点的选取、线程交织的概率、序列触发策略等。恶意代码的拆分主要将一段完整的恶意代码拆分为多段单独无害的代码片段。插入点的选取分析了并行程序中不同的函数类型，挑选出了合适的插入点。线程交织的概率分析了并行程序可能产生的线程交织，以及其发生的概率表现。序列触发策略给出了并行逻辑炸弹选取线程交织的策略。通过这些步骤，可以构造出指定负载下发生概率高、非指定负载下发生概率低的并行逻辑炸弹。

验证并行逻辑炸弹的有效性。为了验证并行逻辑炸弹的有效性，本文使用并行逻辑炸弹隐藏了真实的恶意代码，分别通过静态检测方法和动态检测方法，验证了其隐蔽性。其中动态检测方法几乎包括了目前互联网上全部可用的动态检测方法。同时本文设计了一种远程触发方案验证了并行逻辑炸弹的危害性。

分析并行逻辑炸弹的缺陷以及防范措施。尽管并行逻辑炸弹有着很强的隐蔽性，但是在其他方面还是存在着缺点。目前虽然没有方法可以完美的检测并行逻辑炸弹，但是可以通过在某些方面的妥协在一定程度上防范并行逻辑炸弹。本文分析了目前并行逻辑炸弹存在的缺陷并且给出了实际可行的防范手段。

1.4 组织结构

本文共六章，全文组织及章节内容概述如下：

第1章，绪论。本章简要的介绍了分析并行程序的难点和并行攻击，并且介绍了现有的并行攻击和检测的工作，说明了研究并行程序安全问题的重要性和必要性。整体的说明了本文的研究内容和研究目的，概括了论文的整体结构。

第2章，并行程序与恶意代码检测。本章总结了并行程序中现有的缺陷，以 *Windows* 中线程的实现举例说明。以及介绍了作者之前分析并行程序的工作和实际中面临的问题。总结了目前的恶意代码检测方法，包括静态检测方法和动态检测方法。分析了现有恶意代码检测方法在面对并行程序时的缺陷。

第3章，并行逻辑炸弹。本章提出了一种基于并行程序时序交织的恶意代码隐藏方法——并行逻辑炸弹，并且完整介绍了实现过程。主要为恶意代码的拆分、插入点的选取、线程交织的概率、序列触发策略等。

第4章，实验。本章分别对并行逻辑炸弹进行了静态检测，动态检测，远程触发。

第5章，讨论和防御。本章讨论了并行逻辑炸弹的缺陷，基于这些缺陷给出了实际可行的防御方法。

第6章，总结与展望。本章总结了本文的贡献和不足，并对未来的研究计划简要说明。

2　并行程序与恶意代码检测

2.1　并行程序

串行程序是完全确定的，程序会完全按照编写的代码执行，并行程序则是充满不确定性[25]。对于一个串行程序如图2-1所示，其中有三个分支，分别为第 2 行、第 7 行和第 9 行的 *if* 语句，*input_1* 和 *input_2* 为输入。这个程序中因为第 7 行的 *if* 语句导致永远 $y \geq 0$，从而第 9 行的 *if* 语句在任何输入下不可能执行 *true* 分支。对于任意的 *input_1* 和 *input_2*，一条路径会覆盖三个分支的一边。例如当 *input_1=0* 和 *input_2=0* 时，将会执行的路径行号为 $1-2-4-5-6-7-9-11-12$，覆盖了第 2 行 *if* 的 *false* 分支、第 7 行 *if* 的 *false* 分支和第 9 行 *if* 的 *false* 分支。通过改变不同的输入，例如得到当 *input_1=-1* 和 *input_2=-1* 时，将会执行的路径行号为 $1-2-3-6-7-8-9-11-12$。这两种输入覆盖了所有可覆盖的分支，但是很明显，从路径覆盖的角度上还有输入组合为 *input_1=-1*、*input_2=0* 和 *input_1=0*、*input_2=-1* 的情况。在这两种输入下，分支的组合分别为第 2 行 *if* 的 *true* 分支、第 7 行 *if* 的 *false* 分支、第 9 行 *if* 的 *false* 分支和第 2 行 *if* 的 *false* 分支、第 7 行 *if* 的 *true* 分支、第 9 行 *if* 的 *false* 分支。这四种输入就完全覆盖了图2-1所示的串行程序的全部不同路径。换句话说，只要覆盖了所有不同的输入，那么就覆盖了串行程序所有的路径。

代码 2.1　Thread T1

```
1  x = input_1;
2  if(x < 0)
3      y = y + 1;
4  else
5      y = y - 1;
6  y = input_2;
7  if(y < 0)
8      y = - y;
9  if(y < 0)
10     x = x + 1;
11 else
12     x = x - 1;
```

图 2-1　串行程序示例

当图2-1所示的串行程序改写为图2-2中所示的并行程序时，路径覆盖发生了根本性的变化。代码2.2为线程 *T1*，代码2.3为线程 *T2*。当 *input_1=0* 和 *input_2=0* 时，可能将会执行的路径行号为 $1-2-4-5-6-7-9-11-12$，覆盖了第 2 行 *if* 的 *false* 分支、第 7 行 *if* 的 *false* 分支和第 9 行 *if* 的 *false* 分支。然而还可能会执行的路径行号为 $6-7-1-2-4-5-9-10$，覆盖了第 2 行 *if* 的 *false* 分支、第 7 行 *if* 的 *false* 分支和第 9 行 *if* 的 *true* 分支。事实上还可能会执行的路径行号为 $1-6-7-9-11-12-2-3$，覆盖了第 2 行 *if* 的 *true* 分支、第 7 行 *if* 的 *false* 分支和第 9 行 *if* 的 *false* 分支。可以看出在并行程序中，即便在相同输入下，不同的时序也可能导致产生新的路径。可能某

些时序产生的路径通过改变输入可以得到相同的路径，然而某些时序产生的路径则是无法通过改变输入得到。比如行号为 $6-7-1-2-4-5-9-10$ 的路径，执行了第 9 行 *if* 的 *true* 分支。

代码 2.2　Thread T1

```
1  x = input_1;
2  if(x < 0)
3      y = y + 1;
4  else
5      y = y - 1;
```

代码 2.3　Thread T2

```
6  y = input_2;
7  if(y < 0)
8      y = - y;
9  if(y < 0)
10     x = x + 1;
11 else
12     x = x - 1;
```

图 2-2　并行程序示例

一个有 n 个线程、每个线程有 m 个指令的并行程序，其可能的时序交织多达 $(nm)!/(m!)^n$ 种。想要覆盖所有不同路径，一个直观的方法就是覆盖所有的时序交织，然而这是不可能的。

在 *Windows* 操作系统中中，线程调度采用优先级加随机调度[26]。如果某个线程优先级较高，则优先执行优先级高的线程。优先级相同的线程随机分配 *CPU* 时间片。每次分配给处于就绪态的线程一个 *CPU* 时间片，时间大概为 *20ms*，用完后重新分配，当线程由就绪态转变为阻塞态时则自动交出剩余的 *CPU* 时间片，当重新得到资源变为就绪态时会重新请求 *CPU* 时间片

如果程序每次开始执行的环境相同，因为 *CPU* 时间片的长度相同，单位时间执行的指令数量相同，那么线程切换的位置也将会相似。但是在真实的世界中，环境复杂多变，程序每次执行的环境不可能完全相同，线程切换的位置也将会发生变化，所以真实世界中可能发生的路径不可预计。

为了探索并行程序所有不同的路径，我们曾今使用具体-符号执行和约束求解的方法探索新的路径[27,28]。通过约束求解的确可以有效地探索新的路径，甚至在理论上达到覆盖全部路径。

然而由于并行程序本身的复杂性，即便通过约束求解也只能对规模较小的程序进行求解，并且计算时间随着程序规模指数级上升。

也许在量子计算成熟后可以通过超级强大的计算能力解决并行程序路径爆炸的问题，但目前来说没有可以在实际中覆盖所有路径的方法。

2.2　恶意代码检测

现有的恶意代码检测方法尽管种类繁多，采取的手段也各种各样。从本质上讲都是根据已经发生的或者可能发生的行为进行判断是否为恶意代码。

在 1995 年 *Raymond W. Lo* 等人提出了判断程序是否为恶意的一些特征[29]。不管是通过何种方法，如果检测到程序包括这些特征，那么程序就很有可能是恶意的。这

里主要讨论检测到这些特征的方法，主要分为静态检测和动态检测。

2.2.1　静态检测方法

静态检测不执行目标程序，通过分析源代码或者编译好的二进制文件得到程序可能发生的行为。如果可以得到目标程序的源代码，那么静态分析将会有较好的效果。例如在 1995 年 *Raymond W. Lo* 等人提出了 *MFC* 进行恶意代码的判定，通过检测恶意程序的源代码是否包括一些恶意特征进而判断是否为恶意程序[29]。如果只有目标程序的二进制文件，由于编译过程中会损失一些信息，静态分析将会变得复杂，但是仍会有不错的效果[30]。例如在 2005 年 *Mihai Christodorescu* 等人提出了基于语义的恶意代码检测方法[31]。通常情况下目标程序的源代码并不可获得，分析二进制文件将会面对各种各样的混淆方法，这些都使得静态分析变得无力。有些恶意程序甚至加密恶意程序，在动态执行时解密程序然后执行，面对这样的恶意程序时静态检测可以说是无效的[32]。由于动态检测没有以上限制，目前恶意代码检测方法中，动态检测是比静态检测更加有效的方法。

2.2.2　动态检测方法

动态检测方法需要执行目标程序，通过分析目标程序真正执行过程中的指令进行判断目标程序是否为恶意程序。最早动态检测主要用来辅助静态检测，因为静态检测无法得到目标程序真正的指令，而动态执行可以相对容易的得到目标程序执行的指令。例如在 2006 年 *Paul Royal* 等人提出的通过静态分析和动态分析结合抽取隐藏在目标程序中的指令然后进行分析[33]。在 2007 年 *Kang, Min Gyung* 等人提出的通过动态执行记录下真正执行的指令，抽取隐藏在目标程序中的指令，然后对进行分析[34]。

更为强大的动态检测方法是沙箱、虚拟环境、符号执行，强制执行等等。

CWSandbox 是 2007 年 *Carsten Willems* 等人提出的沙箱[35]。沙箱完整的模拟了一个操作系统环境，目标程序在沙箱中执行就如同在真实的操作系统中执行。可以完全监控目标程序调用的的系统函数、网络连接、生成的文件信息等等。通过分析目标程序这些行为进行判断是否为恶意程序。然而，沙箱毕竟是模拟操作系统而不是真正的操作系统，通过一些细微差别恶意程序可以检测到自己处于沙箱环境中[36]。这时恶意代码不执行恶意的路径，执行正常的路径，使得沙箱检测不到恶意行为。

Ether 是 *Artem Dinaburg* 等人在 2008 年提出的一种硬件仿真的虚拟环境[37]。从硬件而不是操作系统进行仿真，大大提高了被检测到的难度。但是恶意程序还是通过执行指令时间的细微差别检测出了虚拟环境与真实环境的不同，隐藏了自己的恶意行为[36]。恶意代码总有办法判断出所处的环境并且改变自己的行为，一个非常有效的应对方法就是符号执行。

KLEE 是 *Cristian Cadar* 等人在 2008 年提出的符号执行工具[38]。符号执行中进行逻辑判断的条件为符号表达式而不是真实的值。执行时通过求解器进行求解，只要该

分支存在满足的解就会执行该分支。通过符号执行可以将目标程序的全部路径的全部行为检测出来。

X-Force 是 *Kevin Fu* 等人在 2014 年提出的强制执行工具[39]。强制执行更加暴力，无条件的执行每一个分支，甚至真实情况下不可能执行的分支或者是该状态下执行会发生错误的分支。通过这些方法大大加强了恶意代码检测的能力，然而不同的分支之间可以组合成不同的路径，在不同路径下也可能隐藏着恶意代码。尽管执行了全部分支，但是由于没有执行到完整包含恶意代码的路径，仍然可能检测不到恶意代码。

早在 2007 年 *Andreas Moser* 等人就开始探索程序中不同的路径进行恶意代码的检测[40]。后来 *Fuzzing* 是为更加有效的探索不同输入导致不同路径的工具[41–43]，通过各种策略生成输入尽可能的覆盖更加全面的路径。

2.3 分析

在可以获得目标程序源代码进行检测时，静态检测相对于动态检测的好处也是显而易见的，不像动态检测只能覆盖执行的代码，静态检测可以覆盖所有的代码[6]。除此之外，静态分析相对于动态分析可以速度更快，因为动态分析需要执行代码，模拟环境，探索各种路径[44]。动态分析相对于静态分析拥有着更好的误报率和漏报率，却只能分析规模更小的程序。[45] 但是在没有目标程序源代码时面对各种各样的混淆和加密时，静态分析可以说处于完全失效的状态，动态分析成为了唯一的选择。

根据第2.2.2节的介绍, 动态分析目前可以分析目标程序在不同输入下的不同路径中的所有行为。但是在并行程序中，根据第2.1节的介绍，即便是在完全相同的输入下，路径也不一定完全相同。那么，在并行程序中，即便按照串行程序的分析方法分析了在所有输入下程序的表现，仍旧不能判定目标程序是安全的。需要对并行程序在每一个输入下由时序交织产生的所有不同路径进行分析才可以确保目标程序是安全的。目前并不存在这样的安全工作，这是现有安全工作的一个缺失。

安全问题本身就是一个不断对抗的问题，为了引起大家对于并行攻击的重视，提出更加高效实用的防御方法，本文提出了一种基于并行程序时序交织的恶意代码隐藏方法。希望本文提出的这种恶意代码隐藏方法可以提前进行针对性的防御，并且引出更多有关并行程序安全问题的研究。

2.4 本章总结

本章介绍了并行程序在分析时面临的最困难的问题——指数级增长的由时序交织产生的路径。同时介绍了主要的恶意代码检测的方法，静态检测已经无法有效地针对新型的恶意代码，动态检测通过探索目标程序的不同输入得到新的路径进行检测。面对并行程序时，现有的动态检测方法也将因为无法检测由时序导致的新路径而失效。

3　并行逻辑炸弹

基于第2章关于目前并行程序设计和恶意代码检测方法之间的缺陷，本文提出了一种新的恶意代码隐藏方法——并行逻辑炸弹。并行逻辑炸弹将恶意代码拆分为多个片段，插入到并行程序的不同位置。只有按照特定的线程交织执行，拆分好的代码段才会按照正确的顺序执行。实现了将恶意代码隐藏在并行程序中数量爆炸的线程交织中。

逻辑炸弹[46] 指的是一段代码注入到一个正常的软件系统中，只有当某些特定的条件满足的时候恶意行为才会触发。传统的逻辑炸弹需要满足的特定条件大多为特定的时间，特定的输入数据。并行逻辑炸弹扩张了逻辑炸弹的范围，并行逻辑炸弹需要满足的特定条件为特定的时序交织。只有满足特定的时序交织，恶意代码才会按照正确的顺序执行。否则恶意代码可能按照错误的顺序执行，或者不执行。

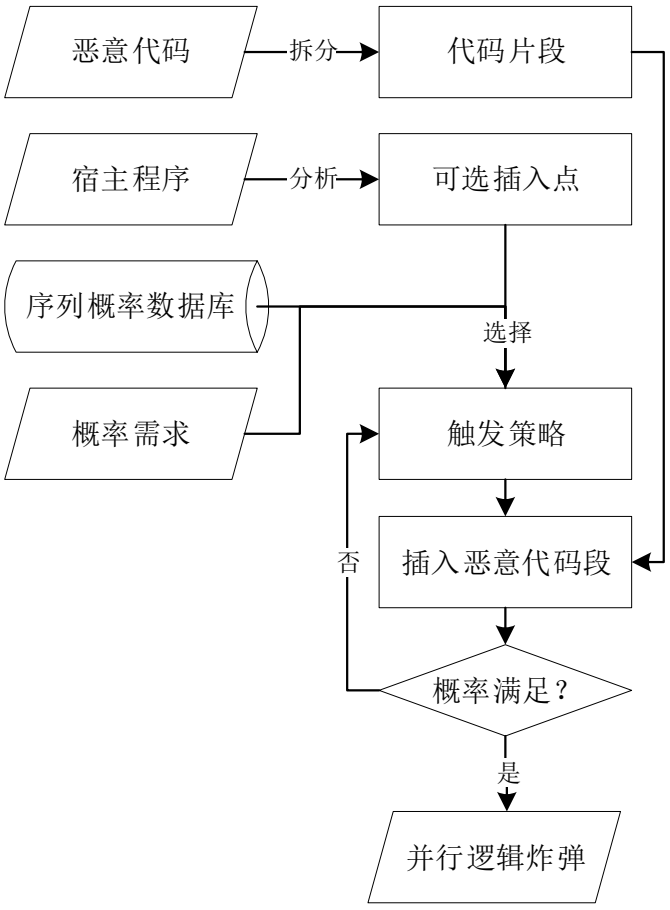


图 3-1　并行逻辑炸弹的工作流程

并行逻辑炸弹的工作流程如图3-1。输入有三个，分别为恶意代码，宿主程序，概率需求。恶意代码是需要隐藏的代码，要求和宿主程序可以在同类平台运行。理论上恶意代码可以是源代码，可以是编译完成的二进制文件。宿主程序是将恶意代码隐藏的宿主代码，要求宿主程序为并行程序。同样，理论上宿主程序可以使源代码，可以是编

译完成的二进制文件。概率需求是恶意代码触发的期望概率。理论上可以是%0∽%100之间任意值。准备数据为交织概率数据库，包括各种条件下各种线程交织的触发概率。各种条件包括各种 CPU 型号，各种操作系统，各种并行环境，各个并行数目等等指标。主要步骤有三个，分别为拆分恶意代码得到代码片段，分析宿主程序得到可选插入点，选择满足概率需求的线程交织。恶意代码为一段完整的代码，通过恶意代码拆分将其拆分为多个代码片段。多个代码片段只要按照顺序执行就等同于原本完整的恶意代码。宿主程序为并行程序，通过对并行程序分析，得到其函数类型和数目，进而确定其中可以作为插入点的位置。这些可选插入点为概率需求提供选择。根据给定的概率需求和可选插入点从交织概率数据库中得到满足要求的线程交织。可选拆入点决定了交织概率数据库中可以选择的线程交织类型和数目。概率需求和可选择的线程交织类型和数目以及它们的触发概率决定了线程交织的选择策略。辅助步骤有两个，插入恶意代码，验证概率。根据实际情况将恶意代码片段插入选择好的插入点，然后验证插入的恶意代码片段触发概率是否满足需求。若是不满足概率需求则重新选择新的线程交织，然后重新验证是否满足概率需求，直到满足概率需求。若是满足需求则生成并行逻辑炸弹。

因为目前恶意软件主要为 *Microsoft Windows* 操作系统设计，占据所有恶意软件 90% 以上[45]。所以本文以 *Microsoft Windows* 操作系统作为分析和实验的环境。并行环境包括多进程和多线程等等，为了便于分析和实验，本文只讨论线程，但原理上都相同。虽然理论上恶意代码和宿主程序可以为源代码，可以为编译好的二进制文件。为了更加直观的解释，以及有效的实验，本文采用的恶意代码和宿主程序均为源代码。

本章组织及内容如下：第3.1节恶意代码拆分，主要包括了如何将恶意代码拆分为多段代码片段。第3.2节插入点选取，主要分析了并行程序得到可以作为候选的插入点。第3.3节线程交织概率，分析了线程调度的主要影响因素以及可能产生的线程交织和各种序列，通过统计学方法得到序列概率数据库。第3.4节序列选择策略，根据给定的概率需求，以及可用的插入点，结合序列概率数据库选择合适的触发策略。第3.5节本章总结，总结本章内容。

3.1 恶意代码拆分

恶意代码拆分是将恶意代码拆分为多个片段，之前已有相关类似工作[20,21]。与普通的代码拆分不同，恶意代码拆分需要满足单独执行拆分后的代码片段不会被动态检测为恶意代码。也就是说需要将原本一段代码完成的恶意行为拆分到多个代码片段中协同完成。除此之外，本文中恶意代码拆分后的代码片段不仅要满足正确顺序可以正常执行，还要满足其他特殊要求。在错误的顺序遇到代码片段时，代码片段执行不会出错，只会将其中正确的部分执行。在重复遇到相同代码片段时，代码片段只会第一

次执行一次，后续重复的部分不会执行。甚至在需要的情况下，只有遇到同一代码片段若干次后，执行代码片段一次，后续重复的部分不会执行。同时代码片段要保证在复杂的并行执行情况下不出错。

2014 年 *Fei Peng* 等人提出了强制执行工具 *X-Force*[39]。*X-Force* 可以执行分支的任意一边，所以可以执行条件不符合的分支。因此可以执行全部拆分好的代码片段，但是不能保证正确的执行顺序。2015 年 *David A. Ramos* 等人提出了部分执行代码工具[47]。同样可以执行全部拆分好的代码片段，但是不能保证正确的执行顺序。

1）恶意代码拆分算法

输入: 恶意代码 $MaliciousCode$，期望代码片段数目 n
输出: n 个恶意代码片段 $CodeFragments$

function GETFRAGMENT($MaliciousCode, n$)
    $ActionNumber, Action \leftarrow$GETACTION($MaliciousCode$)
    for $i = 0 \rightarrow n$ **do**
        $CodeFragments[i] \leftarrow \emptyset$
    end for
    for $i = 0 \rightarrow ActionNumber$ **do**
        $HarmlessCodeNumber[i], HarmlessCode[i], CorrelatedVariable[i]$
                $\leftarrow$ GETACTION($Action[i]$)
        GLOBALIZED($CorrelatedVariable[i]$)
        if $HarmlessCodeNumber[i] > n$ **then**
            return Failure
        end if
        for $j = 0 \rightarrow HarmlessCodeNumber[i]$ **do**
            $j = j + 1$
            $CodeFragments[j] \leftarrow CodeFragments[j] \cup HarmlessCode[i][j]$
        end for
    end for
    for $i = 0 \rightarrow n$ **do**
        $CodeFragments[i] \leftarrow CodeFragments[i] \cup RepeatFramework$
        $CodeFragments[i] \leftarrow CodeFragments[i] \cup SequenceFramework$
        $CodeFragments[i] \leftarrow CodeFragments[i] \cup ConcurrencyFramework$
    end for
    return CodeFragments
end function

图 3-2　恶意代码拆分算法

基于以上要求，本文提出的代码拆分算法如图3-2所示。输入为恶意代码 $MaliciousCode$ 和期望代码片段数目 n。输出为 n 个恶意代码片段 $CodeFragments$。首先根据恶意代码 $MaliciousCode$ 调用图3-4得到恶意行为 $Action$ 和恶意行为个数 $ActionNumber$。将恶意代码片段 $CodeFragments$ 初值设为空集。由于恶意代码片段 $CodeFragments$ 的数目已经设定为 n，所以如果恶意代码 $MaliciousCode$ 无法拆分至 n 个，使用空集将其补齐。对于每个恶意行为 $Action$，调用图3-5得到 $HarmlessCodeNumber$ 个无害代码片段 $HarmlessCode$, 以及不同片段共同使用的相关变量 $CorrelatedVariable$。保证每个单独的代码片段无害可以避免只执行了单个片段就被检测为恶意代码。全局化相关变量 $CorrelatedVariable$，因为不同片段都要共同使用，所以需要将其作用域从原本的函数内扩大到全局。这样可以使得原本整体的代码可以分散到并行环境执行。如果拆分的无害代码片段个数 $HarmlessCodeNumber > n$ 的

话，那么无法将其拆分为 n 个恶意代码片段，返回拆分失败。因为已经设定好最多拆分为 n 个，但是无法保证拆分为 n 个不会被检测到的片段。对于 $HarmlessCodeNumber$ 个无害代码片段 $HarmlessCode$，将其一一分配到 $CodeFragments$ 中。因为每个无害代码片段 $HarmlessCode$ 属于不同的恶意行为 $Action$，所以放到同一个代码片段中相互之间没有任何直接关联，故单个 $CodeFragments$ 不会被检测为恶意代码。对于每个代码片段 $CodeFragments$，为其添加重复执行框架 $RepeatFramework$，顺序执行框架 $SequenceFramework$，并行执行框架 $ConcurrencyFramework$。

2）代码片段示例

```
输入: 执行顺序 N 和重复次数 M
 1: function CodeFragmentsN( )
 2:     Lock( )
 3:     if (N − Sequence) == 0 then
 4:         Repeat + +
 5:         if Repeat == M then
 6:             Sequence = N + 1
 7:             HarmlessCode
 8:         end if
 9:     end if
10:     UnLock( )
11:     return
12: end function
```

图 3-3　代码片段示例

代码片段 $CodeFragments$ 如图3-3所示。重复执行框架 $RepeatFramework$ 为第四、五行，用来保证到达指定次数重复后执行代码片段 $CodeFragments$。顺序执行框架 $SequenceFramework$ 为第三、六行，保证代码片段 $CodeFragments$ 只有按照指定的顺序才会执行。并行执行框架 $ConcurrencyFramework$ 为第二、十行，保证代码片段 $CodeFragments$ 在并行环境中正确的执行。

3）恶意行为获取算法

```
输入: 恶意代码 MaliciousCode
输出: ActionNumber 个恶意行为 Action
 1: function GetAction(MaliciousCode)
 2:     DataFlowDiagram ← MaliciousCode
 3:     ActionNumber ← Number of DataFlowDiagram
 4:     for i = 0 → ActionNumber do
 5:         CodeFragments[i] ← DataFlowDiagram[i]
 6:     end for
 7:     return ActionNumber, Action
 8: end function
```

图 3-4　恶意行为获取算法

恶意行为获取算法具体如图3-4所示。输入为恶意代码 $MaliciousCode$。输出为 $ActionNumber$ 个恶意行为 $Action$。恶意行为 $Action$ 指的是单个恶意的行为。比如打开文件，读取文件，发送数据，关闭文件这一系列行为，任意的单个行为都不是恶意的，但是这一系列行为就是一个窃取数据的恶意行为 $Action$。首先根据恶意代码 $MaliciousCode$ 得到数据流图 $DataFlowDiagram$。这里使用数据流图

$DataFlowDiagram$ 来区分单个恶意行为 $Action$，数据流图 $DataFlowDiagram$ 中相互之间有数据依赖关系的行为属于同一个恶意行为 $Action$，相互之间没有数据依赖关系的行为属于不同的恶意行为 $Action$。所以数据流图 $DataFlowDiagram$ 中每一个连通图就是一个恶意行为 $Action$。数据流图 $DataFlowDiagram$ 中连通图的个数就是恶意行为的个数 $ActionNumber$。

4）无害代码获取算法

无害代码获取算法具体如图3-5所示。输入为单个恶意行为 $Action$。输出为 $HarmlessCodeNumber$ 个无害代码片段 $HarmlessCode$, *和相关变量* ***CorrelatedVariable***。单个代码不会产生恶意行为，只有足够多的恶意代码才会构成恶意行为 $Action$。将恶意行为 $Action$ 中的 $Code$ 逐个分配到 $HarmlessCode[HarmlessCodeNumber]$ 中。如果 $Code \cup HarmlessCode[HarmlessCodeNumber]$ 是恶意的，那么将 ***HarmlessCodeNumber*** 加一，将 $Code$ 分配到新的 $HarmlessCode[HarmlessCodeNumber]$ 中。当得到所有的无害代码片段 $HarmlessCode$ 后，将所有 $HarmlessCode$ 中用到的变量两两取交集，然后取并集，最后得到 $CorrelatedVariable$

输入: 恶意行为 $Action$
输出: $HarmlessCodeNumber$ 个无害代码片段 $HarmlessCode$, 相关变量 $CorrelatedVariable$

function GETHARMLESSCODE($Action$)
  $HarmlessCodeNumber \leftarrow 1$
  $i = HarmlessCodeNumber$
  for $Code \in Action$ **do**
    if $Code \cup HarmlessCode[i] \quad is \quad Malicious$ **then**
      $HarmlessCodeNumber = HarmlessCodeNumber + 1$
      $i = HarmlessCodeNumber$
    end if
    $HarmlessCode[i] \leftarrow HarmlessCode[i] \cup Code$
  end for$CorrelatedVariable \leftarrow \emptyset$
  for $i = 1 \rightarrow HarmlessCodeNumber - 1$ **do**
    for $j = 1 \rightarrow HarmlessCodeNumber - 1 - i$ **do**
      $Variable \leftarrow Variable \in HarmlessCode[i] \cap Variable \in HarmlessCode[i + j]$
      $CorrelatedVariable \leftarrow CorrelatedVariable \cup Variable$
    end for
  end for
  return $HarmlessCodeNumber, HarmlessCode, CorrelatedVariable$
end function

图 3-5 无害代码获取算法

至此得到了恶意代码拆分完毕的恶意代码片段 ***CodeFragments***，简称为 ***CFn***，其中 ***n*** 为恶意代码片段的顺序。

3.2 插入点选取

并行程序在本质上与串行程序不同。串行程序在同一时间只有一个函数在执行，而并行函数同一时间可以有若干不同或者相同的函数在执行。

并行函数示例如图3-6所示, 其执行时间顺序示意图如3-7所示。首先线程 ***Thread0*** 调用函数 ***Function1***。在时间 t_0 到 t_1 之间，只有函数 ***Function1*** 在执行。这段时间执

```
1:  function MAIN( )
2:      CALL(Function1)
3:      CREATE(Thread1,Function2)
4:      CREATE(Thread2,Function3)
5:      WAIT(Thread1)
6:      WAIT(Thread2)
7:      CREATE(Thread1,Function4)
8:      CREATE(Thread2,Function4)
9:      return
10: end function
```

图 3-6　并行函数示例

行情况与串行程序相同。函数 *Function1* 执行完后然后创建线程 *Thread1* 调用函数 *Function2*，创建线程 *Thread2* 调函数用 *Function3*。在时间 t_1 到 t_2 之间，函数 *Function2* 和函数 *Function3* 在执行。这段时间两个不同的函数 *Function2* 和函数 *Function3* 在同时执行，这两个函数可能会产生各种各样的交织。等待线程 *Thread1* 和线程 *Thread2* 执行完后，创建线程 *Thread1* 和线程 *Thread2* 调函数用 *Function4*。在时间 t_2 到 t_3 之间，两个函数 *Function4* 在执行。这段时间两个相同的函数 *Function4* 在同时执行，这两个相同的函数产生各种各样的交织要远比在时间 t_1 到 t_2 之间两个不同的函数复杂。

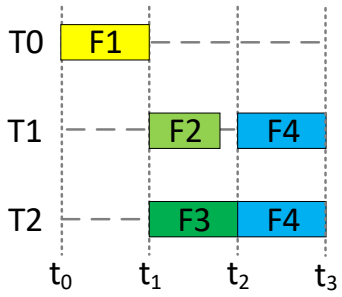
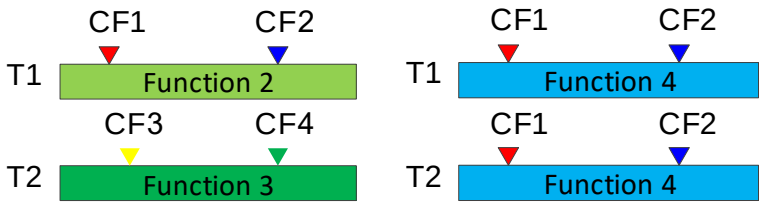


图 3-7　并行程序中的三种函数

两种插入点选取方式如图3-8所示。在函数 *Function2* 和函数 *Function3*、函数 *Function4* 中均插入两个恶意代码片段。



(a) 不同函数　　(b) 相同函数

图 3-8　两种插入点选取方式

1）两个不同函数插入四个恶意代码片段

在两个不同的函数 *Function2* 和函数 *Function3* 中插入四个恶意代码片段如图3-8(a)所示。其中函数 *Function2* 中插入恶意代码片段 *CF1* 和 *CF2*，函数 *Function3* 中插入恶意代码片段 *CF3* 和 *CF4*。根据线程 *Thread1* 和线程 *Thread2* 交织的不同，四个恶意代码片段发生的序列也不同，最终真正执行的序列也不同。

具体线程交织如表3-1所示。其中，*序号*代表每一个线程交织的序号，*线程交织*代

表 3-1　两个不同函数插入四个恶意代码片段线程交织

序号	线程调度	线程交织	触发序列	触发
I1	*T1-T1-T2-T2*	*CF1-CF2-CF3-CF4*	*CF1-CF2-CF3-CF4*	是
I2	*T1-T2-T1-T2*	*CF1-CF3-CF2-CF4*	*CF1-CF2*	否
I3	*T1-T2-T2-T1*	*CF1-CF3-CF4-CF2*	*CF1-CF2*	否
I4	*T2-T1-T1-T2*	*CF3-CF1-CF2-CF4*	*CF1-CF2*	否
I5	*T2-T1-T2-T1*	*CF3-CF1-CF4-CF2*	*CF1-CF2*	否
I6	*T2-T2-T1-T1*	*CF3-CF4-CF1-CF2*	*CF1-CF2*	否

表两个线程交织的顺序，*发生序列*表示插入的四个恶意代码片段发生的序列，*触发序列*表示真正执行的恶意代码片段序列，*触发*表示最终是否触发完整的恶意代码。这里为了更直观的解释，采用 *CF1-CF2-CF3-CF4* 的顺序作为触发顺序，事实上可以采取任意顺序，比如 *CF1-CF2-CF3-CF4*。在两个不同函数中插入四个恶意代码片段共计可以发生六种线程交织。其中只有线程交织 *I1* 可以触发以 *CF1-CF2-CF3-CF4* 为顺序的恶意代码片段。

2）两个相同函数插入四个恶意代码片段

在两个相同的函数 *Function4* 中插入两个恶意代码片段如图3-8(b)所示。其中函数 *Function4* 中插入恶意代码片段 *CF1* 和 *CF2*。两个线程均调用函数 *Thread4*，所以恶意代码片段 *CF1* 和 *CF2* 均会发生两次。根据线程 *Thread1* 和线程 *Thread2* 交织的不同，两个恶意代码片段各两次发生的序列也不同，最终真正执行的序列也不完全相同。

表 3-2　两个相同函数插入四个恶意代码片段线程交织

序号	线程调度	线程交织
I1	*T1-T1-T2-T2*	*CF1(1)-CF2(1)-CF1(2)-CF2(2)*
I2	*T1-T2-T1-T2*	*CF1(1)-CF1(2)-CF2(1)-CF2(2)*
I3	*T1-T2-T2-T1*	*CF1(1)-CF1(2)-CF2(2)-CF2(1)*
I4	*T2-T1-T1-T2*	*CF1(2)-CF1(1)-CF2(1)-CF2(2)*
I5	*T2-T1-T2-T1*	*CF1(2)-CF1(1)-CF2(2)-CF2(1)*
I6	*T2-T2-T1-T1*	*CF1(2)-CF2(2)-CF1(1)-CF2(1)*

具体线程交织如表3-2所示。其中，*序号*代表每一个线程交织的序号，*线程交织*代表两个线程交织的顺序，*发生序列*表示插入的四个恶意代码片段发生的序列。在同一的函数中插入两段恶意代码，并且让两个线程调用这个相同的函数，线程交织有六种。

表 3-3　两种不同触发策略

序号	触发顺序：*CF2<CF1*		触发顺序：*CF1<CF2*	
	执行情况	触发	执行情况	触发
I1	*CF1(1,F)-CF2(1,T)-CF1(2,T)-CF2(2,F)*	是	*CF1(1,T)-CF2(1,T)-CF1(2,F)-CF2(2,F)*	是
I2	*CF1(1,F)-CF1(2,F)-CF2(1,T)-CF2(2,F)*	否	*CF1(1,T)-CF1(2,F)-CF2(1,T)-CF2(2,F)*	是
I3	*CF1(1,F)-CF1(2,F)-CF2(2,T)-CF2(1,F)*	否	*CF1(1,T)-CF1(2,F)-CF2(2,T)-CF2(1,F)*	是
I4	*CF1(2,F)-CF1(1,F)-CF2(1,T)-CF2(2,F)*	否	*CF1(2,T)-CF1(1,F)-CF2(1,T)-CF2(2,F)*	是
I5	*CF1(2,F)-CF1(1,F)-CF2(2,T)-CF2(1,F)*	否	*CF1(2,T)-CF1(1,F)-CF2(2,T)-CF2(1,F)*	是
I6	*CF1(2,F)-CF2(2,T)-CF1(1,T)-CF2(1,F)*	是	*CF1(2,T)-CF2(2,T)-CF1(1,F)-CF2(1,F)*	是

两段恶意代码的触发顺序共两种，分别为 *CF1<CF2* 和 *CF2<CF1*。具体触发情况如表3-3所示。例如线程交织 *I1* 的在触发顺序 *CF2<CF1* 下的执行情况 *CF1(1,F)-CF1(2,T)-*

CF2(1,T)-CF2(2,F)，其中 *CF1(1,F)* 表示这个恶意代码片段 *CF1* 是线程 *Thread1* 中的，并且执行情况是 *false(F)*。在触发顺序为 *CF2<CF1* 的情况下，六种线程交织共有两种可以触发。

可以看出，在相同的函数中插入代码的线程交织远比在不同的函数中插入代码的线程交织复杂。因此，本文主要讨论插入相同函数的情况。

3.3 线程交织概率

第3.2节分析了并行程序中可以选择的拆入点，并且给出了两个线程时的六种线程交织，以及两种序列在这六种交织中的发生情况。但是，在实际环境中，这六种线程交织并不是等概率事件，即便在相同的环境下概率也不完全相同，在不同的环境下概率更是会发生新的变化。在不同的环境下线程在 *CPU* 上的调度情况也会发生变化，虽然调度策略没有变化，但是不同的环境下不同的资源会成为主要影响因素。并且这六种线程交织仅仅是两个线程所产生的线程交织，如果有三个、四个乃至多个线程那么会产生更复杂的交织。在相同的函数中插入两段恶意代码产生的序列有两种，每一种序列在不同的线程交织中发生的情况也不一完全样。多段恶意代码片段产生的序列也会更加复杂，复杂的序列和复杂的线程交织组合会产生更为复杂的概率。

3.3.1 线程调度

并行程序与串行程序最大的不同就是并行程序可以同时运行多个函数，即便是在单核处理器上。正如上文讨论，现代操作系统提供的线程环境采用优先级加抢占式分配方案对线程分配 *CPU* 时间片[26]。然而，在具体的不同条件下会产生完全不同的线程交。在不考虑优先级的情况下，主要的影响因素可以分为两类：

影响线程是否处于就绪态的因素。

影响就绪态的线程是否分配 *CPU* 时间片的因素。

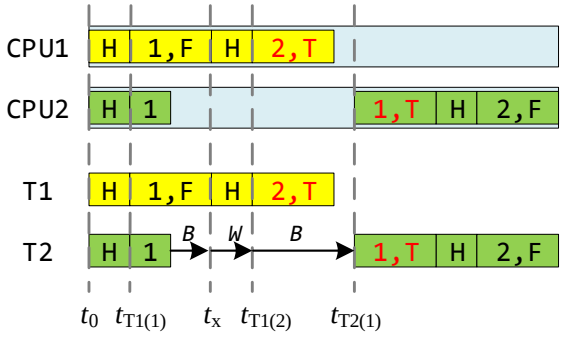
本文只详细分析被锁保护和高低负载两种情况下的线程交织，其他情况不做详细的分析。其中的函数均可以在一个 *CPU* 时间片内执行完毕，超出一个时间片的情况暂时不做讨论。其中被锁保护属于对线程是否属于就绪态的影响因素，高低负载属于对就绪线程是否分配 *CPU* 时间片的因素。

这样的线程调度机制在双核 *CPU* 上两个线程分别插入两段恶意代码片段的具体调度情况如图3-9和图3-10所示。图中 *H* 代表 *host*，表示宿主程序本身的代码。图中标有 *1/2,T/F* 代表恶意代码片段。其中 *1/2* 代表序号，表明恶意代码的插入顺序。*T/F* 代表 *true/false*，表明此段恶意代码是否触发。这里以 *CF2<CF1* 的触发顺序举例说明，其中所有触发的恶意代码片段均以红字标出。图中 *B* 代表 *block*，表明线程处于阻塞态，等待其他线程释放锁。图中 *W* 代表 *wait*，表明线程处于就绪态，等待系统分配 *CPU* 时间片。图中 t_0 表示开始分析的时刻，$t_{T1(1)}$ 表示线程 *T1* 第 1 段恶意代码开始执行的

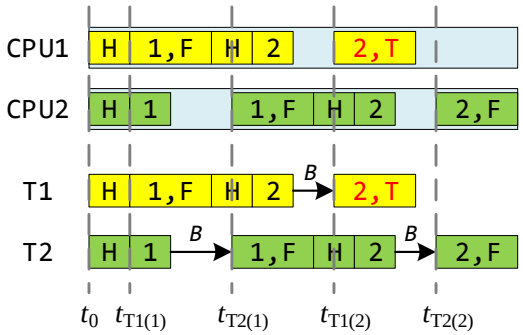
时刻，图3-9(a)中 t_x 表示线程 *T1* 第 1 段恶意代码执行完释放锁的时刻，图3-10中 t_y 表示线程 *T1* 第 2 段恶意代码执行完释放锁的时刻。

1）锁对线程调度的影响

并行逻辑炸弹需要将拆分好的被所保护的恶意代码片段插入到宿主程序中。对于一个线程中遇到加锁的部分，线程首先需要得到锁。如果顺利得到锁，那么线程继续以运行态运行。如果锁已经被其他线程所使用，那么该线程就会由运行态转为阻塞态，并且等待着锁的到来。当锁到来时，所有阻塞态等待该锁的线程均会得到消息，每个线程得到锁的机会均等。但是阻塞态的线程是没有 *CPU* 时间片的，阻塞态的线程必须转变为就绪态先得到 *CPU* 时间片进入运行态。只有处于运行态的线程得到了锁，锁才彻底的分配给了该线程。



(a) *CPU* $t_{Host} < t_{Get-CPU}$



(b) *CPU* $t_{Host} > t_{Get-CPU}$

图 3-9　线程在 *CPU* 上的调度

处于阻塞态的线程在锁被释放后，阻塞态转变为就绪态。处于就绪态的线程需要时间等待操作系统分配 *CPU* 时间片，此时该线程并没有真正得到锁。但是，如果在这段时间内锁被其他处于运行态的线程得到，那么该线程将会重新变为阻塞态，如图3-9(a)所示。在 t_0 时刻，两个线程 *T1* 和 *T2* 同时分别在 *CPU1* 和 *CPU2* 上运行，各自执行自己的 *H* 部分。在 $t_{T1(1)}$ 时刻，两个线程 *T1* 和 *T2* 同时竞争锁，被线程 *T1* 随机的得到了锁。线程 *T1* 继续执行恶意代码 *1*，*F*，线程 *T2* 处于阻塞态。在 t_x 时刻，线程 *T1* 释放锁，继续执行 *H* 部分，线程 *T2* 则从阻塞态转变为就绪态，等待操作系统分配 *CPU* 时间片。在 $t_{T1(2)}$ 时刻，线程 *T1* 重新得到锁，执行恶意代码 *2*，*T*，线程 *T2* 还未得到 *CPU* 时间片，从就绪态转变为阻塞态。在 $t_{T2(1)}$ 时刻，线程 *T2* 得到锁继续运行。

如果处于就绪态的线程在这段时间不仅得到了 *CPU* 时间片，还得到了锁，其线程调度如图3-9(b)所示。同样，在 t_0 时刻，两个线程 *T1* 和 *T2* 同时分别在 *CPU1* 和 *CPU2* 上运行，各自执行自己的 *H* 部分。在 $t_{T1(1)}$ 时刻，两个线程 *T1* 和 *T2* 同时竞争锁，被线程 *T1* 随机的得到了锁。线程 *T1* 继续执行恶意代码 *1*，*F*，线程 *T2* 处于阻塞态。但是，在 $t_{T2(1)}$ 时刻，线程 *T2* 得到锁，执行恶意代码 *1*，*F*。线程 *T1* 在执行完 *H* 部分后，由于锁被线程 *T2* 得到，转变为阻塞态。在 $t_{T1(2)}$ 时刻，线程 *T1* 重新得到锁，执行恶意代码 *2*，*T*。线程 *T2* 在执行完 *H* 部分后，由于锁被线程 *T1* 得到，转变为阻塞态。在 $t_{T2(2)}$ 时刻，线程 *T2* 重新得到锁，执行恶意代码 *2*，*F*。

以 *CF2*<*CF1* 的触发顺序举例说明，图3-9(a)所示情况触发完整序列，图3-9(b)则仅触发 *CF2*。

2）负载对线程调度的影响

锁会影响线程在阻塞态与就绪态之间的转换，进一步影响线程的调度。*CPU* 的负载会影响就绪态的线程是否能获得 *CPU* 时间片，进一步影响线程的调度。处于就绪态的线程按照优先级加抢占式分配 *CPU* 时间片这种资源。但是如果 *CPU* 时间片非常少，那么处于就绪态的线程也只能等待系统分配。如图3-10所示，其中 *CPU2* 被其他高优先级程序占有，操作系统只有 *CPU1* 的资源可以分配。

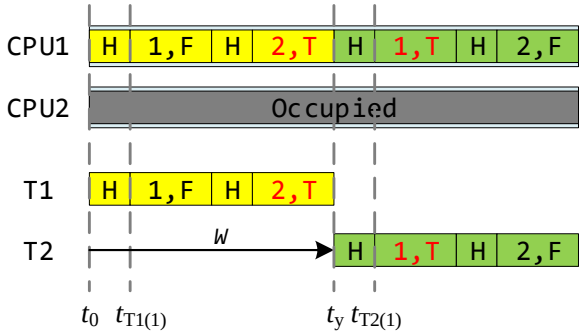


图 3-10　线程在高负载 *CPU* 上的调度

在 t_0 时刻，两个线程 *T1* 和 *T2* 同时竞争 *CPU* 时间片，被线程 *T1* 随机的得到。在 $t_{T1(1)}$ 时刻，只有线程 *T1* 竞争锁，被线程 *T1* 的得到了锁，线程 *T1* 继续执行。直到 t_y 时刻，处于就绪态的线程 *T2* 得到 *CPU* 时间片，执行恶意代码 *1*，*T*。在 $t_{T2(1)}$ 时刻，只有线程 *T2* 竞争锁，被线程 *T2* 的得到了锁，线程 *T2* 继续执行。

综上所述，可以得出如下结论：

在操作系统低负载时，影响并行逻辑炸弹线程调度的主要因素是锁的归属。而 *H* 部分的时间长度影响着锁的调度，在一个 *CPU* 时间片内改变 *H* 部分的长度可以影响锁的线程归属，进而影响线程的调度。*H* 部分长度越长，新的线程得到锁的概率越大，线程切换概率越大。

在操作系统高负载时，影响并行逻辑炸弹线程调度的主要因素是 *CPU* 时间片。由于 *CPU* 时间片紧缺，得到 *CPU* 时间片的线程可以执行，没有得到 *CPU* 时间片的线

程只有等待操作系统分配。操作系统负载越高，新的线程得到 *CPU* 时间片的概率越小，线程切换概率越小。

3.3.2　交织种类和序列种类

将 *N* 段恶意代码片段插入 *M* 个执行相同函数的线程中，交织的种类数量巨大，序列的种类数量巨大。而且不同的交织中不同的序列发生的情况不同，不同的交织发生的概率也不同。

1）交织种类

对于 *M* 个执行相同函数的线程，每个线程插入 *N* 段恶意代码片段，可能的线程交织的数目为 $(MN)!/(N!)^M$ 种。*N* 段恶意代码片段插入 *M* 个执行相同函数的线程中序列的总长度为 *MN*，所以其全排列为 *(MN)!* 种。对于每个线程，其 *N* 段恶意代码片段的全排列为 *N!* 种。但是由于拆入顺序已经固定，所以只会发生一种。又因为有 *M* 个执行相同函数的线程，所以线程交织的全排列 *(MN)!* 需要除恶意代码片段的 *N!* 共计 *M* 次。最终所有可能的线程交织的数目为 $(MN)!/(N!)^M$ 种。

表 3-4　不同线程数目恶意代码片段数目下线程交织数目

线程数目	恶意代码片段数目	线程交织数目
2	2	6
2	3	20
3	2	90
3	3	1,680
3	4	34,650
4	3	396,000
4	4	63,063,000

表3-4列出了在 2、3、4 个线程中插入 2、3、4 段恶意代码片段线程交织的数目。可以看出即便是 4 个完全相同的线程，在插入 4 段恶意代码片段后其可能的线程交织竟然达到了 63,063,000 种。数目巨大的线程交织印证了并行程序的复杂性，也说明了检测的困难。

2）序列种类

对于 *N* 段恶意代码片段，其可能的序列种类共 *N*! 种。这里为了方便说明，使用 *1234* 代替 *CF1-CF2-CF3-CF4*。同理，*4321* 代表 *CF4-CF3-CF2-CF1*。

为了更好的对序列进行分类，本文提出了序列模式理论对序列进行唯一表示并且按照概率分类。首先定义顺序度和逆序度。

逆序度：对于序列 *abcd*，如果 *a>b*，那么称 *(a-b)* 为 *ab* 的逆序度。

顺序度：对于序列 *abcd*，如果 *a<b*，那么称 *(b-a)* 为 *ab* 的顺序度。

按照逆序度和顺序度可以将每一个序列唯一表示，称之为模式。定义 + 表示顺序度，– 表示逆序度，之间用 | 分开。

以四个执行相同函数的线程中插入四段恶意代码为例说明。逆序的发生不可能出现在单独的线程中，一个线程执行必然按照顺序执行。一个线程中，序列最多发生零

次逆序；二个线程中，序列最多发生一次逆序；*M* 个线程中，序列最多发生 *(M-1)* 次逆序。所以逆序发生的次数相比于逆序度的大小对概率的影响更大。线程切换概率越小，各个线程执行时间越不统一，各个线程执行位置越不统一，发生逆序的可能性越大。同样，线程切换概率越大，各个线程执行执行时间越统一，各个线程执行位置越统一，发生逆序的可能性越小。根据第3.3.1节的分析，在操作系统低负载时，*host* 代码长度主要影响线程切换概率。*host* 代码长度越长，发生逆序的概率越低。*host* 代码长度越短，发生逆序的概率越高。在操作系统高负载时，负载主要影响线程切换概率。操作系统负载越低，逆序发生概率越低。操作系统负载越高，逆序发生概率越高。考虑到序列是在交织中的序列，而完整交织必然是以 1 开头，以 4 结尾，并且其中必然有 4 个不同的 1234 序列，这些情况的发生概率为 1。顺序度和逆序度表示序列是为了对概率进行分类，所以在计算顺序度或者逆序度时需要根据这些必然事件对模式做出调整以简化。在计算顺序度时，首位序列的顺序度计算为 *a-1*，因为交织必然以 1 开头。相邻的顺序度可以合并，因为交织必然包括 4 个不同的 1234 序列。末位如果是顺序度，那么顺序度省略，因为无论顺序度计算出多少，交织必然以 4 结尾。例如对于序列 *abcd*，如果 *a*<*b* 并且 *b*<*c*、*c*<*d*，那么 *abcd* 的顺序度为 $+0|+1|+1|+1$。然后简化为 $+3$，最终变为 $\emptyset$。

最终，4 段恶意代码片段的所有可能序列如表3-5所示。

表 3-5　4 段恶意代码片段的所有可能序列及其模式

序列	模式	序列	模式	序列	模式
1234	$\emptyset$	1243	$+3\|-1$	1324	$+2\|-1$
1342	$+3\|-2$	1423	$+3\|-2$	1432	$+3\|-1\|-1$
2134	$+1\|-1$	2143	$+1\|-1\|+3\|-1$	2314	$+2\|-2$
2341	$+1\|-1$	2413	$+3\|-3$	2431	$+3\|-1\|-2$
3124	$+2\|-2$	3142	$+2\|-2\|+3\|-2$	3214	$+2\|-1\|-1$
3241	$+2\|-1\|+2\|-3$	3412	$+3\|-3$	3421	$+3\|-2\|-1$
4123	$+3\|-3$	4132	$+3\|-3\|+2\|-1$	4213	$+3\|-2\|-1$
4231	$+3\|-2\|+1\|-2$	4312	$+3\|-1\|-2$	4321	$+3\|-1\|-1\|-1$

可以发现，某些序列拥有共同的模式。而模式是为了表征序列的概率，所以这些具有相同模式的序列其概率也完全相同。例如对于模式 $+3|-3$，将其按照交织的特点扩展，得到其交织中必然包括序列 12341234。而序列 12341234 包括所有拥有模式 $+3|-3$ 的序列 4123、2413、2341、3412。同样，序列 4123、2413、2341、3412 出现并且仅出现在包括序列 12341234 的线程交织中。模式与序列、概率是完全等同对应的不同表示。

将具有相同的模式整理如表3-6所示。24 种序列被 17 种模式完全表示并且进行了分类。对于模式 $+2|-2$，有两种序列 $2314, 3124$。对于模式 $+3|-2$，有两种序列 $1423, 1342$。对于模式 $+3|-2|-1$，有两种序列 $3421, 4213$。对于模式 $+3|-1|-2$，有两种序列 $4312, 2431$。而对于模式 $+3|-3$，有多达四中序列 $4123, 2413, 2341, 3412$。

而考虑到顺序度的必然性，顺序度虽然也会影响概率，但是相对于逆序度影响较

表 3-6　4 段恶意代码片段的所有可能模式及其序列

模式	序列
$\emptyset$	1234
$+1\|-1$	2134
$+2\|-1$	1324
$+3\|-1$	1243
$+2\|-2$	$2314, 3124$
$+3\|-2$	$1423, 1342$
$+3\|-3$	$4123, 2413, 2341, 3412$
$+2\|-1\|-1$	3214
$+3\|-1\|-1$	1432
$+1\|-1\|+3\|-1$	2143
$+2\|-2\|+3\|-2$	3142
$+3\|-2\|+1\|-2$	4231
$+3\|-2\|-1$	$3421, 4213$
$+3\|-1\|-2$	$4312, 2431$
$+2\|-1\|+2\|-3$	3241
$+3\|-3\|+2\|-1$	4132
$+3\|-1\|-1\|-1$	4321

小。故如果只是考虑相近的概率的情况下，可以将顺序度省略。逆序度对概率的影响更多的是是否有逆序，有几个逆序。至于逆序度的大小也会影响概率，但是相差不大。同样如果只是考虑相近的概率的情况下，可以将逆序度进行合并。将顺序度省略逆序度合并后的模式整理如表3-7所示。

表 3-7　4 段恶意代码片段的所有相似模式及其序列

模式	序列
$\emptyset$	1234
-1	$1243, 1324, 2134$
$-1\|-1$	2143
$-1\|-1\|-1$	4321
$-2/3$	$1342, 1423, 2314, 3124, 2341, 2413, 3412, 4123$
$-1/2/3\|-1/2/3$	$1432, 3214, 2431, 4312, 3241, 3421, 4213, 4132, 3142, 4231$

最终得到六种不同模式的序列。注意，对于模式 $-1/2/3|-1/2/3$，其中不包括模式 $-1|-1$，包括 $-1/2/3|2/3$ 和 $2/3|-1/2/3$。按照模式的分类，同一模式的序列的概率大致相似。而且按照理论推测，序列的模式包括次数少逆序的概率要高于次数多逆序的概率，低逆序度的概率要高于高逆序度的概率。所以理论上表中的六种模式的概率从高到低排序有：

$$\emptyset > -1 > -2/3 \tag{3-1}$$

$$-1 > -1|-1 > -1|-1|-1 \tag{3-2}$$

$$-1|-1 > -1/2/3|-1/2/3 \tag{3-3}$$

最终，本文在每种模式分类中选出一个代表序列，作为后续分析的主要序列。

当然，这些都是按照序列模式理论分析出的概率，具体真实情况需要通过实验进行统计分析。第3.3.3节介绍了具体的概率并且通过实验统计出了各种序列在不同交织下的概率。概率事实证明了序列模式理论的正确性。本文仅仅将序列模式理论应用在

了四个线程插入四段恶意代码中的分析，事实上序列模式理论可以应用在任意线程中插入任意段恶意代码的情况中。

表 3-8　4 段恶意代码片段的所有相似模式及其代表序列

模式	序列
$\emptyset$	1234
-1	1243
$-1\vert-1$	2143
$-1\vert-1\vert-1$	4321
$-2/3$	4123
$-1/2/3\vert-1/2/3$	3241

3.3.3　序列概率

由于序列种类巨大，线程交织复杂，各种限制条件多变，每种序列在不同环境发生的概率几乎是不可计算的。因此本文采用了抽样统计的方法对各种序列的概率进行统计。个体是执行一次产生的线程交织，总体是执行无数次产生的所有线程交织之和，样本是执行一定次数产生的线程交织。所以这个问题是一个有放回的简单随机重复抽样问题，所需要的样本容量如公式(3-4)和公式(3-5)所示[48]。

$$n = \frac{n_0}{1 + \frac{n_0 - 1}{N}} \tag{3-4}$$

$$n_0 = \frac{u^2}{r^2} * (1 - P)P \tag{3-5}$$

公式(3-4)和公式(3-5)中，N 代表总体，这里 $N = \infty$。式中 n_0 代表有放回的所需样本量，n 代表无放回的所需样本量。在 95% 的置信度下，$u = 1.96$，r 代表相对误差，这里本文取 $r = 0.01$。因为 $N = \infty$，所以根据公式(3-4)，近似有 $n \approx n_0$。将 $u = 1.96$ 和 $r = 0.01$ 带入，又根据 $(1 - P)P \leq 0.25$，根据公式(3-5)可得 $n \leq 10000$。所以在后续的实验中本文选取的样本数量 $n = 10000$，可以保证置信度在 95% 以上，误差在 0.01 以下。

根据第3.3.1节的分析，影响线程调度有两个主要因素，分别是锁和负载。对于锁而言，在实际中影响锁是否就绪的主要为执行 *host* 代码的时间。因为各种代码各自的执行时间不同，无法统一度量，这里本文采取一个循环代码 *loop* 来度量 *host* 代码的执行时间。通过测量不同的 *loop* 情况下不同的线程调度导致的各种序列的概率来度量不同实际代码的下各种序列的概率。第3.3.3.1节统计了不同 *loop* 代码的长度对序列概率的影响。对于负载而言，只需要测量不同负载下不同的线程调度导致的各种序列的概率。第3.3.3.2节统计了不同负载对序列概率的影响。在确定了合适的 *loop* 代码长度和负载后，第3.3.3.3节统计出各种序列的概率。

概率统计的实验环境如下：操作系统为 *Windows 7*，处理器为 *Intel(R)Core(TM)i5-3470*，内存 *16GB*。

1）不同 *loop* 代码的长度对序列概率的影响

前面分析了 *Windows* 操作系统一个时间片长度为 *20ms*，而第3.3.1节关于 *host* 代码影响线程调度的分析建立在一个时间片长度中。所以采取 *loop* 代码的长度对 *host* 代码度量，也需要保证执行时间在 *20ms* 内。为此本文首先测量了不同长度 *loop* 代码的执行时间，如表3-9所示。

表 3-9　不同 *loop* 代码的长度对应的时间

loop	时间/s
10	$1.53651e-006$
100	$1.67619e-006$
1000	$4.95873e-006$
10000	$4.06476e-005$
100000	$3.52349e-004$
1000000	$3.53495e-003$
10000000	$3.53671e-002$
100000000	$3.51403e-001$
1000000000	3.38357

可以看出在 *loop* 的长度为 10000000 时执行时间为 $3.53671e-002s$ 已经超过 *20ms*。所以测量不同 *loop* 代码的长度对序列概率的影响只测量从 $0 \backsim 60000000$ 的范围，保证可以在一个时间片内执行完 *loop* 的代码。

事实上从不同长度 *loop* 代码的执行时间可以看出，在 *loop* 代码长度较长时执行时间基本按照线性变化。而在 *loop* 代码长度较短时执行时间则不按照线性变化，因为这种情况下线程开始的准备工作时间占的比重更大。侧面印证了上文关于线程调度的分析。

本文统计了从 $0 \backsim 60000000$ 的 *loop* 代码对序列概率的影响，并且截取了其中有实际变化意义的部分。低负载下 *loop* 代码影响触发次数变化折线如图3-11所示，高负载下 *loop* 代码影响触发次数变化折线如图3-13所示。纵坐标 *10000 次实验触发次数*采用以 10 为低的对数坐标，横坐标为截取了从 $1000 \backsim 200000$ 的范围，间隔为 10000。

在图3-11低负载下 *loop* 代码影响触发次数变化折线可以看出，随着 *loop* 的逐渐变大，*loop* 代码长度变长，*loop* 代码执行时间变长，各种序列触发的次数逐渐变少。

因为这里采取的是对数坐标，所以各种序列的触发次数从图中看起来比较分散。但是实际上各种序列的触发次数非常集中，所以才会选择纵坐标采取对数坐标。在 *loop* 到达 100000 后，各种序列的触发次数迅速下降，触发次数最多仅仅 3054，也就是 30% 的触发概率。

事实上，可以看出各种序列的变化趋势可以归为几类相同的或者相似的。大致可以归为六类，正好对应表3-8分析选出的代表序列。所以按照之前分析选出的代表序列将图3-11低负载下 *loop* 代码影响触发次数变化折线简化如图3-12代表序列在低负载下 *loop* 代码影响触发次数变化折线。第一类代表序列为 1234，触发次数恒定为 10000。第二类代表序列为 1243，触发次数在 *loop* 为 60000 时开始下降，在 *loop* 为 120000 时

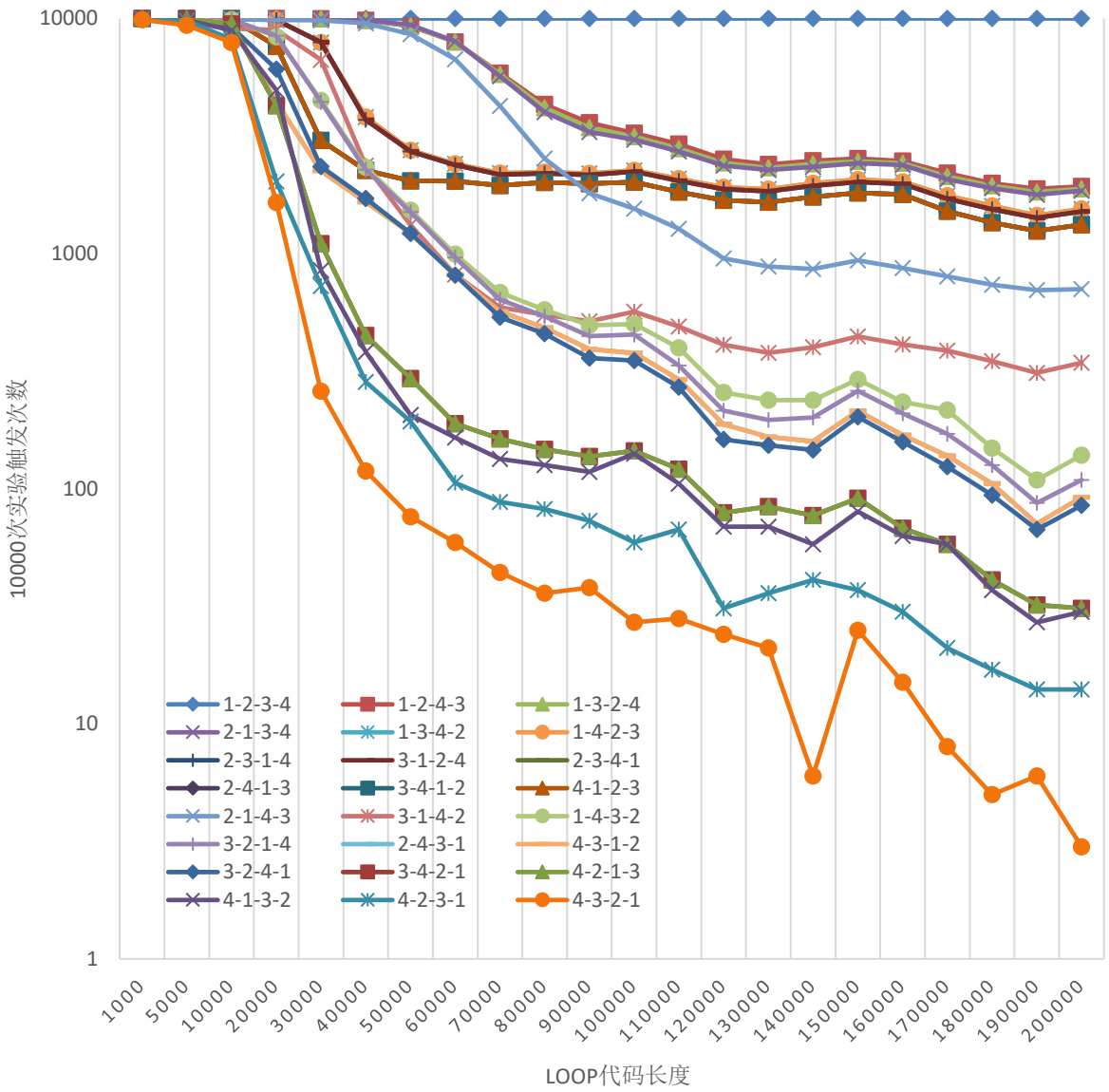


图 3-11　低负载下 *loop* 代码影响触发次数变化折线

下降至 3000 左右。第三类代表序列为 2143，触发次数在 *loop* 为 60000 时开始下降，在 *loop* 为 120000 时下降至 1000 左右。第四类代表序列为 4321，触发次数在 *loop* 为 30000 时开始下降，在 *loop* 为 40000 时下降至 100 以下。第五类代表序列为 4123，触发次数在 *loop* 为 30000 时开始下降，在 *loop* 为 40000 时下降至 2000 以下。第六类代表序列为 3241，触发次数在 *loop* 为 30000 时开始下降，在 *loop* 为 40000 时下降至 1000 以下。

从这些代表序列的触发次数变化趋势可以看出其触发概率大小完全符合公式3-1所示规律，侧面印证了序列模式理论的正确性。除了概率大小之外，逆序度相同的代表序列变化发生的节点都比较相近，逆序的次数又会有规律的影响其发生概率的大小，并且其变化趋势也较为相近。

值得一提的是序列 4321 在 *loop* 为 14000 时出现了看起来明显的波动。事实上由于采用对数坐标看起来波动明显，实际上绝对变化很小。

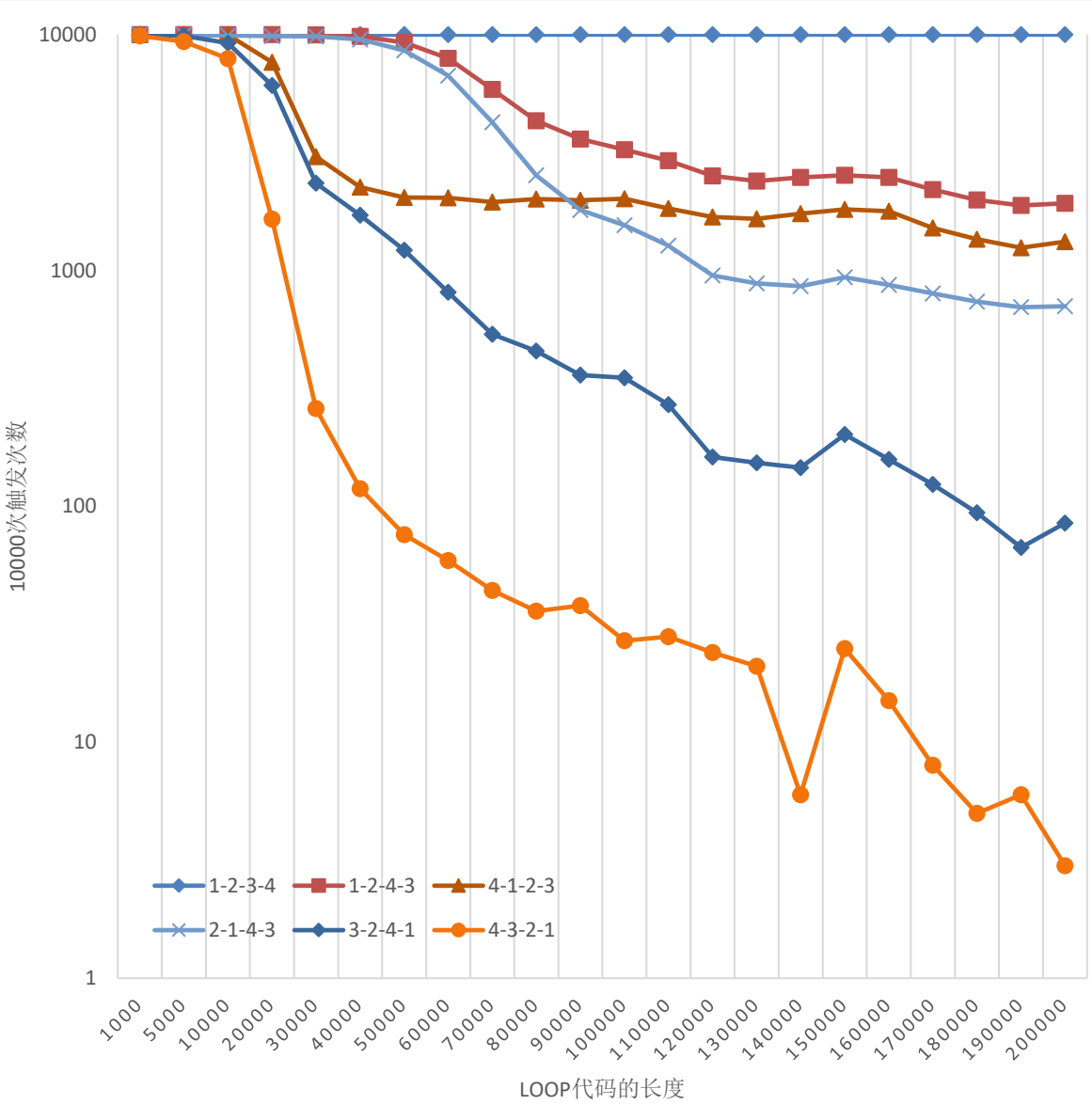


图 3-12　代表序列在低负载下 *loop* 代码影响触发次数变化折线

图3-13高负载下 *loop* 代码影响触发次数变化折线可以看出果然如理论上的分析。各种序列在高负载的情况下 *loop* 代码长度的变化不再是主要因素。主要因素为负载影响 *CPU* 最终影响线程调度。在高负载下各种序列在 *loop* 变化的情况下始终保持恒定的高触发次数。

由于需要序列在高低负载变化明显，因此选择 *loop* 为 120000 为最合适的 *loop* 代码长度。

但是考虑到各种序列在 *loop* 在 30000、40000、60000 和 120000 时均发生了显著的变化，后续实验中将选择其他三个 *loop* 值作为对比。

2）不同负载对序列概率的影响

除了不同 *host* 代码长度外，另一个影响线程调度的重要因素就是操作系统的负载。根据第3.3.1节的分析，随着操作系统负载的提升，线程切换概率下降，各种序列发生

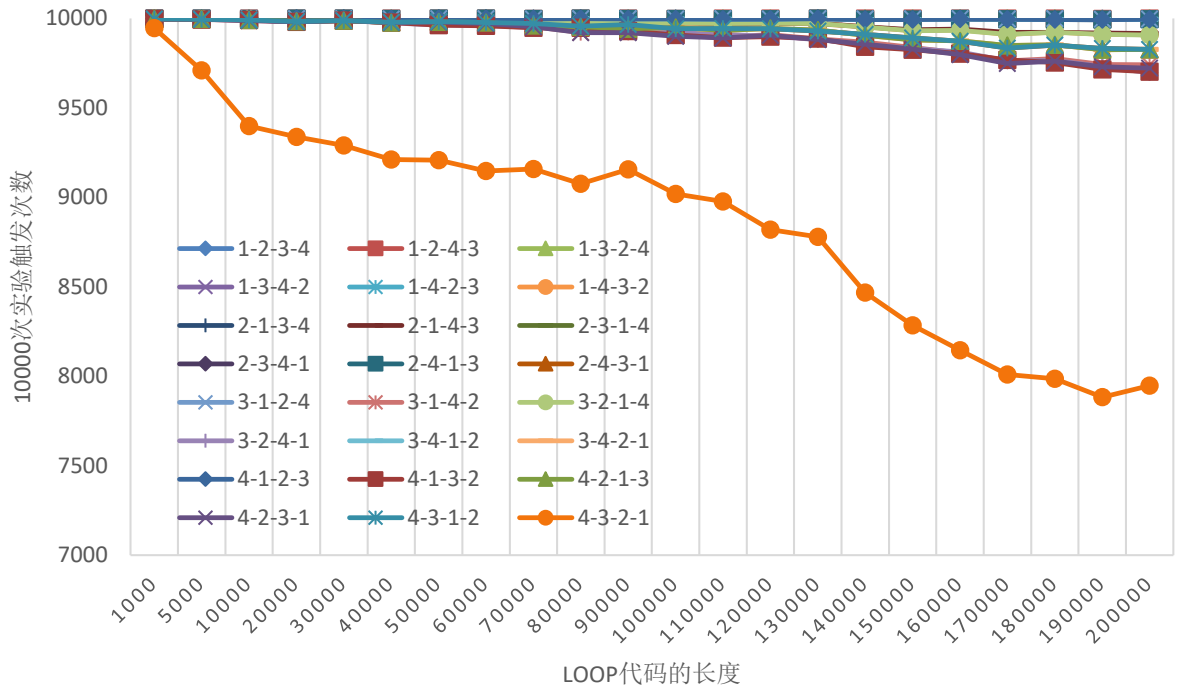


图 3-13　高负载下 *loop* 代码影响触发次数变化折线

概率上升。

操作系统的负载最直接的表现为 *CPU* 的占用比，但是由于操作系统本身会占用一定的负载，所以 *CPU* 负载无法为 0.00%。经过测试，一个完全空闲的 *Windows 7* 操作系统在占用 2.00% 的 *CPU*。本文通过执行无限循环程序提高 *CPU* 负载，从 0 个到 8 个共 9 种情况。其实际 *CPU* 负载分别为 2.00%、14.00%、27.00%、39.00%、52.00%、64.00%、77.00%、89.00%、100.00%。

在 $loop = 120k$ 时不同负载影响触发次数折线如图3-14所示，可以看出总体上随着负载的提高各种序列的触发次数也逐渐变多。但是部分序列在负载为 27.00% 时出现第一次下滑，在负载为 77.00% 出现第二次下滑。这些不寻常的下滑可以储备用作满足特殊的触发概率。

同样可以看出这些序列的变化有许多相同或者相似的地方，归类后依旧为表3-8所示的代表序列。所以按照之前分析选出的代表序列将图3-14所示 $loop = 120k$ 时不同负载影响触发次数折线简化如图3-15代表序列在 $loop = 120k$ 时不同负载影响触发次数折线。第一类代表序列为 1234，在所有负载情况下触发次数恒定为 10000。第二类代表序列为 1243，在负载为 2.00% 时触发次数为 7000 左右，在负载为 14.00% 时触发次数发生小幅度下降，在负载为 27.00% 时触发次数开始急速上升，然后在负载为 52.00% 时触发次数几乎达到顶峰。第三类代表序列为 2143，在负载为 2.00% 时触发次数为 5000 左右，在负载为 14.00% 时触发次数发生小幅度下降，在负载为 52.00% 时触发次数开始急速上升，在负载为 64.00% 时触发次数发生急速下降后在负载为 77.00% 时触

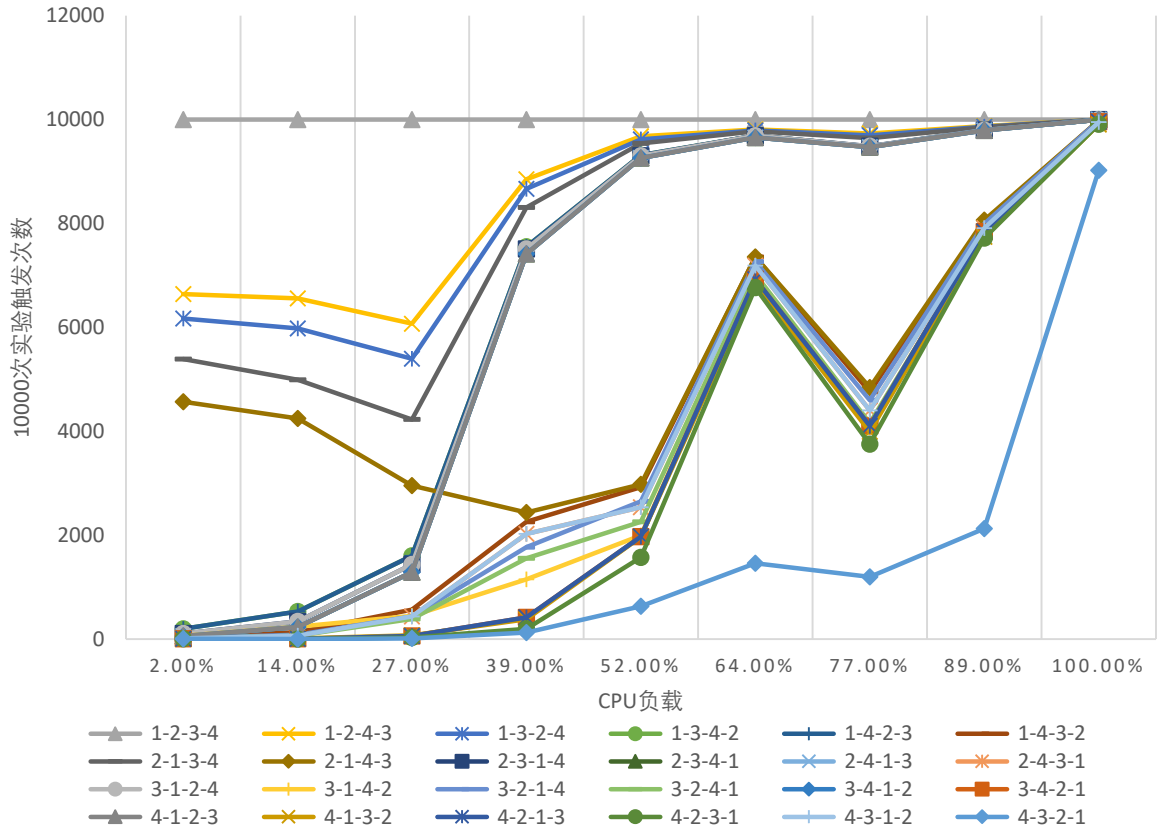


图 3-14　$loop = 120k$ 时不同负载影响触发次数折线

发次数又开始急速上升，最终在负载为 100.00% 时到达顶峰。第四类代表序列为 4321，在负载为 2.00% 时触发次数几乎为 0，在负载为 27.00% 时触发次数开始小幅度上升，然后在负载为 89.00% 时触发次数开始急速上升，最终在负载为 100.00% 时到达顶峰。第五类代表序列为 4123，在负载为 2.00% 时触发次数几乎为 0，在负载为 14.00% 时触发次数发生小幅度上升，在负载为 27.00% 时触发次数开始急速上升，然后在负载为 52.00% 时触发次数几乎达到顶峰。第六类代表序列为 3241，在负载为 2.00% 时触发次数几乎为 0，在负载为 14.00% 时触发次数发生小幅度上升，在负载为 52.00% 时触发次数开始急速上升，在负载为 64.00% 时触发次数发生急速下降后在负载为 77.00% 时触发次数又开始急速上升，最终在负载为 100.00% 时到达顶峰。

从这些代表序列的触发次数变化趋势可以看出其触发概率大小完全符合公式3-1所示规律，侧面印证了序列模式理论的正确性。除了概率大小之外，逆序度相同的代表序列变化发生的节点都比较相近，逆序的次数又会有规律的影响其发生概率的大小，并且其变化趋势也较为相近。

除去代表序列 1234，其他序列均有显著明显的特征，可以满足不同的概率需求。比如代表序列 4123 可以概括为低负载时触发概率较低，中高负载时触发概率较高。代表序列 4321 可以概括为低中负载时触发概率较低，高负载时触发概率较高。将这些序列进行取反操作又可以得到新的概率，最终组合出防不胜防的序列。

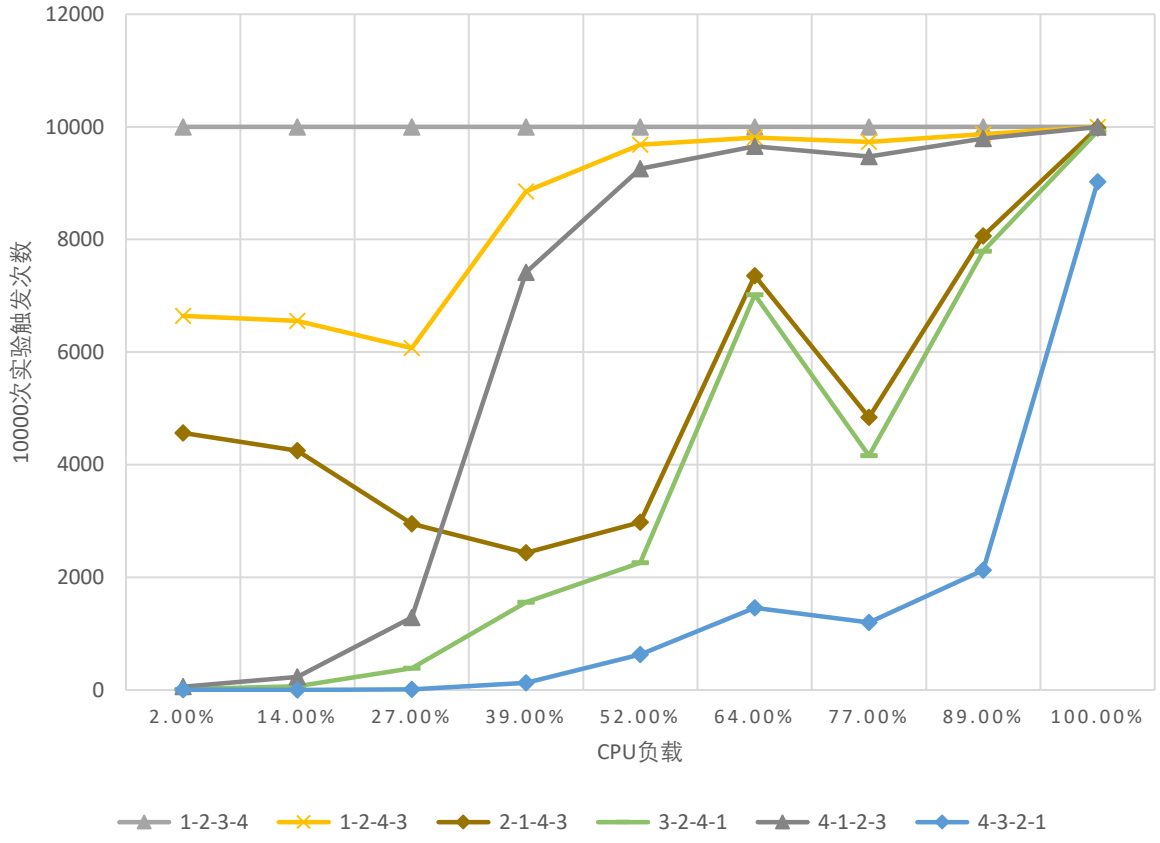


图 3-15　代表序列在 $loop = 120k$ 时不同负载影响触发次数折线

3）不同序列的概率

本节将对除 1234 外每种代表序列在不同的 *loop* 不同的负载下进行对比。通过对比可以扩展到其他环境下其他条件下其他种类的序列概率数据库。

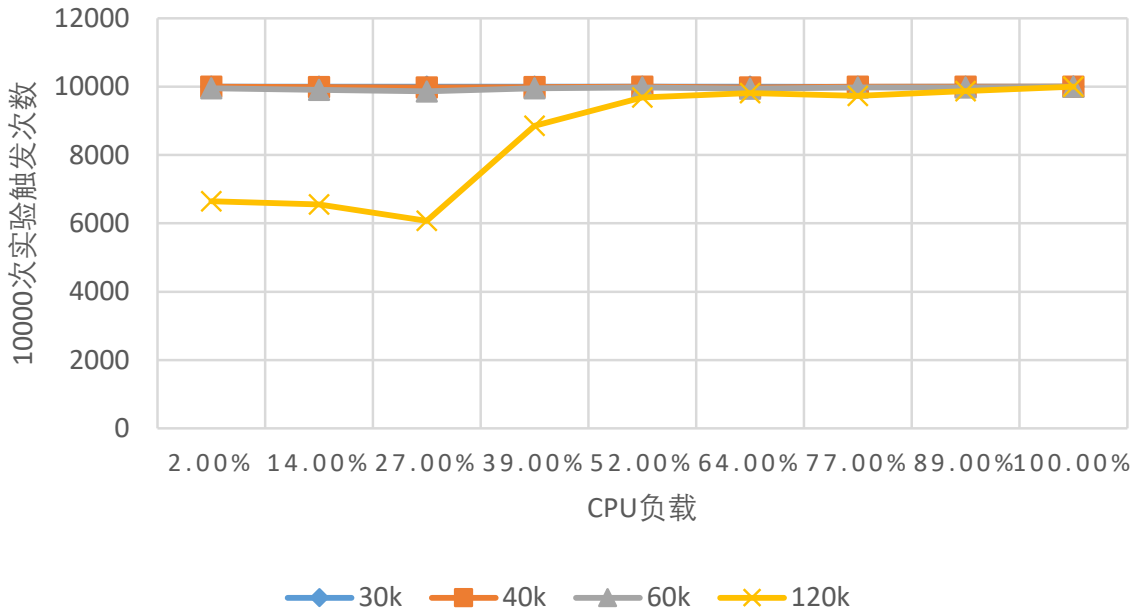


图 3-16　代表序列 1243 在不同 *loop* 下不同负载触发次数折线

代表序列 1243 和 2143 在不同 *loop* 下不同负载触发次数折线分别如图3-16和图3-

17所示。其中可以看出在 $loop = 120k$ 时与 $loop = 30k$、$loop = 40k$ 或者 $loop = 60k$ 时代表序列的触发次数有着完全不同的表现。在 $loop = 30k$、$loop = 40k$ 或者 $loop = 60k$ 时，代表序列的触发次数几乎没有任何变化，或者变化幅度非常小。所以，不同的 *loop* 值对各种序列的触发概率影响是十分明显的，在不同环境下需要重新测定合适的 *loop* 长度。对于并行逻辑炸弹而言，甚至可以采取自适应的方式动态的调整 *loop* 的长度以达到更好的隐藏效果。

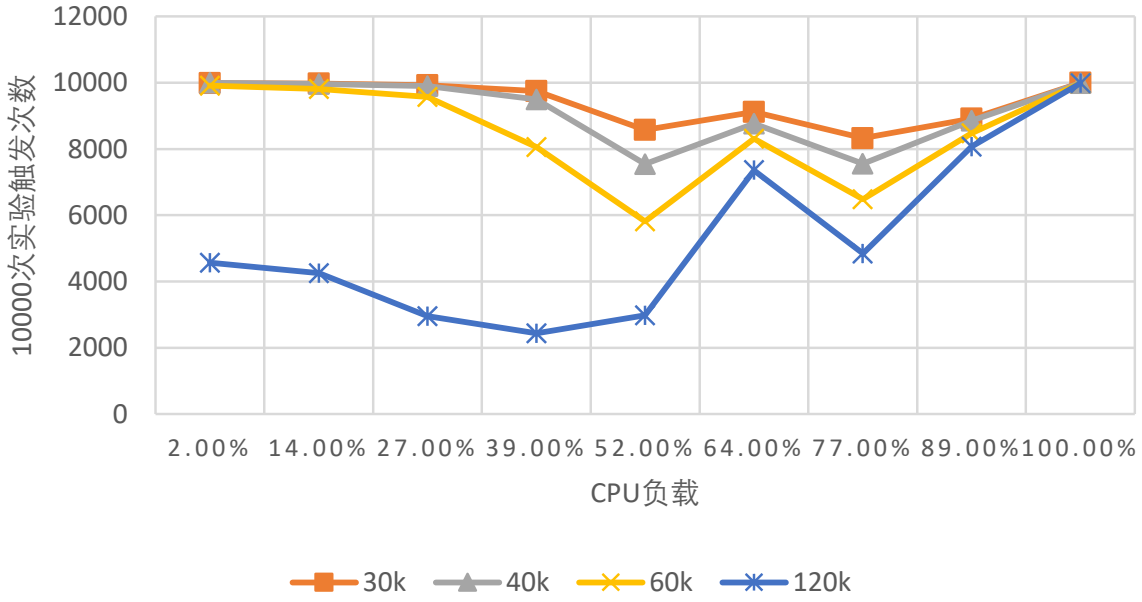


图 3-17　代表序列 2143 在不同 *loop* 下不同负载触发次数折线

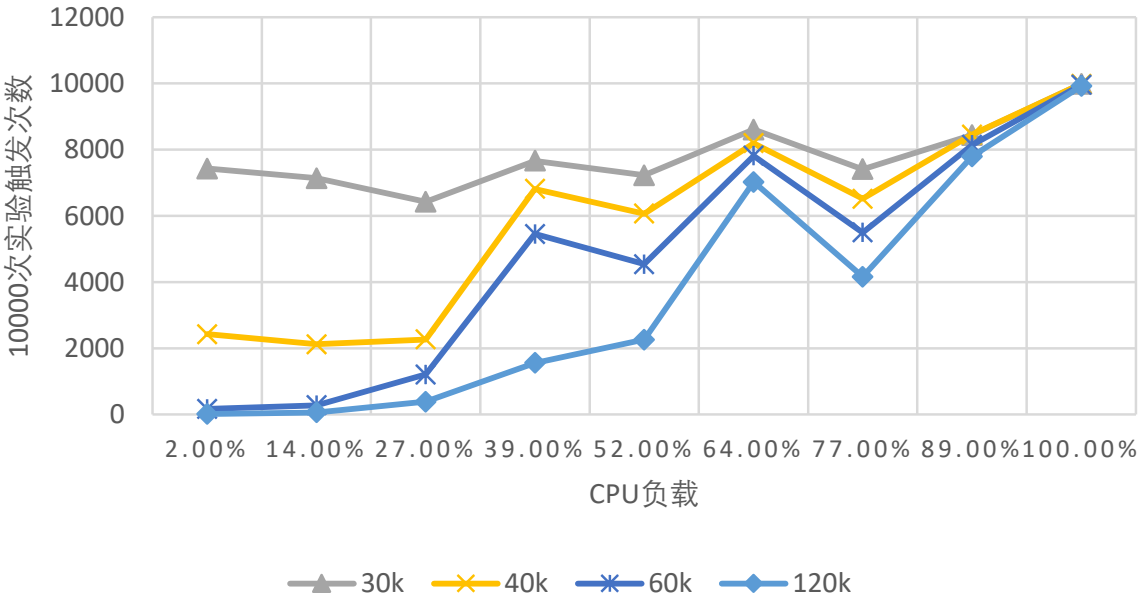


图 3-18　代表序列 3241 在不同 *loop* 下不同负载触发次数折线

代表序列 3241 和 4123、4321 在不同 *loop* 下不同负载触发次数折线分别如图3-18和图3-19、图3-20所示。这些代表序列在不同负载时触发概率有着巨大的差别，从负载

为 2.00% 的 0 到负载为 100.00% 的 10000。而且发生变化的负载节点也不相同，包括 27.00%、64.00%、89.00% 等等。所以不同的负载对各种序列的触发概率影响是十分明显的。由于负载的变化与 *CPU* 的逻辑核心数量相关，所以不同的 *CPU* 逻辑核心的情况下可能需要改变负载的设定值。对于并行逻辑炸弹而言，可以通过检测不同的 *CPU* 型号进行触发策略的调整，增加了隐蔽性。

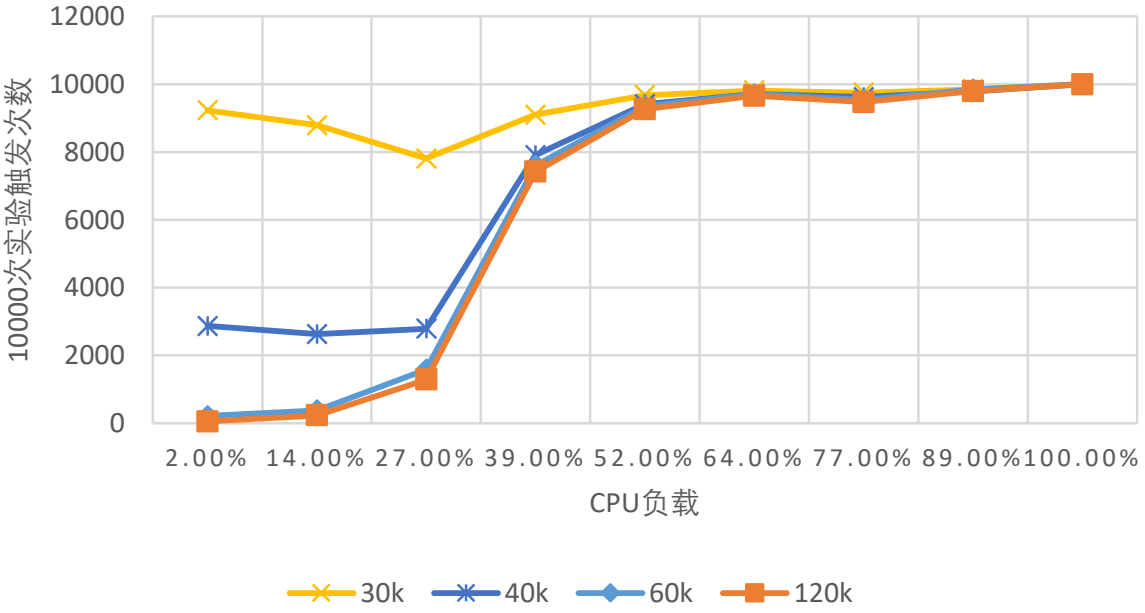


图 3-19　代表序列 4123 在不同 *loop* 下不同负载触发次数折线

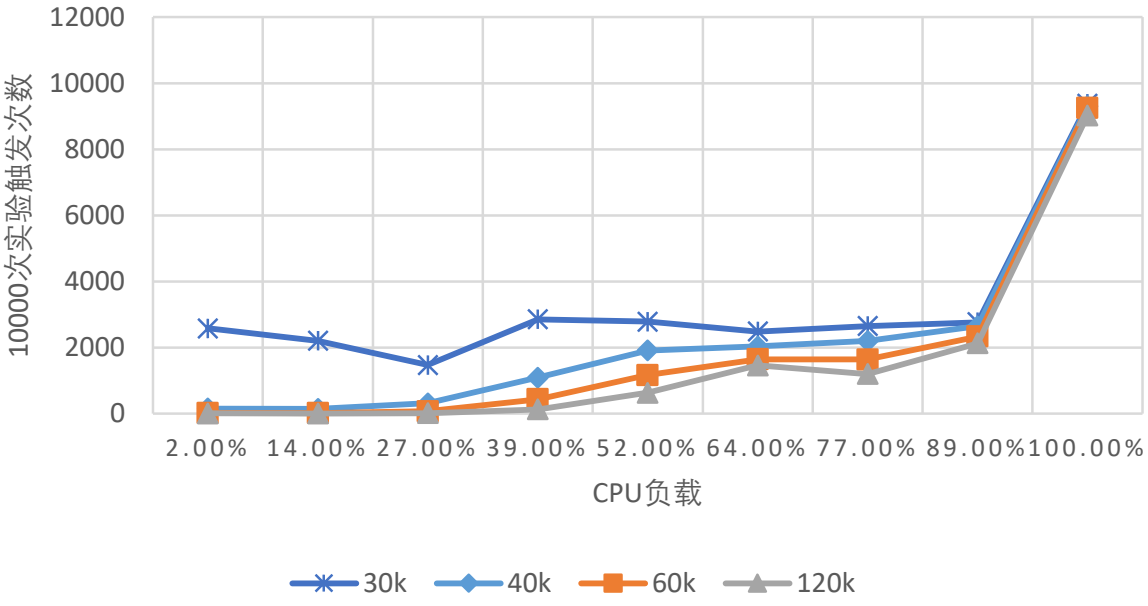


图 3-20　代表序列 4321 在不同 *loop* 下不同负载触发次数折线

除此之外，例如代表序列 4123 和代表序列 4321。代表序列 4321 为低中负载下触发概率低，高负载下触发概率高。用 !4321 表示序列 4321 不发生，选择 !4321 为触发策略，那么变为低中负载下触发概率高，高负载下触发概率低。代表序列 4123 为低负

载下触发概率低，中高负载下触发概率高。然后将序列 !4321 与序列 4123 发生组合，在这种触发策略下为低高负载下触发概率低，中负载下触发概率高。通过这些组合可以得到更多更隐蔽的触发策略，提高了并行逻辑炸弹的随机触发难度。

3.4 序列触发策略

根据并行逻辑炸弹的工作流程图3-1，本文已经得到了由恶意代码拆分好的代码片段、宿主程序中可供选择的插入点、各种序列在不同情况下触发概率的序列概率数据库和事先提出的触发概率需求。序列的触发策略选择步骤如下：

根据宿主程序中可供选择的插入点，从序列概率数据库中选出可供选择序列如果宿主程序中代码长度不够选定 *loop* 代表的长度，可以采用添加循环代码增加长度。根据触发概率需求，分别得到低中高三种负载下概率需求，或者在更加细的粒度上进行选择。根据触发概率高的负载要求，选出满足要求的主要序列，然后根据触发概率低的负载要求在具体选择合适的辅助序列。如果选出的序列长度小于拆分好的代码片段数目，适量添加不影响概率的补充序列，最终选定触发策略。

以触发概率需求为中负载下触发概率高，低负载和高负载下触发概率低为例说明。设定宿主程序为四线程，四个线程均执行相同的函数，恶意代码拆分为四段代码片段。

因为宿主程序为四个执行相同函数的线程，所以挑选出逆序次数小于四的序列供选择。根据触发概率需求，需要在中负载下触发概率高，选择序列 4123 作为主要序列。根据触发概率需求，需要在低负载和高负载下触发概率低，其中低负载下触发概率低已经满足，需要一个满足仅在高负载下触发概率低的序列，选择序列 !4321 作为辅助序列。目前选定的触发策略中序列序列长度大于等于代码片段数目，不需要添加补充序列。最终选择的触发策略为 !4321 + 4123。

3.5 本章总结

本章基于第2章关于目前并行软件设计和恶意代码检测方法的缺陷提出了一种新的恶意代码隐藏方法——并行逻辑炸弹。首先在理论上分析了并行逻辑炸弹的可行性，然后在实际中提出了切实可行的实施方案，并且给出了示例。详细的分析了多线程程序的线程调度策略以及提出了一种切实可行的利用方法。证明了第2章提出的问题不仅仅存在与理论上，在实际中确实存在着利用并行程序的恶意代码隐藏方法。

4 实验

本章用实际的例子证明了并行逻辑炸弹在面对静态恶意代码检测方法和动态恶意代码检测方法均有良好的隐蔽性。并且设计了一种远程控制负载触发并行逻辑炸弹的方案，证明了并行逻辑炸弹的危害性。

实验所有使用的程序源代码和实验结果截图等元数据均在https://github.com/ZHYfeng/malicious-code-conceal可以找到。

BullMoose[49] 是一个在研究中广使用的恶意代码，其主要恶意行为是将自身复制到系统盘，改变注册表使这个程序替代 *iexplore.exe*。本文选取 *BullMoose* 作为恶意代码样例因为其不仅广泛使用，而且其代码简洁、行为直观，易于解释和分析。

多线程宿主实验程序本文选取了 9 个来自著名的并行测试用例集 *SPLASH-2*[50] 的程序和 1 个微软开源的程序 *Multiverso*[51]。

多线程宿主实验程序的具体情况如表4-1所示：

表 4-1　多线程宿主实验程序实验程序

程序	代码行数	插入点
cholesky	5491	5
fft	1482	8
lu_c	1401	6
lu_nc	1182	6
ocean_c	5408	15
ocean_nc	3561	15
radix	1547	7
water_n	2593	7
water_s	3139	7
Multiverso	16254	1

Awesome Malware Analysis[52] 几乎罗列了所有互联网上现存的静态检测系统和动态检测系统。静态检测系统本文选择了广泛使用并且覆盖最全的 *VirusTotal*[53]。动态检测系统本文从中挑选出了可以正常使用并且具有广泛知名度的在线动态检测系统共计 8 个，分别是 *Malwr*[54]、*Payload*[55]、*Jevereg*[56]、*Comodo*[57]、*Falcon*[58]、*ThreatExpert*[59]、*Anlyz*[60]、*Secondwrite*[61]。

本章组织及内容如下：第4.1节静态检测，主要对并行逻辑炸弹进行静态检测。第4.2节动态检测，主要对并行逻辑炸弹进行动态检测。第4.3节远程触发，主要对并行逻辑炸弹进行远程触发。第4.4节本章总结，总结本章内容。

4.1 静态检测

VirusTotal[53] 是 2004 年创建，主要用来免费的分析文件和链接是否包含病毒、蠕虫、木马等等恶意代码。*VirusTotal* 目前包括 61 个每日更新病毒库并且配置好的杀毒软件，可以在线的扫描文件并且给出扫描结果。

表 4-2 恶意代码 BullMoose 在 VirusTotal 静态检测结果

杀毒软件	更新日期	结果
Ad-Aware	2017.06.09	Trojan.GenericKD.5005639
AegisLab	2017.06.09	Generic38.Bmdy!c
AhnLab-V3	2017.06.09	0
Alibaba	2017.06.09	-
ALYac	2017.06.09	Trojan.GenericKD.5005639
Antiy-AVL	2017.06.09	Trojan/Win32.TSGeneric
Arcabit	2017.06.09	Trojan.Generic.D4C6147
Avast	2017.06.09	Win32:Malware-gen
AVG	2017.06.09	Win32:Malware-gen
Avira (no cloud)	2017.06.09	TR/AD.Malex.nbrdi
AVware	2017.06.09	Trojan.Win32.Generic!BT
Baidu	2017.06.08	Win32.Trojan.WisdomEyes.16070401.9500.9999
BitDefender	2017.06.09	Trojan.GenericKD.5005639
Bkav	2017.06.09	W32.Clod9d8.Trojan.fee8
CAT-QuickHeal	2017.06.09	Trojan.Malexf
ClamAV	2017.06.09	0
CMC	2017.06.09	0
Comodo	2017.06.09	UnclassifiedMalware
CrowdStrike Falcon (ML)	2017.04.20	0
Cyren	2017.06.09	W32/Trojan.OLWA-3259
DrWeb	2017.06.09	Trojan.Siggen7.22061
Emsisoft	2017.06.09	Trojan.GenericKD.5005639 (B)
Endgame	2017.05.15	0
eScan	2017.06.09	Trojan.GenericKD.5005639
ESET-NOD32	2017.06.09	a variant of Generik.FKZGLVQ
F-Prot	2017.06.09	0
F-Secure	2017.06.09	Trojan.GenericKD.5005639
Fortinet	2017.06.09	Generik.FKZGLVQ!tr
GData	2017.06.09	Trojan.GenericKD.5005639
Ikarus	2017.06.09	Trojan.Win32.Malex
Jiangmin	2017.06.09	Trojan.Cometer.aq
K7AntiVirus	2017.06.09	Riskware (0040eff71)
K7GW	2017.06.09	Riskware (0040eff71)
Kaspersky	2017.06.09	UDS:DangerousObject.Multi.Generic
Kingsoft	2017.06.09	0
Malwarebytes	2017.06.09	0
McAfee	2017.06.09	Artemis!74D8EF888925
McAfee-GW-Edition	2017.06.08	BehavesLike.Win32.BadFile.lm
Microsoft	2017.06.09	Trojan:Win32/Malex.gen!F
NANO-Antivirus	2017.06.09	Trojan.Win32.AD.eoiqyz
nProtect	2017.06.09	0
Palo Alto Networks (Known Signatures)	2017.06.09	0
Panda	2017.06.09	Trj/GdSda.A
Qihoo-360	2017.06.09	0
Rising	2017.06.09	0
SentinelOne (Static ML)	2017.05.16	0
Sophos AV	2017.06.09	Mal/Generic-S
Sophos ML	2017.06.07	0
SUPERAntiSpyware	2017.06.09	0
Symantec	2017.06.09	Trojan.Gen
Symantec Mobile Insight	2017.06.08	-
Tencent	2017.06.09	Win32.Trojan.Generic.Ajca
TheHacker	2017.06.07	0
TotalDefense	2017.06.09	0
TrendMicro	2017.06.09	TROJ_GEN.R0EDC0DEA17
TrendMicro-HouseCall	2017.06.09	TROJ_GEN.R0EDC0DEA17
Trustlook	2017.06.09	-
VBA32	2017.06.09	0
VIPRE	2017.06.09	Trojan.Win32.Generic!BT
ViRobot	2017.06.09	Trojan.Win32.Z.Malex.19968[h]
Webroot	2017.06.09	0
WhiteArmor	2017.06.08	-
Yandex	2017.06.08	Trojan.Agent!FXhD/GsyCbY
Zillya	2017.06.08	0
ZoneAlarm by Check Point	2017.06.09	UDS:DangerousObject.Multi.Generic
Zoner	2017.06.09	0

这些杀毒软件和更新日期如表4-2所示，包括广泛使用的 *Avast*、*McAfee*、*Microsoft* 等等。除了极个别杀毒软件外，大部分均更新到了 *2017.06.09* 版本。

表 4-3　并行逻辑炸弹在 VirusTotal 静态检测结果

杀毒软件	cholesky	fft	lu_c	lu_nc	ocean_c	ocean_nc	radix	water_n	water_s	Multiverso
Ad-Aware	0	0	0	0	0	0	0	0	0	0
AegisLab	0	0	0	0	0	0	0	0	0	0
AhnLab-V3	0	0	0	0	0	0	0	0	0	0
Alibaba	-	-	-	-	-	-	-	-	-	-
ALYac	0	0	0	0	0	0	0	0	0	0
Antiy-AVL	0	0	0	0	0	0	0	0	0	0
Arcabit	0	0	0	0	0	0	0	0	0	0
Avast	0	0	0	0	0	0	0	0	0	0
AVG	0	0	0	0	0	0	0	0	0	0
Avira (no cloud)	0	0	0	0	0	0	0	0	0	0
AVware	0	0	0	0	0	0	0	0	0	0
Baidu	0	0	0	0	0	0	0	0	0	0
BitDefender	0	0	0	0	0	0	0	0	0	0
Bkav	0	0	0	0	0	0	0	0	0	0
CAT-QuickHeal	0	0	0	0	0	0	0	0	0	0
ClamAV	0	0	0	0	0	0	0	0	0	0
CMC	0	0	0	0	0	0	0	0	0	0
Comodo	0	0	0	0	0	0	0	0	0	0
CrowdStrike Falcon (ML)	0	0	0	0	0	0	0	0	0	0
Cyren	0	0	0	0	0	0	0	0	0	0
DrWeb	0	0	0	0	0	0	0	0	0	0
Emsisoft	0	0	0	0	0	0	0	0	0	0
Endgame	0	0	0	0	0	0	0	0	0	0
eScan	0	0	0	0	0	0	0	0	0	0
ESET-NOD32	0	0	0	0	0	0	0	0	0	0
F-Prot	0	0	0	0	0	0	0	0	0	0
F-Secure	0	0	0	0	0	0	0	0	0	0
Fortinet	0	0	0	0	0	0	0	0	0	0
GData	0	0	0	0	0	0	0	0	0	0
Ikarus	0	0	0	0	0	0	0	0	0	0
Jiangmin	0	0	0	0	0	0	0	0	0	0
K7AntiVirus	0	0	0	0	0	0	0	0	0	0
K7GW	0	0	0	0	0	0	0	0	0	0
Kaspersky	0	0	0	0	0	0	0	0	0	0
Kingsoft	0	0	0	0	0	0	0	0	0	0
Malwarebytes	0	0	0	0	0	0	0	0	0	0
McAfee	0	0	0	0	0	0	0	0	0	0
McAfee-GW-Edition	0	0	0	0	0	0	0	0	0	0
Microsoft	0	0	0	0	0	0	0	0	0	0
NANO-Antivirus	0	0	0	0	0	0	0	0	0	0
nProtect	0	0	0	0	0	0	0	0	0	0
Palo Alto Networks (Known Signatures)	0	0	0	0	0	0	0	0	0	0
Panda	0	0	0	0	0	0	0	0	0	0
Qihoo-360	0	0	0	0	0	0	0	0	0	0
Rising	0	0	0	0	0	0	0	0	0	0
SentinelOne (Static ML)	0	0	0	0	0	0	0	0	0	0
Sophos AV	0	0	0	0	0	0	0	0	0	0
Sophos ML	0	0	0	0	0	0	0	0	0	0
SUPERAntiSpyware	0	0	0	0	0	0	0	0	0	0
Symantec	0	0	0	0	0	0	0	0	0	0
Symantec Mobile Insight	-	-	-	-	-	-	-	-	-	-
Tencent	0	0	0	0	0	0	0	0	0	0
TheHacker	0	0	0	0	0	0	0	0	0	0
TotalDefense	0	0	0	0	0	0	0	0	0	0
TrendMicro	0	0	0	0	0	0	0	0	0	0
TrendMicro-HouseCall	0	0	0	0	0	0	0	0	0	0
Trustlook	-	-	-	-	-	-	-	-	-	-
VBA32	0	0	0	0	0	0	0	0	0	0
VIPRE	0	0	0	0	0	0	0	0	0	0
ViRobot	0	0	0	0	0	0	0	0	0	0
Webroot	0	0	0	0	0	0	0	0	0	0
WhiteArmor	-	-	-	-	-	-	-	-	-	-
Yandex	0	0	0	0	0	0	0	0	0	0
Zillya	0	0	0	0	0	0	0	0	0	0
ZoneAlarm by Check Point	0	0	0	0	0	0	0	0	0	0
Zoner	0	0	0	0	0	0	0	0	0	0

将原始的恶意代码 *BullMoose* 上传检测结果如表4-2所示①，61 个杀毒软件中有 41 个检测出恶意代码。

将恶意代码 *BullMoose* 使用并行逻辑炸弹的方式分别隐藏在 10 个宿主程序中，触发策略选择为 4321，其结果如表4-3所示。并行逻辑炸弹成功的躲避了所有的杀毒软件，可以看出并行逻辑炸弹对静态检测有着很好的隐藏效果。

4.2 动态检测

根据第2.2节的说明，相比于杀毒软件通常使用的静态检测方法，动态检测可以更加有效的检测恶意代码。本文选择了 8 个动态检测系统，分别是 *Malwr*[54]、*Payload*[55]、*Jevereg*[56]、*Comodo*[57]、*Falcon*[58]、*ThreatExpert*[59]、*Anlyz*[60]、*Secondwrite*[61]。这些动态检测系统基于一些著名的沙箱，比如 *Cuckoo*、*VxStream*、*Amnpardaz*、*Falcon* 等[62–65]。

动态检测可以检测到所有执行过的行为，想要躲避动态检测只有在检测时不运行恶意代码。传统的隐藏方法通过检测自身是否被检测进而执行不同的路径，但是动态检测可以检测到恶意代码检测自身是否被检测这一过程。

将恶意代码 *BullMoose* 使用并行逻辑炸弹的方式分别隐藏在 10 个宿主程序中，在每个插入点触发策略选择为 3241、4123 和 4321，样本数量如表4-4所示。八个动态检测系统对并行逻辑炸弹的检测结果如表4-4所示。

表 4-4 并行逻辑炸弹动态检测结果

实验样本	样本个数	Malwr	Payload	Jevereg	Comodo	Falcon	ThreatExpert	Anlyz	Secondwrite
BullMoose	1	1	1	1	1	1	1	1	1
cholesky	15	0	0	0	0	0	0	0	0
fft	24	0	0	18	0	0	0	0	1
lu_c	18	0	0	16	0	0	0	0	0
lu_nc	18	0	0	17	0	0	0	0	1
ocean_c	45	0	0	41	0	0	0	0	0
ocean_nc	45	0	0	41	0	0	0	0	1
radix	21	0	0	17	0	0	0	0	0
water_n	21	0	0	0	0	0	0	0	0
water_s	21	0	0	0	0	0	0	0	0
Multiverso	3	0	0	0	0	0	0	0	0

可以看出并行逻辑炸弹成功的躲避了其中六个动态检测系统，未能完全躲避 *Jevereg* 和 *Secondwrite* 的检测。

由于采用的触发策略 3241、4123 和 4321 均可以在高负载下大概率触发，而在线动态检测系统有一定可能性长期处于高负载状态，所以可能这两种动态检测系统恰巧负载较高而检测到并行逻辑炸弹。为了验证究竟是这两种动态检测系统可以主动的探

① 原始结果地址https://www.virustotal.com/en/file/3fb9d76fdbf1426d70c98256d5421abb56fc18aa1299e4734cf069047cdcc5ce/analysis/

索检测到并行逻辑炸弹还是仅仅因为被动的恰巧负载较高而检测到高负载触发策略并行逻辑炸弹，我在 10 个宿主程序中采用 !4321 + 4123 的触发策略生成新的并行逻辑炸弹重新进行检测，检测结果如表4-5。可以看出这两种动态检测系统完全不能检测出中负载触发策略生成的并行逻辑炸弹。中负载触发策略生成的并行逻辑炸弹需要负载不高不低才能大概率触发恶意代码，高负载和低负载下触发概率均很低。所以根据对比实验，可以确定这两种动态检测系统仅仅是因为被动的恰巧负载较高而检测到高负载触发策略并行逻辑炸弹。所以，并行逻辑炸弹对现有的动态检测系统也有很好的隐蔽作用。

表 4-5　中负载触发策略并行逻辑炸弹动态检测结果

Malware	Jevereg	Secondwrite
cholesky	0	0
fft	0	0
lu_c	0	0
lu_nc	0	0
ocean_c	0	0
ocean_nc	0	0
radix	0	0
water_n	0	0
water_s	0	0
Multiverso	0	0

4.3　远程触发

前面的实验已经证明并行逻辑炸弹可以通过现有的静态检测和动态检测，那么并行逻辑炸弹可以认为完全的隐藏在可信任的软件中。本实验将证明一个可信任的包括并行逻辑炸弹的软件在远程目标机器上运行在一定的条件下可以被主动触发。

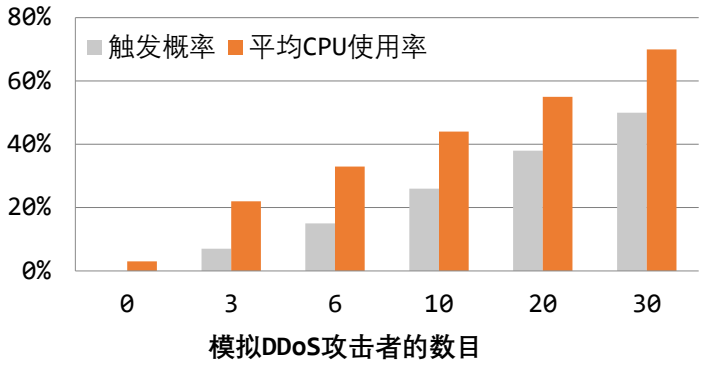


图 4-1　模拟远程攻击触发并行逻辑炸弹

Red5[66] 可以便捷的在主机上搭载一个可供访问的媒体服务，这里使用的 *Red5* 注入了通过并行逻辑炸弹隐藏的恶意代码，其触发策略为 4123，所以在高负载下可以触发。将 *Red5* 布置在一台实验主机上，具体配置为处理器 *Intel(R)Core(TM)i5-3470*，内存 *16GB*。

Goldeneye[67] 是一个可以模拟 DDos 攻击的开源工具，通过模拟不同数量的攻击者达到不同强度攻击的目的。在另一台机器上使用 *Goldeneye* 分别使用不同数目的攻

击者攻击配置好 *Red5* 主机。

最终实验结果如图4-1所示：

可以看出随着模拟攻击者数目的增加，目标主机因为需要处理 DDos 攻击，其 CPU 负载也在不断地增加，最终并行逻辑炸弹的触发概率也不断地增加。

4.4 本章总结

本章分别对并行逻辑炸弹进行了静态检测，动态检测，远程触发。证明了并行逻辑炸弹面对静态检测有着很好的隐藏能力，面对动态检测采取合适的触发策略也可以达到很好的隐藏效果，同时给出了一个可以主动触发并行逻辑炸弹的方法。证明现有的恶意代码检测方法对并行逻辑炸弹的无效，在面对并行攻击时需要改进，同时也证明了并行逻辑炸弹的危害。

5 讨论和防御

5.1 局限

显然，想要通过现有动态恶意代码检测方法来检测和防御并行逻辑炸弹是非常困难的。然而，并行逻辑炸弹虽然有着非常强的隐藏能力，但同时也存在着一些缺陷：

并行逻辑炸弹只能在并行程序中隐藏恶意代码。并行程序相对于串行程序的优点就是可以在多核处理器上极大的提高程序的效率。虽然并行程序越来越普遍，但是在某些情况下串行程序仍然发挥着作用，因此并行逻辑炸弹适用范围在一定程度上受限。

并行逻辑炸弹利用了并行程序的不确定性。并行程序与串行程序最大的不同就在于不确定性，而并行逻辑炸弹完全基于这种不确定性。如果不能保证并行程序的不确定性，那么并行逻辑炸弹也将不存在。

并行逻辑炸弹通过不同的处理器负载触发隐藏的恶意代码。在不同的处理器负载下，并行逻辑炸弹触发概率不同。如果恰巧在高触发概率的处理器负载下进行恶意代码的检测，那么很有可能将并行逻辑炸弹隐藏的恶意代码检测出来。

并行逻辑炸弹隐藏和触发均按照概率发生。并行逻辑炸弹无论是隐藏或者触发均基于概率，也就是无法保证在需要隐藏的时候一定隐藏，也无法保证在需要触发时一定触发。如果长期隐藏多次触发的情况下，并行逻辑炸弹会按照概率隐藏或者触发。而如果要求每次精确的触发则无法做到。

5.2 防御

2000 年 *Gary McGraw* 提出了面对恶意程序时的一般性防御措施[68]：

对程序进行静态分析，如果执行程序可能导致危害则采取措施。

重写程序，保证程序来源的可靠性。

运行时监控程序，在程序将要造成危害时采取措施。

对程序进行动态分析，如果检测到程序造成了危害则采取措施。

这些方法在现在依旧适用，目前分析恶意程序的方法为静态分析和动态分析，尽管静态分析目前并不是很有效。重写程序可以理解为现在的安全认证，通过来源的可靠性保证程序的安全。监控程序则是一种非常有效地方法，只要监控系统处于正常的运行状态，恶意程序就不会造成任何危害。然而监控系统对系统性能造成的损耗成本非常高，在安全和效率两者之间需要衡量。

结合并行逻辑炸弹的局限性和以上一般性防御措施，提出了如下防御并行逻辑炸弹的方法：

并行程序串行化。因为并行逻辑炸弹需要并行程序才能隐藏恶意代码，并行程序串行化后并行逻辑炸弹将完全失效，这种方法的缺点是效率的降低。并行程序的出现

就是因为并行程序的高效，可以更加合理的利用系统资源。在某些安全需求大于效率需求时，为了确保安全放弃一部分效率，并行程序串行化可以作为一种防御并行逻辑炸弹的有效方法。

并行程序确定化。因为并行逻辑炸弹主要利用并行程序的不同时序产生的路径进行恶意代码的隐藏，将并行程序不确定的路径固定为每次执行确定的路径。那么只需要检测一条确定的路径并且保证没有恶意行为，每次按照这条检测过的确定的路径执行并行程序，就可以保证安全性[69]。这种方法的缺点是在不同的输入或者其他外部条件下，需要重新检测一条新的路径，而每次检测一条新的路径是非常低效的。在程序输入变化较小时，可以采取这种方式防御并行逻辑炸弹。

动态检测时覆盖更多处理器负载并且增加执行次数。因为在某些处理器负载下，并行逻辑炸弹的触发概率较高。在动态检测时覆盖尽可能多的处理器负载可以更大概率的检测到并行逻辑炸弹隐藏的恶意代码。同样，因为并行逻辑炸弹的触发是基于概率的，所以只要执行的次数足够多，再小的概率仍然会触发。通过提高动态执行的次数，将会从统计学上提高检测到并行逻辑炸弹隐藏的恶意代码的概率。这种方法的缺点一方面是会造成效率极大地降低，另一方面是只能针对并行逻辑炸弹着一种并行攻击。但这种方法也是目前最容易实施的方法。

安全认证。安全认证是面对任何安全问题都有效的一种方法，并行逻辑炸弹也不例外。如果从根源上保证程序是安全的，并且没有被其他人改变过，那么就认为程序是安全。

6 总结与展望

6.1 工作总结

随着并行程序的普遍使用，各种并行攻击逐渐增多。由于并行程序自身的复杂性，防御并行攻击本身就是十分困难的。现有的恶意代码检测方法面对部分并行攻击时是完全失效的，更令人担忧的是防御并行攻击还没得到足够的重视。

为了引起对并行攻击的重视，本文分析了现有恶意代码检测方法和并行程序之间的一个缺失。并行程序有着复杂不确定的时序交织并且会产生数量巨大的路径，而现有的恶意代码检测方法在检测并行程序时却没有相应的措施。

基于此本文提出了一种并行恶意代码隐藏方法——并行逻辑炸弹，并行逻辑炸弹成功的躲避了现有的静态检测方法和动态检测方法。对现有恶意代码检测方法在并行程序中的使用提出了根本上的不同需求。

本文的主要贡献如下：

并行程序比串行程序更加复杂的根本原因是不确定的时序交织，这些不确定的时序交织会产生指数级增长的路径。本文分析了现有得恶意代码检测方法在面对并行程序不确定的时序交织时的不足，现有恶意代码检测方法是无法检测到发生在时序交织中的恶意行为。基于现有恶意代码检测方法在面对并行程序时的缺失，提出了并行逻辑炸弹这一并行恶意代码隐藏方法。

尽管之前也有恶意代码拆分的需求，却没有完整可用的恶意代码拆分算法。为了满足并行性逻辑炸弹需要将恶意代码拆分的需求，本文首次明确提出了一种恶意代码拆分算法，将恶意代码拆分为多段单独无害的代码片段。

并行程序的线程调度是不确定的，本文总结了并行程序中不同类型函数的交织情况，并且分析了锁和负载对线程调度的影响。得到主要结论，低负载下锁之间的间隔代码长度主要影响线程调度，间隔代码越长，线程切换概率越大；高负载下负载主要影响线程调度，负载越高，线程切换概率越小。

为了更好的分析不同序列在不同交织的概率，提出了序列模式理论对序列从概率上进行表示和分类。统计了不同模式的序列在不同情况下的触发概率，验证了序列模式理论的有效性，相同模式序列在概率上的一致性。根据不同模式的概率提出了并行逻辑炸弹的序列触发策略。

使用真实的恶意代码通过静态检测实验和动态检测实验证明了并行逻辑炸弹的有效性和隐蔽性。同时设计了一种远程触发并行逻辑炸弹的方案，并且在模拟真实的环境下远程触发了真实的恶意代码证明了其危害性。

总结了并行逻辑炸弹的不足，根据一般性的防御措施和并行逻辑炸弹独有的特定设想了在实际中可行的防御方法。

6.2 下一步工作

目前版本的并行逻辑炸弹已经有足够的隐蔽性和危害性，会引起对并行攻击更多的重视。然而还有如下三个方面可以加强：

现有的并行逻辑炸弹序列概率数据库仅仅统计了四线程以下的概率，更多线程数目的概率需要进一步统计。除此之外，负载仅仅是影响线程调度的一个因素，更多影响线程调度的因素有待挖掘，并且可以基于此提出并行逻辑炸弹的变种，基于负载的并行逻辑炸弹只是一个开始。

正如基于负载的并行逻辑炸弹只是并行逻辑炸弹的一个开始，并行逻辑炸弹也只是并行攻击的一个开始。并行程序的复杂性的根源在于不确定性，由此导致了复杂的时序交织进一步造成了并行逻辑炸弹的出现。除此之外，并行攻击理论上存在更多的形式，这一切都是需要预先防范的。

尽管本文提出了一些实际可行的防御方法，但是事实上并没有从根本上解决并行逻辑炸弹，解决并行攻击。文本最主要的目的就是提高对并行攻击的重视，提出从根本上解决并行程序的安全问题的措施。未来最重要的工作就是从根本上对并行攻击进行防御。

致　　谢

首先感谢我的指导教师刘烃老师，在我三年的硕士研究生学习阶段，始终对我抱有非常巨大的期望，不断地激励我，并且指引着我在学术道路上的前行。平时经常指导我的工作，更是在我犯错时及时指出，以身作则的告诉我一个科学研究人员是如何刻苦拼搏的。也告诉我科学研究不能闭门造车，要与时俱进，为我们创造交流学习的机会。除此之外，刘老师非常关心我们的身体健康和心理健康，在我们研究遇到挫折时鼓励我们，开导我们。我完成本论文的过程中，刘老师在我迷茫的时候指导着我，在我灰心的时候激励着我，在我懈怠的时候督促着我，非常衷心的感谢我的指导教师刘老师。

感谢国家自然科学基金，没有国家自然科学基金的支持我是不可能完成本论文的。

感谢郑庆华老师，在郑老师的带领下，实验室为全体同学提供了良好的科研环境，为科研工作提供了良好的物质基础和组织基础，并且郑老师在百忙之中抽出时间给参加每周实验室组会，指出同学们工作中的不足。

感谢整个可信小组，正是在这样富有科研激情的团队我才能完成论文。感谢张晓东师兄、樊浩涵师兄、刘沛师兄的指导，教会我很多知识和科研的技巧。感谢郭嘉琪师弟、崔迪、尹文浩师弟在工作完成过程中的帮助。

感谢扬子江老师、蔡彦老师、陈凯老师、陈浩老师在完成工作过程中的指导和非常有建设意义的意见。

最后感谢参与论文评审和答辩的老师。

参考文献

[1] wikipedia. Sunway TaihuLight[EB/OL]. 2018[March]. https://en.wikipedia.org/wiki/Sunway_TaihuLight.

[2] Lu S, Park S, Seo E, et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics[C/OL] // Eggers S J, Larus J R. Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008. [S.l.]: ACM, 2008: 329–339. http://doi.acm.org/10.1145/1346281.1346323.

[3] Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs[C/OL] // Ferrante J, Mckinley K S. Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. [S.l.]: ACM, 2007: 446–455. http://doi.acm.org/10.1145/1250734.1250785.

[4] Taylor R N, Levine D L, Kelly C D. Structural Testing of Concurrent Programs[J/OL]. IEEE Trans. Software Eng., 1992, 18(3): 206–215. https://doi.org/10.1109/32.126769.

[5] Adl-tabatabai A, Kozyrakis C, Saha B. Transactional programming in a multi-core environment[C/OL] // Yelick K A, Mellor-crummey J M. Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007. [S.l.]: ACM, 2007: 272. http://doi.acm.org/10.1145/1229428.1229484.

[6] Yang J, Cui A, Stolfo S J, et al. Concurrency Attacks[C/OL] // Boehm H, Ceze L. 4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012. [S.l.]: USENIX Association, 2012. https://www.usenix.org/conference/hotpar12/workshop-program/presentation/yang.

[7] Özkan S. Common vulnerabilities and exposures database[EB/OL]. 2018[March]. https://cvedetails.com.

[8] Cve. CVE-2004-1235[EB/OL]. 2018[March]. http://www.cvedetails.com/cve/CVE-2004-1235.

[9] Starzetz P. Linux kernel uselib() privilege elevation[EB/OL]. 2018[March]. https://isec.pl/en/vulnerabilities/isec-0021-uselib.txt.

[10] Hat R. RHBA-2009:1634-1[EB/OL]. 2018[March]. https://access.redhat.com/errata/RHBA-2009:1634.

[11] Cve. CVE-2011-0990[EB/OL]. 2018[March]. https://www.cvedetails.com/cve/CVE-2011-0990.

[12] Chandra S, Chen P M. Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software[C/OL] // 2000 International Conference on Dependable Systems and Networks (DSN 2000) (formerly FTCS-30 and DCCA-8), 25-28 June 2000, New York, NY, USA. [S.l.]: IEEE Computer Society, 2000: 97–106. https://doi.org/10.1109/ICDSN.2000.857521.

[13] Engler D R, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks[C] // SOSP. [S.l.]: ACM, 2003: 237–252.

[14] Prvulovic M, Torrellas J. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes[C/OL] // Gottlieb A, Li K. 30th International Symposium on Computer Archi-

tecture (ISCA 2003), 9-11 June 2003, San Diego, California, USA. [S.l.] : IEEE Computer Society, 2003 : 110 – 121.
https://doi.org/10.1109/ISCA.2003.1206993.

[15] Boyapati C, Lee R, Rinard M C. Ownership types for safe programming: preventing data races and deadlocks[C/OL] // Ibrahim M, Matsuoka S. Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002.. [S.l.] : ACM, 2002 : 211 – 230.
http://doi.acm.org/10.1145/582419.582440.

[16] Flanagan C, Freund S N. Atomizer: A dynamic atomicity checker for multithreaded programs[J/OL]. Sci. Comput. Program., 2008, 71(2) : 89 – 109.
https://doi.org/10.1016/j.scico.2007.12.001.

[17] Lu S, Tucek J, Qin F, et al. AVIO: detecting atomicity violations via access interleaving invariants[C/OL] // Shen J P, Martonosi M. Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006. [S.l.] : ACM, 2006 : 37 – 48.
http://doi.acm.org/10.1145/1168857.1168864.

[18] Xu M, Bodík R, Hill M D. A serializability violation detector for shared-memory server programs[C/OL] // Sarkar V, Hall M W. Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005. [S.l.] : ACM, 2005 : 1 – 14.
http://doi.acm.org/10.1145/1065010.1065013.

[19] Qadeer S, Wu D. KISS: keep it simple and sequential[C/OL] // Pugh W, Chambers C. Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004. [S.l.] : ACM, 2004 : 14 – 24.
http://doi.acm.org/10.1145/996841.996845.

[20] Ramilli M, Bishop M. Multi-stage delivery of malware[C/OL] // 5th International Conference on Malicious and Unwanted Software, MALWARE 2010, Nancy, France, October 19-20, 2010. [S.l.] : IEEE, 2010 : 91 – 97.
https://doi.org/10.1109/MALWARE.2010.5665788.

[21] Ramilli M, Bishop M, Sun S. Multiprocess malware[C/OL] // 6th International Conference on Malicious and Unwanted Software, MALWARE 2011, Fajardo, Puerto Rico, USA, October 18-19, 2011. [S.l.] : IEEE Computer Society, 2011 : 8 – 13.
https://doi.org/10.1109/MALWARE.2011.6112320.

[22] Fan L, Wang Y, Cheng X, et al. Privacy theft malware multi-process collaboration analysis[J/OL]. Security and Communication Networks, 2015, 8(1) : 51 – 67.
https://doi.org/10.1002/sec.705.

[23] Bidoki S M, Jalili S, Tajoddin A. PbMMD: A novel policy based multi-process malware detection[J/OL]. Eng. Appl. of AI, 2017, 60 : 57 – 70.
https://doi.org/10.1016/j.engappai.2016.12.008.

[24] Hajmasan G, Mondoc A, Portase R, et al. Evasive Malware Detection Using Groups of Processes[C/OL] // di Vimercati S D C, Martinelli F. IFIP Advances in Information and Communication Technology, Vol 502 : ICT Systems Security and Privacy Protection - 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings. [S.l.] : Springer, 2017 : 32 – 45.
https://doi.org/10.1007/978-3-319-58469-0_3.

[25] Lee E A. The Problem with Threads[J/OL]. IEEE Computer, 2006, 39(5) : 33 – 42.

https://doi.org/10.1109/MC.2006.180.

[26] Microsoft. Processes and Threads(Windows)[EB/OL]. 2017[May]. https://msdn.microsoft.com/en-us/library/windows/desktop/ms684841(v=vs.85).aspx.

[27] Zhang X, Yang Z, Zheng Q, et al. Debugging Multithreaded Programs as if They Were Sequential[C/OL] // International Conference on Software Analysis, Testing and Evolution, SATE 2016, Kunming, China, November 3-4, 2016. [S.l.] : IEEE, 2016 : 78 – 83. https://doi.org/10.1109/SATE.2016.20.

[28] Zhang X, Yang Z, Zheng Q, et al. Automated Testing of Definition-Use Data Flow for Multithreaded Programs[C/OL] // 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017. [S.l.] : IEEE Computer Society, 2017 : 172 – 183. https://doi.org/10.1109/ICST.2017.23.

[29] Lo R W, Levitt K N, Olsson R A. MCF: a malicious code filter[J/OL]. Computers & Security, 1995, 14(6) : 541 – 566. https://doi.org/10.1016/0167-4048(95)00012-W.

[30] Egele M, Scholte T, Kirda E, et al. A survey on automated dynamic malware-analysis techniques and tools[J/OL]. ACM Comput. Surv., 2012, 44(2) : 6:1 – 6:42. http://doi.acm.org/10.1145/2089125.2089126.

[31] Christodorescu M, Jha S, Seshia S A, et al. Semantics-Aware Malware Detection[C/OL] // 2005 IEEE Symposium on Security and Privacy (S&P 2005), 8-11 May 2005, Oakland, CA, USA. [S.l.] : IEEE Computer Society, 2005 : 32 – 46. https://doi.org/10.1109/SP.2005.20.

[32] Gadhiya S, Bhavsar K. Techniques for malware analysis[J]. International Journal of Advanced Research in Computer Science and Software Engineering, 2013, 3(4).

[33] Royal P, Halpin M, Dagon D, et al. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware[C/OL] // 22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA. [S.l.] : IEEE Computer Society, 2006 : 289 – 300. https://doi.org/10.1109/ACSAC.2006.38.

[34] Kang M G, Poosankam P, Yin H. Renovo: A hidden code extractor for packed executables[C] // Proceedings of the 2007 ACM workshop on Recurring malcode. 2007 : 46 – 53.

[35] Willems C, Holz T, Freiling F C. Toward Automated Dynamic Malware Analysis Using CWSandbox[J/OL]. IEEE Security & Privacy, 2007, 5(2) : 32 – 39. https://doi.org/10.1109/MSP.2007.45.

[36] Raffetseder T, Krügel C, Kirda E. Detecting System Emulators[C/OL] // Garay J A, Lenstra A K, Mambo M, et al. Lecture Notes in Computer Science, Vol 4779 : Information Security, 10th International Conference, ISC 2007, Valparaíso, Chile, October 9-12, 2007, Proceedings. [S.l.] : Springer, 2007 : 1 – 18. https://doi.org/10.1007/978-3-540-75496-1_1.

[37] Dinaburg A, Royal P, Sharif M I, et al. Ether: malware analysis via hardware virtualization extensions[C/OL] // Ning P, Syverson P F, Jha S. Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008. [S.l.] : ACM, 2008 : 51 – 62. http://doi.acm.org/10.1145/1455770.1455779.

[38] Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[C/OL] // Draves R, van Renesse R. 8th USENIX Symposium

on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. [S.l.] : USENIX Association, 2008 : 209 – 224.
http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

[39] Peng F, Deng Z, Zhang X, et al. X-Force: Force-Executing Binary Programs for Security Applications[C/OL] // Fu K, Jung J. Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.. [S.l.] : USENIX Association, 2014 : 829 – 844.
https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/peng.

[40] Moser A, Krügel C, Kirda E. Exploring Multiple Execution Paths for Malware Analysis[C/OL] // 2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA. [S.l.] : IEEE Computer Society, 2007 : 231 – 245.
https://doi.org/10.1109/SP.2007.17.

[41] Corina J, Machiry A, Salls C, et al. DIFUZE: Interface Aware Fuzzing for Kernel Drivers[C/OL] // Thuraisingham B M, Evans D, Malkin T, et al. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. [S.l.] : ACM, 2017 : 2123 – 2138.
http://doi.acm.org/10.1145/3133956.3134069.

[42] Han H, Cha S K. IMF: Inferred Model-based Fuzzer[C/OL] // Thuraisingham B M, Evans D, Malkin T, et al. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. [S.l.] : ACM, 2017 : 2345 – 2358.
http://doi.acm.org/10.1145/3133956.3134103.

[43] Rebert A, Cha S K, Avgerinos T, et al. Optimizing Seed Selection for Fuzzing[C/OL] // Fu K, Jung J. Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.. [S.l.] : USENIX Association, 2014 : 861 – 875.
https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert.

[44] Szor P. The art of computer virus research and defense[M]. [S.l.] : Pearson Education, 2005.

[45] Rudd E M, Rozsa A, Günther M, et al. A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions[J/OL]. IEEE Communications Surveys and Tutorials, 2017, 19(2) : 1145 – 1172.
https://doi.org/10.1109/COMST.2016.2636078.

[46] anonymous. Logic bomb[EB/OL]. 2017[March].
https://en.m.wikipedia.org/w/index.php?title=Logic_bomb&welcome=yes.

[47] Ramos D A, Engler D R. Under-Constrained Symbolic Execution: Correctness Checking for Real Code[C/OL] // Jung J, Holz T. 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.. [S.l.] : USENIX Association, 2015 : 49 – 64.
https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos.

[48] 胡细宝, 孙洪祥, 王丽霞. 概率论· 数理统计· 随机过程 [M]. [S.l.] : 北京: 北京邮电大学出版社, 2004.

[49] Moose B. BullMoose[EB/OL]. 2017[March].
http://vxheaven.org/src.php?info=bullmoose.zip.

[50] Woo S C, Ohara M, Torrie E, et al. The SPLASH-2 Programs: Characterization and Methodological Considerations[C/OL] // Patterson D A. Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995. [S.l.] : ACM, 1995 : 24 – 36.
http://doi.acm.org/10.1145/223982.223990.

[51] Microsoft. Multiverso[EB/OL]. 2017[March].

https://github.com/Microsoft/Multiverso.
[52] rshipp. Awesome Malware Analysis[EB/OL]. 2017[March]. https://github.com/rshipp/awesome-malware-analysis.
[53] Virustotal. VirusTotal[EB/OL]. 2017[June]. https://www.virustotal.com/.
[54] Team M. Malwr[EB/OL]. 2017[March]. https://malwr.com/.
[55] Analysis H. Hybrid Analysis[EB/OL]. 2017[October]. https://www.hybrid-analysis.com/.
[56] Amnpardaz. Amnpardaz Sandbox[EB/OL]. 2017[October]. http://jevereg.amnpardaz.com/.
[57] Group C. Valkyrie[EB/OL]. 2017[October]. https://valkyrie.comodo.com.
[58] Analysis H. reverse.it[EB/OL]. 2017[October]. https://www.reverse.it/.
[59] Threatexpert. ThreatExpert[EB/OL]. 2017[October]. http://www.threatexpert.com/.
[60] anlyz. anlyz[EB/OL]. 2017[October]. https://sandbox.anlyz.io/.
[61] Secondwrite. SecondWrite Sandbox[EB/OL]. 2017[October]. https://webportal.secondwrite.com/dashboard/.
[62] Ferrand O. How to detect the Cuckoo Sandbox and to Strengthen it?[J/OL]. J. Computer Virology and Hacking Techniques, 2015, 11(1): 51–58. https://doi.org/10.1007/s11416-014-0224-9.
[63] Sandbox C. Automated malware analysis[J]. URL: https://cuckoosandbox. org, 2013.
[64] Kirat D, Vigna G, Kruegel C. BareCloud: Bare-metal Analysis-based Evasive Malware Detection[C/OL] // Fu K, Jung J. Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.. [S.l.]: USENIX Association, 2014: 287–301. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat.
[65] Kharraz A, Arshad S, Mulliner C, et al. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware[C/OL] // Holz T, Savage S. 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.. [S.l.]: USENIX Association, 2016: 757–772. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kharaz.
[66] Red5. Red5[EB/OL]. 2017[October]. https://github.com/Red5.
[67] jseidl. Goldeneye[EB/OL]. 2017[October]. https://github.com/jseidl/GoldenEye/.
[68] Mcgraw G, Morrisett J G. Attacking Malicious Code: A Report to the Infosec Research Council[J/OL]. IEEE Software, 2000, 17(5): 33–41. https://doi.org/10.1109/52.877857.
[69] Cui H, Wu J, Tsai C, et al. Stable Deterministic Multithreading through Schedule Memoization[C/OL] // Arpaci-dusseau R H, Chen B. 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings. [S.l.]: USENIX Association, 2010: 207–221. http://www.usenix.org/events/osdi10/tech/full_papers/Cui.pdf.

[70] Thuraisingham B M, Evans D, Malkin T, et al. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017[C/OL]. [S.l.]: ACM, 2017.
http://doi.acm.org/10.1145/3133956.

[71] Fu K, Jung J. Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014[C/OL]. [S.l.]: USENIX Association, 2014.
https://www.usenix.org/conference/usenixsecurity14.

攻读学位期间取得的研究成果

[1] 刘烃, **郝宇**, 尹文浩, 张晓东, 刘沛, 樊浩涵, 郑庆华. 一种基于符号计算的动态并行程序污点分析方法. 中国专利 (CN105955877)

[2] 刘烃, **郝宇**, 尹文浩, 张晓东, 刘沛, 樊浩涵, 郑庆华. 一种基于符号计算的动态并行程序污点分析方法. 国际 PCT 专利 (PCT/CN2016/102362)

[3] Zhang X, Yang Z, Zheng Q, **Hao Y**, Liu P, Yu L, Liu T. Debugging Multithreaded Programs as if They Were Sequential. IEEE Access.

[4] Zhang X, Yang Z, Zheng Q, **Hao Y**, Liu P, Yu L, Fan M, Liu T. Debugging Multithreaded Programs as if They Were Sequential. International Conference on Software Analysis, Testing and Evolution, SATE 2016, IEEE.

[5] Zhang X, Yang Z, Zheng Q, Liu P, Chang J, **Hao Y**, Liu T. Automated Testing of Definition-Use Data Flow for Multithreaded Programs. 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, IEEE Computer Society.

[6] **Hao Y**, Zhang X, Yang Z, Liu T. Poster: Dynamic Taint Analysis of Concurrent Program Based on Symbolic Execution. IEEE Security & Privacy, 2017.

学位论文独创性声明（1）

本人声明：所呈交的学位论文系在导师指导下本人独立完成的研究成果。文中依法引用他人的成果，均已做出明确标注或得到许可。论文内容未包含法律意义上已属于他人的任何形式的研究成果，也不包含本人已用于其他学位申请的论文或成果。

本人如违反上述声明，愿意承担以下责任和后果：

1. 交回学校授予的学位证书；

2. 学校可在相关媒体上对作者本人的行为进行通报；

3. 本人按照学校规定的方式，对因不当取得学位给学校造成的名誉损害，进行公开道歉。

4. 本人负责因论文成果不实产生的法律纠纷。

论文作者（签名）：　　日期：　　年　　月　　日

学位论文独创性声明（2）

本人声明：研究生 _____ 所提交的本篇学位论文已经本人审阅，确系在本人指导下由该生独立完成的研究成果。

本人如违反上述声明，愿意承担以下责任和后果：

1. 学校可在相关媒体上对本人的失察行为进行通报；

2. 本人按照学校规定的方式，对因失察给学校造成的名誉损害，进行公开道歉。

3. 本人接受学校按照有关规定做出的任何处理。

指导教师（签名）：　　日期：　　年　　月　　日

学位论文知识产权权属声明

我们声明，我们提交的学位论文及相关的职务作品，知识产权归属学校。学校享有以任何方式发表、复制、公开阅览、借阅以及申请专利等权利。学位论文作者离校后，或学位论文导师因故离校后，发表或使用学位论文或与该论文直接相关的学术论文或成果时，署名单位仍然为西安交通大学。

论文作者（签名）：　　日期：　　年　　月　　日

指导教师（签名）：　　日期：　　年　　月　　日