基于异常分布导向的智能 Fuzzing 方法

欧阳永基* 魏 强 王清贤 尹中旭
(解放军信息工程大学 郑州 450002)
(数学工程与先进计算国家重点实验室 郑州 450002)

摘 要：现有主流智能 Fuzzing 测试一般通过对程序内部结构的精确分析构造新测试样本，因而严重依赖于当前计算机的性能，往往忽略了已发现的程序异常信息对新测试样本构造的指导意义。为了克服上述缺陷，该文提出一种基于异常分布导向的智能 Fuzzing 方法。该方法针对二进制程序测试，建立了 TGM(Testcase Generation Model) 样本构造模型：首先根据计算能力收集测试样本集的相关信息；然后随机选择初始测试样本进行测试；最后，基于测试结果初始化模型参数，根据模型优先选择更有效的输入属性构造新样本并进行新一轮测试，通过重复进行该步骤，在迭代测试中不断更新模型参数，用于指导下一轮新测试样本构造。实验数据表明该方法可以辅助 Fuzzing 选择更有效的样本优先进行测试，设计的原型工具 CombFuzz 在异常检测能力和代码覆盖能力上都有良好表现，同时，在对大型应用程序进行测试时，与微软 SDL 实验室的 MiniFuzz 测试器相比，在限定时间内平均异常发现率提高近 18 倍，并在 WPS 2013 等软件中发现了 7 个 MiniFuzz 无法发现的未公开“可利用”脆弱点。

关键词：软件测试；智能 Fuzzing；异常分布；脆弱点
中图分类号： TP311.1 **文献标识码**： A **文章编号**：1009-5896(2015)01-0143-07
DOI: 10.11999/JEIT140262

Intelligent Fuzzing Based on Exception Distribution Steering

Ouyang Yong-ji Wei Qiang Wang Qing-xian Yin Zhong-xu
(*The PLA Information Engineering University*, *Zhengzhou* 450002, *China*)
(*State Key Laboratory of Mathematical Engineering and Advanced Computing*, *Zhengzhou* 450002, *China*)

Abstract: The current mainstream intelligent Fuzzing often constructs new test samples through precise analysis of the program's internal structure, which is heavily dependent on the performance of the computer and often overlooks the guiding significance of the discovered program information of exceptions for construction of new testing samples. To overcome these shortcomings, this paper presents a method based on intelligent Fuzzing exception distribution steering, which establishes a data-constructing model named TGM (Testcase Generation Model) for binary program testing. Firstly the relevant information of testing samples is collected according to the computing capability. Then random initial testing samples are selected for testing. Finally, the testing results are used to initialize parameters of the model, which guides the priority selection of more effective input attributes to construct new samples for the next round of testing. This procedure is repeated in iterative testing to constantly update model parameters for guiding the next testing. Experimental data shows that this method can assist Fuzzing to prioritize more effective samples for testing. Design prototyping tool CombFuzz has good performance in the exception detection capability and code coverage capability, meanwhile, when the tests are carried out on large programs, compared with MiniFuzz of Microsoft's SDL lab, this method increases the average of exception detection rate by nearly 18 times in a limited period of time, and has found 7 undisclosed "exploitable" vulnerabilities in WPS 2013 and other softwares that MiniFuzz did not find.

Key words: Software test; Smart fuzzing; Exception distribution; Vulnerability

1 引言

Fuzzing[1]测试是软件测试中一种很重要的测试方法，它使用大量随机变异的用例作为输入，监视所检测程序是否在运行这些用例时发生异常。以 zzuf[2], MiniFuzz[3]等为代表的传统 Fuzzing 测试方法广泛应用于代码的检测中，它们可以挖掘出其他检测方法无法检测的脆弱点。但是传统的 Fuzzing

2014-03-04 收到，2014-08-27 改回
国家 863 计划项目(2012AA012902)资助课题
*通信作者：欧阳永基 oyyj07@gmail.com

测试方法的缺陷也很明显：由于测试数据的构造方法过于随机简单，以及测试具有盲目性，导致测试速度快但效率低；难以确定覆盖率，导致无法对 Fuzzing 结果进行评估；由于不能保证充分的代码覆盖导致漏报率高；由于测试数据的相互独立，导致难以发现复杂的脆弱点等。以 EFS[4]和 peach[5]等为代表的智能 Fuzzing 测试工具，根据对测试程序的分析、理解，利用启发式算法、结构化的数据构造描述，能够构造丰富多样的测试数据，但仍然忽视了已有异常对潜在异常的指示作用，往往不能在宏观层面上研究测试数据的构造。而以 Ruijters[6]等为代表的研究人员，探索了利用马尔科夫链建模等技术进行更有效测试的可行性，虽然他们的模型可以生成大量有差异的样本，并能有效评估覆盖率，但是他们依然没有建立程序异常和测试因素的整体关系模型，局限于改善某个、或某几个测试因素，对 Fuzzing 测试的贡献有限。

为了克服现有 Fuzzing 技术对目标程序缺乏理解、测试完全随机且盲目的缺点，研究者提出了许多新技术和新方法[7,8]，基于动态符号执行的智能测试技术逐渐成为其中的研究热点。其具体做法是首先通过静态分析获得危险点，然后通过符号执行收集路径条件，最后利用约束求解工具求解能够执行到指定路径的测试数据进行测试。微软公司软件可靠性部门 Godefroid 等人[9]设计的 SAGE，加州大学伯克利分校的 Molnar 等人[10]在 Valgrind 基础之上设计开发的 CatchConv，斯坦福大学 Engler 等人开发的 ARCHER 以及威斯康星大学 Balakrishnan 等人[11]开发的 CodeSurfer/x86，北京大学 Wang 等人[12]所开发 TaintScope，阿姆斯特丹自由大学[13]开发的 Dowser，上述软件测试模型采用中间表示、污点传播分析、符号执行、执行路径遍历、路径约束求解等思想，不同程度上改进了以 peach 为代表的 Fuzzing 技术的弊端，反映了软件脆弱性测试由模糊化测试向精确化测试转变的趋势。然而不同于 Fuzzing 技术，基于动态符号执行测试中的关键技术，如符号执行、约束求解等技术都需要大量计算资源，并且技术实现较为复杂，不利于大规模推广应用。迄今为止，安全社区最具代表性的开源原型工具 Fuzzgrind, Pingrind, avalanche 和 KLEE，尚无法对大型应用程序进行有效测试，例如 WPS office 2013 软件、暴风影音 5 媒体播发软件、PDF 阅读软件 SumatraPDF 等。实际应用中，安全研究人员发现脆弱点普遍存在出错代码区域重合、触发原理相似的现象。例如研究人员 Exodusintel 利用对 CVE-2013-3147 脆弱点的分析，采用与之相似的对编辑聚焦事件处理的方法构造了能触发新脆弱点的样本。同时，Arcuri 等人通过分析、研究现有随机测试的因素，提供了一种新颖的评估随机测试有效性的理论，它为建立程序异常和测试因素的关系模型提供了理论基础[14]。

本文的主要贡献如下：(1)综合 Fuzzing 技术和动态符号执行技术的优缺点，并考虑已发现脆弱点和潜在脆弱点的相关性，提出了一种基于异常分布导向的样本构造模型 TGM(Testcase Generation Model)。TGM 模型能够优先选择异常概率高与基本块覆盖多的样本进行数据变异，并可以根据测试结果迭代更新模型参数，使模型不断趋近实际测试样本异常分布。(2)在 TGM 模型基础上，设计并构建了智能 Fuzzing 框架，并对其中测试输入参数对应属性的处理等关键问题进行了解决，为合理利用 TGM 模型进行程序测试提供了基础。(3)实现了基于 TGM 的模糊测试器 CombFuzz，利用该系统分别对 BegBunch[15]测试集和 WPS 等大型应用程序进行了测试，并与现有工具进行了对比，实验表明基于 TGM 模型的智能 Fuzzing 测试方法有着较强的异常挖掘能力和代码覆盖能力。

2 基于异常分布导向的样本构造模型

为了优先选择异常触发概率高的样本进行测试，本文对一些开源软件的 Bug 库进行了统计分析，从中发现已测试出的安全缺陷存在一定的隐含规律。例如，在 Mozilla 的 Bugzilla 中，我们发现，有近 200 个安全问题发生在 assertion 语句周围。因此，考虑到安全缺陷发生的规律性，本文提出一种基于异常分布导向的样本构造模型 TGM(Testcase Generation Model)。

为了更好地对模型进行叙述，我们以文件测试为例进行说明，下面给出模型适用于文件测试的一些重要概念的定义：(1)文件偏移：测试文件的基本组成单元，记为 r；(2)文件类型：即测试文件的结构，记为 cat；(3)文件：由 m 个偏移组成，表示为 $s=\{r_1,r_2,\cdots,r_m \mid m>0且m\in N\}$；(4)测试集：由 n 个测试文件组成，表示为 $S=\{s_1,s_2,\cdots,s_n \mid n>0且n\in N\}$。

在单个 Fuzzing 测试中，如果将产生异常标记为事件 A，没有产生异常记为 $\overline{A}$，则发生异常的概率可以记为 $p(A)$。如果以该次 Fuzzing 测试的特征(如测试文件偏移 $r_1,r_2,\cdots,r_m$ 和测试文件类型 cat)为先验，则测试发生异常的概率可以表示为 $p(A \mid r_1,r_2,\cdots,r_m,\text{cat})$。显然单个 Fuzzing 测试为伯努利实验，如果对所有 r_i，cat 的组合都进行无穷次测试，那么就可以精确地获得在给定文件偏移、文件类型

的前提下，事件 A 发生的概率。然而出于测试成本的考虑，重复进行大量实验是不可取的。因而可利用泊松分布在一定的置信区间计算出异常触发的概率，进而选择概率高的样本进行测试。

基于该测试思想，本文建立了 TGM 样本构造模型，辅助 Fuzzing 方法进行程序测试，其概率图表示如图 1 所示，其中，t 表示脆弱性模式，I 表示测试输入参数对应属性，θ 表示 I 可能触发的异常概率，α 表示 θ 的概率分布。

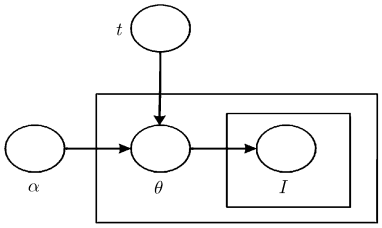

图 1　TGM 概率图

在 TGM 模型中，将脆弱性模式 t 和异常触发的相对强弱 α 作为样本构造的先验条件，由参数 (α,t) 可以获得测试输入参数 I 可能触发的异常概率 θ。采用 TGM 模型构造测试样本，用户首先对测试样本集进行去重等预处理，根据要测试的脆弱性模式，选择 I 中的测试样本，如对于缓冲区溢出检测，用户可选择样本中能污染 strcpy 函数中长度参数的偏移，并以递增和递减该偏移的方式选取待测偏移集合，然后在测试中，根据这些偏移触发的异常数量，迭代调整参数 α ,θ 值，并优先选择异常触发概率高的样本进行测试。

对于 Fuzzing 测试，最重要的是基于 TGM 模型选择参数 I，然后通过改变属性中的元素来构造新的样本。I 的取值可以为多种属性，例如，对于本文研究的文档类 Fuzzing　测试，I 的取值为文档类型 cat 和偏移 r 两种属性。具体而言，给定一组测试样本，首先根据选定的属性 I 进行分组，如根据样本覆盖的基本块数目、指令条数、执行时间、样本文件类型、覆盖函数的偏移等进行分类；然后从组中挑选样本进行测试，并计算可能的异常概率分布；最后，建立合适的模型分析该分布的分布来指导测试器选择下一个待测试的样本。若一个测试集中包含 k 个属性类型，总的异常数量 Q，每种类型的异常数量为 q，则每种类型的异常概率为 q_k / Q。我们将 I 可能触发的异常概率表示为 θ，显然 θ 服从多项分布(multinomial distribution)。对于不同的脆弱性模式，异常触发的概率分布一般并不相同，因此，我们以 α 表示在不同脆弱性模式下异常触发的相对强弱。由于狄利克雷分布是多项分布的共轭先验分布，我们假定 α 服从狄利克雷分布。综上所述，TGM 模型将按照以下步骤来构造新的样本：(1) $\theta \sim \mathrm{Dir}(\alpha)$；(2) $I \sim \mathrm{Multinomial}(\theta)$；(3)根据 I 构造新样本。

若总共有 M 个属性，即 $I=\{r, \mathrm{cat}, \cdots\}$，则 TGM 的概率模型可表示为

$$p(\theta,I|t,\alpha)=p(\theta|t,\alpha)\prod_{m=1}^{M}p(i_m|\theta) \tag{1}$$

如果对于每一个脆弱性模式都训练一个 TGM 模型，则可以去掉参数 t 来简化模型，即式(1)可转化为式(2)：

$$p(\theta,I\mid\alpha)=p(\theta\mid\alpha)\prod_{m=1}^{M}p(i_m\mid\theta) \tag{2}$$

其中 θ 服从分布 $\mathrm{Dir}(\alpha)$，即

$$\mathrm{Dir}(\theta\mid\alpha)=\frac{\Gamma(\alpha_0)}{\Gamma(\alpha_1)\cdots\Gamma(\alpha_k)}\prod_{k=1}^{k}\theta_k^{\alpha_k-1} \tag{3}$$

其中 $\alpha_0=\sum_{k=1}^{k}\alpha_k$。

对于某个 $\mathrm{Dir}(\alpha)$ 分布，则对应输入参数 I 以 p_i 触发异常的概率为

$$p(p(I)=p_i\mid\alpha)=p(p(I)=p_i)p(\alpha) \tag{4}$$

为提高模糊测试器发现测试程序异常的效率，我们首先根据初始测试样本的测试结果，初始化 TGM 模型的参数 α 和 θ，然后根据已初始化的模型，选择属性 I 进行新样本构造，并利用新的样本测试结果，对概率模型参数进行更新，使得模型更准确地反映出偏移、类型对异常触发的影响程度，进而指导测试器选取更有效的测试样本。迭代进行属性选择、新样本构造与测试、模型更新的过程，直到达到预定测试效果或测试时间结束为止。

3　基于 TGM 的模糊测试器设计

基于 TGM 模型，本文设计了针对文件处理软件的测试器，其总体框架如图 2 所示。

该测试器主要由数据分析、变异策略、样本构造、Fuzzing 4 个主要部分组成。其中，数据分析主要负责对样本集合进行去重、优化、以及样本属性

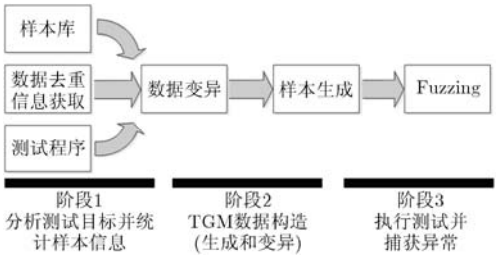

图 2 模糊测试器总体框架

的信息收集；变异策略根据 TGM 模型选择样本属性作为候选变异对象；样本构造通过畸形的数据替换构造新的样本；Fuzzing 主要包括执行测试和 crash 捕获两个主要功能。下面将详细描述基于 TGM 模型的测试器所涉及的关键算法与技术。

3.1 样本集去重

在 Fuzzing 测试中，如果重叠样本过多，将会严重影响测试效率和样本概率的统计。因此，本文的首要目标就是要确保在达到同样测试效果的前提下减少样本集的规模，提高测试用例的构造效率。为了完成上述目的，本文设计了一种基于贪心算法的样本去重算法，该算法每次都在剩下的样本中选取最大的样本进行去重，并记录基本块的地址，防止重复比较，与开源社区 peach 的 MinSet 相比，改善了它简单遍历基本块的模式，减少了优化时间。经过上述处理，在达到同样测试效果的前提下，样本集中的样本间将不会存在执行路径完全相同的两个样本，缓解了相同效果样本的干扰，为后面样本的异常概率统计提高了准确度。

3.2 样本变异属性选取

在基于变异的 Fuzzing 测试中，测试样本类型和变异偏移是 Fuzzing 的两个重要测试输入参数，也是影响测试效率的主要因素。合适的参数选取将会大大提高测试的效率，本节将叙述如何尽可能选取触发异常概率大的测试样本。

3.2.1 样本类型选取算法 由于本文选取样本类型和偏移两种属性来指导数据的变异，为了区分二者的顺序，本文将样本类型作为第一主要因素来指导测试文件的选取。根据已训练好的 TGM 模型，可以获得异常触发的概率分布情况，利用概率分布能快速地计算出文件类型的异常触发概率，因而，模糊测试器依据此信息可选择更合适的样本进行测试，样本类型选取具体算法如表 1 所示。

模糊测试器根据上述算法选取合适的样本类型进行优先测试，其下一步的任务是如何在选取的样本上进行变异，使得测试效果更好。

3.2.2 样本偏移选取 对于测试的目标，无论是文件处理程序、协议或者其它的程序，都有着千差万别的数据结构类型。有些可能是一个“头部+数据”的简单格式，而大多数的格式却更加复杂。本节要解决的主要问题是如何恰当选取样本偏移，既不破坏格式语义，又具备较佳测试效果。为了解决该问题，本文提出了基于快慢表的污点分析技术，首先分析程序中函数的输入偏移，然后根据得到的偏移对输入样本进行变异，采用与文件类型选取相似的算法构造测试数据。

表 1 样本类型选取算法

算法 1 样本类型选取算法

输入：初始样本集合 S，第 i 个样本为 s_i；初始样本的基本块集合 $\mathrm{HBB}=\{\mathrm{HBB}_1,\cdots,\mathrm{HBB}_n \mid n>0且n\in N\}$，$\mathrm{HBB}_i$ 表示第 i 个样本的基本块数，其中 p_i 表示每个样本当前的异常概率；α,θ 和 t 的初始值。

输出：测试程序的异常

(1) get $p_i=\mathrm{TGM}(\alpha,\theta,t),\forall i,1\le i\le n$
(2) if (number of MAX(p_i) >1)
(3) if(number of MAX(HBB_i) >1)
(4) random select the s_i of inputs(equal(p_i), equal(HBB_i))
(5) fuzzing(s_i)
(6) p_i = recalculate(TGM.p_i) and update α and θ
(7) if(isExistException) add exception to set of exceptions
(8) else fuzzing the s_i of MAX(HBB_i)
(9) p_i = recalculate(TGM.p_i) and update α and θ
(10) if(isExistException) add exception to set of exceptions
(11) else fuzzing the s_i of MAX(p_i)
(12) p_i = recalculate(TGM.p_i) and update α and θ
(13) if(isExistException) add exception to set of exceptions
(14)checking fuzzer condition for goto (1) or exit and return exceptions

(1)基于快慢表的污点分析：污点分析是一种有效确认变量间是否相关的方法，为了获得样本对于程序的影响情况，本文利用该技术对程序执行进行精确分析。其主要思想是将不受信任的输入源标记为污点(taint)，然后跟踪这类数据在程序中的执行情况，并把与这些数据具有传播关系的数据同样标记为污点，最后进行安全分析等需要的操作。根据该思想，研究人员开发了大量的污点分析工具，例如 DTA++[16]与 libdft[17]等现代污点分析工具，但是依然存在污点分析效率问题，尤其忽视了对污点分析关键模块影子内存[18]的优化。

考虑到程序执行时往往具备局部性原理，即在一段时间内，整个程序的执行仅限于程序中的某一部分。相应地，执行所访问的存储空间也局限于某个内存区域。为了提高效率，我们可利用页式存储结构的思想来设计影子内存。因此，本文在 libdft 的基础上，设计了基于快慢表影子内存的细粒度污点分析方法，该方法以快慢表的结构跟踪每一个输入字节的污点传播过程，记录任意被污染内存的污点标签(输入偏移)，为偏移的选取提供可选依据。本文设计的基于快慢表的影子内存包含两个数据结

构：污点位图(快表)和污点源映射表(慢表)。污点位图是一段静态分配的内存，其每一位对应 4 GB 虚拟内存中的每一字节，标识其是否被污染。因而采用移位操作可以迅速判断某一内存是否被污染。对于线性地址 x 处单字节，判断其是否被污染的公式为

$$\text{IsTainted} = ((x >> 3)\ \%\text{BIT\_MAP\_SIZE}) \ \&\ (1 << (x\ \%8)) \tag{5}$$

式(5)中 BIT_MAP_SIZE 为污点位图大小，IsTainted 标识该内存是否被污染。污点位图理论大小为 4 GB/8=512 MB。但在实际测试中，将污点位图设置为 64 M，这样会导致多个虚拟地址映射到位图同一位的情况。但由于污点数据仅在寄存器和堆栈区域之间传播，而堆栈区域仅占总内存较小的比例，污点数据在位图中由于更新而产生地址碰撞的可能性极小。此外，如果产生地址碰撞，可以查询污点源映射表进行二次确认。污点源映射表是一张哈希表，用于维护污点单元(包括寄存器和内存)地址和污点源之间的映射关系。实际实现时，只有通过位图判断存在某一内存或寄存器存在污点数据处理时，才更新污点映射表，即添加一个地址项和对应的输入偏移的集合。由于程序执行过程中访问的污点内存只占总内存较小比例，因此可以通过污点位图表快速定位某一内存是否被污染，如果没有被污染(绝大多数情况)，则不必查询污点源映射表，加快污点分析速度。

(2)偏移分析：设计完污点分析策略之后，为了确定偏移的范围，需要对测试程序的执行数据流处理的相关信息进行记录，以供后续模块参考和使用。考虑到函数和指令是处理数据的主体，所以分别对指令和函数进行记录，然后分析它们与偏移位置的映射关系来确定偏移。

(a)指令记录：在程序的执行流中，函数间将包含许多指令，为了分析清楚输入源和函数处理的关系，需要对输入源在程序中的加载处和函数间的指令进行记录，然后在此基础上对每条指令的源操作数和目的操作数进行分析，获取输入源的处理偏移。现实中，程序执行的指令规模十分巨大，为了减少记录规模，本文采用只记录跟污点相关的操作指令。对于每一条污点处理指令，污点来源记录方式如下所示：

$$0x5adc4324\text{: cmp edx, dword ptr ds[ecx]: } [0x7, 0x8, 0x9, 0xa]$$

该记录所表示的含义为：$0x5adc4324$ 指令处理的污点来源为偏移 $0x7$ 处的连续 4 个字节，表示为 $\text{taintsrc}(0x5adc4324) = \{0x7, 0x8, 0x9, 0xa\}$。

因此，通过上述的指令记录方式，可以很快地找寻到程序中某个函数处理的输入来源于样本的哪些字节，即这些字节在样本中所处的偏移。

(b)函数记录：函数记录用于记录程序内部某一子函数的污染数据来源，给 Fuzzing 器提供变异偏移的可选方式。由于测试程序中的函数会频繁地调用系统库函数，因此，该类型函数的调用需要特殊处理。本文主要采用函数摘要的方式对库函数的污点传播进行封装，减少频繁在库函数内部的跟踪次数。在提高效率的同时，记录特定库函数的污染来源。

(3)偏移选取算法：通过污点跟踪和偏移的分析，可以得到输入源一系列的文件偏移。测试时，可以根据实际的情况，适当选取偏移，然后让测试器依次变异这些偏移。对于不同的文件，其偏移的适应度各有差异，偏移选取算法正是为了指导测试器选取最优的偏移进行测试。对于给定的样本文件 s_i，其偏移集合为 $R=\{r_1, r_2, \cdots, r_m \mid m > 0 且 m \in N\}$，根据 TGM 模型，统计每个偏移 r 在测试过程中触发异常的概率大小，根据概率的大小选取下一次测试变异的偏移。具体实现类似于样本类型选取算法，此处不再赘述。

4　实验与分析

为验证方法的有效性和正确性，本文构建了实验环境，并开发了基于异常分布导向的智能 Fuzzing 原型工具 CombFuzz，选取真实的测试用例进行分析与验证。

4.1 已有安全异常检测能力测试

为了验证本文方法对已有安全异常的检测能力，我们选用了 Begbunch 提供的测试用例作为测试对象，并与 zzuf, Fuzzgrind 和 KLEE 工具进行了对比，实验数据如表 2 所示。

经实验测试，本文提出的基于异常分布的智能 Fuzzing 方法能正确发现 Begbunch 测试集中的 67 个安全脆弱点，并没有产生误报，同时可以覆盖 176532 个基本块。对比其它 3 个工具，CombFuzz

表 2 多个工具对比测试结果

测试项		测试工具			
		zzuf	Fuzzgrind	KLEE	本文 CombFuzz
正确性	正确报告数	39	65	73	67
	误报数	0	0	6	0
	漏报数	129	103	95	101
代码覆盖	基本块覆盖数量	122841	197948	225422	176532

在脆弱点检测的正确性方面，明显强于 zzuf，基本能达到以 Fuzzgrind 和 KLEE 为代表的精细化测试工具的异常检测水平。同时，在代码覆盖方面，CombFuzz 能比 zzuf 多覆盖 50000 多个基本块，但是却比 Fuzzgrind 和 KLEE 覆盖的基本块少。分析上述原因，zzuf 是传统的 Fuzzing 技术，它忽视了程序执行信息对测试的反馈能力。Fuzzgrind 是一种典型的基于动态符号执行的智能测试技术，但它的符号执行设计较为简单，实际执行过程中，对于复杂的程序，无法有效生成能覆盖预期目标的用例，导致其无法发现更多的异常。KLEE 相较于 Fuzzgrind 有较大的提升，但由于符号执行技术固有的缺陷，它依然无法摆脱需要大量计算资源的现实要求，对于大型应用程序的测试仍较为勉强。总的来说，CombFuzz 原型工具在对已有安全异常的检测能力和代码覆盖能力都有较好表现。

4.2 未知安全异常检测能力测试

该实验选用了暴风影音 5，WPS office 2013 和 SumatraPDF 作为安全测试对象，同时，利用微软 SDL 实验室的 MiniFuzz 和开源工具 Pingrind 分别对上述 3 个软件进行测试，然后对三者的测试结果进行比较，表 3 为三者测试的结果。

表 3 CombFuzz, MiniFuzz 和 Pingrind 测试发现缺陷数

测试对象	MiniFuzz	Pingrind	CombFuzz	测试时间(h)
WPS office 2013	5	0	55	19
暴风影音 5	1	0	18	19
SumatraPDF	0	0	37	19

从表 3 可以看出，在给定相同的时间下，本方法相比 MiniFuzz 的 Fuzzing 技术，异常发现效率平均提高近 18 倍，并且对于 SumatraPDF 的测试，CombFuzz 发掘了 MiniFuzz 无法发现的异常，而 Pingrind 基本无法找出程序缺陷。对于 Pingrind 工具，其测试思想与 Fuzzgrind 相同，因此，对于大型应用程序进行测试时，为了顺利进行，需对约束进行一定的限制，本文将约束设置为 500 和 1000 分别进行了测试，但都没测试到有效缺陷，可见对于大型应用程序，在有限的测试资源下，基于符号执行的 Fuzzing 技术作用有限；对于 MiniFuzz，它是一款用于简化模糊测试部署的模糊测试工具，其中心思想是构造丰富的测试数据对程序进行测试，它关注随机化数据的样式，却没有分析数据和程序本身的相关性，更缺乏对已有测试数据的反馈学习。同时，为了验证本文方法发掘异常的危害程度，我们通过 MSEC 和人工验证，发现 CombFuzz 挖掘出了 7 个未公开的“可利用”脆弱点，高达 16 个“可能可利用”和“可能不可利用”脆弱点。

另一方面，在利用传统的 Fuzzing 方法进行程序，由于样本文件、偏移变异范围选取不够合适，容易发生长时间无法发现异常的情况，无法判断测试终止条件，影响测试效率。为了验证本文方法对该缺陷的改进，我们对异常发现的时间进行了记录，图 3 是利用 CombFuzz 工具对 WPS office 2013、暴风影音 5 和 SumatraPDF 测试产生异常的数量随时间的分布情况。

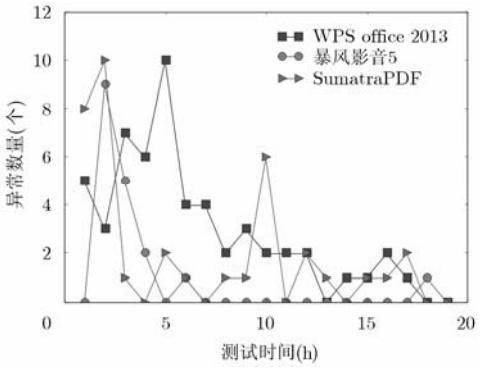

图 3 异常随时间分布情况

从图 3 可以看出，除了对暴风影音 5 的测试结果，其它两个结果都表明 CombFuzz 能较长时间地发现程序的异常，并且，它也能以较短的时间使得异常发现的速度达到最大值。同时，从图 3 可以发现，在后面的测试时间段里，发现异常的数量明显减少，主要可能的因素：一是本文原型工具在测试策略中，采用随机变异的方式存在一定的局限；二是在偏移选取中，不是对每个基本块或函数采取定制的变异范围，而是为方便概率统计和实现，每次变异都采用了同等长度，牺牲了精度；三是没有对程序路径进行分析，无法反馈执行信息，更新偏移范围。

5 结束语

从软件安全测试的角度来看，利用已发现的异常触发概率分布，以尽量覆盖更多的基本块，有针对性地构造测试用例，并优先测试异常概率高的样本，可以显著提高软件测试的效率。本文提出了 TGM 样本构造模型，设计并实现了基于该模型的异常分布导向智能 Fuzzing 测试器，首先通过随机收集测试对象程序的样本集，然后，利用样本去重算法去除冗余测试样本，最终，在基于 TGM 模型的

样本文件类型选取算法和偏移选取算法的支持下，指导测试器优先测试异常概率高的样本。从理论分析和实验数据看，本文给出的基于异常分布导向的智能 Fuzzing 方法，相对微软 SDL 实验室的 MiniFuzz 模糊测试器具有更高的异常发现效率，并在几款具有代表性的常用应用软件中发现了未公开的脆弱点，有效验证了本文方法的正确性。但依然存在一些问题，如对于已经发现的异常，没有对其产生机理进行详细分析，若产生缺陷的代码具有相关性，能对 TGM 模型进行如何改进等，还需要在接下来的工作中进一步研究。

参 考 文 献

[1] Sutton M, Greene A, and Amini P. Fuzzing: Brute Force Vulnerability Discovery[M]. New Jersey: Pearson Education, 2007.

[2] Hocevar S. zzuf — multi-purpose fuzzer[OL]. http://caca.zoy. org/wiki/zzuf, 2013.

[3] Microsoft SDL. MiniFuzz tool[OL]. http://technet.microsoft.com/en-us/edge/minifuzz-overview-and-demo.aspx, 2013.3.

[4] DeMott J, Enbody R, and Punch W F. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing[OL]. https://www.blackhat.com/html/bh-media-archives/bh-archives-2007.html, 2007.

[5] Michael Eddington, Peach[OL]. http://peachfuzzer.com. 2013.10.

[6] Ruijters E. [Master dissertation], Model-checking Markov chains using interval arithmetic[D]. [Master dissertalion], Maastricht University, 2013.

[7] 孙浩, 李会朋, 曾庆凯. 基于信息流的整数漏洞插装和验证[J]. 软件学报, 2013, 24(12): 2767–2781.
Sun Hao, Li Hui-peng, and Zeng Qing-kai. Statically detect and run-time check integer-based vulnerabilities with information flow[J]. *Journal of Software*, 2013, 24(12): 2767–2781.

[8] 崔宝江, 梁晓兵, 王禹, 等. 基于回溯与引导的关键代码区域覆盖的二进制程序测试技术研究[J]. 电子与信息学报, 2012, 34(1): 108–114.
Cui Bao-jiang, Liang Xiao-bing, Wang Yu. The study of binary program test techniques based on backtracking and leading for covering key code area[J]. *Journal of Electronics & Information Technology*, 2012, 34(1): 108–114.

[9] Godefroid P, Levin M Y, and Molnar D. Sage: whitebox fuzzing for security testing[J]. *Queue*, 2012, 10(1): 20.

[10] Molnar D A and Wagner D. Catchconv: symbolic execution and run-time type inference for integer conversion errors[R]. EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2007-23, 2007.

[11] Balakrishnan G, Gruian R, and Reps T. CodeSurfer/x86 — a platform for analyzing x86 executables[C]. Compiler Construction. Springer Berlin Heidelberg, 2005: 250–254.

[12] Wang T, Wei T, Gu G, *et al*.. TaintScope. a checksum-aware directed fuzzing tool for automatic software vulnerability detection[C]. 2010 IEEE Symposium on Security and Privacy (SP), Oakland, USA, 2010: 497–512.

[13] Haller I, Slowinska A, Neugschwandtner M, *et al*.. Dowsing for overflows: a guided fuzzer to find buffer boundary violations[C]. Proceedings of the 22nd USENIX conference on Security, Washington D.C, 2013: 49–64.

[14] Arcuri A, Iqbal M Z, and Briand L. Random testing: theoretical results and practical implications[J]. *IEEE Transactions on Software Engineering*, 2012, 38(2): 258–277.

[15] Cifuentes C, Hoermann C, and Keynes N. BegBunch: benchmarking for C bug detection tools[C]. Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009), Chicago, 2009: 16–20.

[16] Kang M G, McCamant S, and Poosankam P. DTA++: Dynamic taint analysis with targeted control-flow propagation[R]. Proceedings of the 18th Annual Network and Distributed System Security Symposium, San Diego, 2011: 2.

[17] Kemerlis V P, Portokalidis G, and Jee K. libdft: practical dynamic data flow tracking for commodity systems[C]. Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), London, UK, 2012: 121–132.

[18] Min J W, Choi Y H, and Eom J H. Explicit Untainting to Reduce Shadow Memory Usage and Access Frequency in Taint Analysis[M]. Ho Chi Minh City: Springer Berlin Heidelberg, 2013: 175–186.

欧阳永基：男，1985 年生，博士生，研究方向为软件安全与信息安全.

魏　强：男，1979 年生，副教授，研究方向为软件安全与信息安全.

王清贤：男，1960 年生，教授，研究方向为软件安全、信息安全和可信计算.

尹中旭：男，1983 年生，讲师，研究方向为软件安全与信息安全.