自动化白盒模糊测试技术研究

张亚军[1,3] 李舟军[1] 廖湘科[2] 蒋瑞成[3] 李海峰[3]
(北京航空航天大学计算机学院 北京 100191)[1] (国防科学技术大学计算机学院 长沙 410073)[2]
(95934部队 河北 061036)[3]

摘 要 软件的安全性分析和漏洞检测是软件工程和信息安全领域的一个研究热点和难点问题。采用程序分析的方法对软件进行安全性测试,日益受到广泛的关注和重视。首先概述了有关软件安全性测试的基本概念;随后,详细介绍了3种基于程序分析的安全性测试方法:模糊测试、符号执行和自动化白盒模糊测试,并比较了这3种方法的优缺点;最后,给出了自动化白盒模糊测试的分布式模型。

关键词 模糊测试,符号执行,自动化白盒模糊测试,自动化白盒模糊测试分布式模型
中图法分类号 TP301 **文献标识码** A

Survey of Automated Whitebox Fuzz Testing

ZHANG Ya-jun[1,3] LI Zhou-jun[1] LIAO Xiang-ke[2] JIANG Rui-cheng[3] LI Hai-feng[3]
(School of Computer Science and Engineering, Beihang University, Beijing 100191, China)[1]
(School of Computer Science, National University of Defense Technology, Changsha 410073, China)[2]
(95934 Army, Hebei 061036, China)[3]

Abstract Software security analysis and vulnerability testing are one of the researching focus and difficulty in the software engineering. People think highly of the software security testing using program analysis. This paper began with an overview of the concepts of the software security testing, then detailed the popular methods of program analysis in softwared security testing: fuzz testing, symbolic execution and automated whitebox fuzz test and compared them to each other, finally gave an overview of the automated whitebox fuzz testing distributed system.

Keywords Fuzz testing, Symbolic execution, Automated whitebox fuzz test, Automated whitebox fuzz testing distributed system

1 引言

软件作为当今信息社会的重要基础设施,已广泛应用于能源、交通、通信、金融和国防等安全攸关领域中。随着软件的规模越来越大、内部结构越来越复杂,软件中安全漏洞的数量急剧增加。软件中的任何安全漏洞都可能导致非常严重的后果,因此如何自动化地检测软件中的安全漏洞,已成为软件工程和信息安全领域亟需解决的重大课题。

近年来,随着计算机应用到各行各业,由软件漏洞造成的重大损失的新闻屡见不鲜。2009年7月15日,微软向全球用户紧急发布最新安全通告,承认Office组件存在一个严重的安全漏洞,并已遭到黑客的攻击。据估计,此漏洞和以前发现的Office漏洞造成了超过2.8万家中文网站被攻击。据报道,美国每年因为软件漏洞而造成的损失高达600亿美元,如果改进软件测试,则每年可以节省200亿美元[1]。

另一方面,软件漏洞所造成的危害,也从传统的桌面系统传播到手机等其他设备。2012年,世界范围内最知名的手机论坛之一,XDA-Developer曝光,搭载了Exynos 4 SoC构架处理器和使用三星内核源的手机存在重大漏洞,黑客可以通过恶意软件进入到设备的物理内存,更改、盗取用户数据。2013年4月11日,英国媒体报道了一个更为可怕的消息,德国一名安全研究员表示,航空系统存在巨大安全漏洞,仅通过他设计的一个手机应用程序,就可以入侵到客机系统操纵飞机,让飞机改变航线,甚至让其撞向地面。

从上述这几则新闻可知:存在漏洞的软件若进入市场,将造成巨大的安全隐患或无以挽回的重大损失,这些都从一个侧面反映了软件安全性测试的重要性。

2 基于程序分析的软件安全性测试研究

程序分析是指通过对程序进行自动分析,来检验、确认或发现程序的性质(或者规约、约束)。将程序分析的方法应用于软件安全性测试,可有效发现和检测软件中存在的安全缺陷或漏洞。

程序分析的方法大体上可以分为动态分析和静态分析两种。其中,动态分析要通过具体运行待测程序来对软件进行安全性测试;静态分析则不需要运行待测程序,而是从语法或

到稿日期:2013-04-20 返修日期:2013-05-14 本文受国家自然科学基金(61170189,60973105,90718017),教育部博士点基金(20111102130003)资助。

张亚军(1982—),男,硕士生,主要研究方向为网络与信息安全,E-mail:zyj5431875@163.com;李舟军(1963—),男,教授,博士生导师,主要研究方向为网络与信息安全、数据挖掘与文本挖掘;廖湘科(1963—),男,研究员,博士生导师,主要研究方向为系统软件和高性能计算技术。

语义的层面进行分析。

采用程序分析的方法对软件进行安全性测试，可实现测试的自动化，提高测试的效率。下面，主要介绍几个比较常用的程序分析方法。

2.1 模糊测试

模糊测试(Fuzz testing 或 Fuzzing)，其主要思想是向一个系统输入不规则的数据，来测试系统是否会发生崩溃。模糊测试在软件安全性测试中的应用非常广泛[2]。

模糊测试，最初被称为随机测试(Random Testing)[3]，它是一种典型的黑盒动态分析技术。模糊测试是目前最为流行的动态分析方法，存在着很多种采用模糊测试方法的测试工具，如 FileFuzz、SPIKEfile、Peach[4] 等等。

模糊测试是一种有效的软件安全性测试方法，但它也存在自身的局限性。例如，在图 1 所示的程序段中，以 P_i 来代表执行程序段 i 的概率，同时假设 int 类型为 32 位。如果要执行第一个 if 语句后面的程序段，必须满足条件 $a[0]=3$。那么 $P_1+P_2+P_3=2^{-32}$，而 $P_4=1-2^{-32}$。同样，$a[1]=4$、$a[2]=5$ 的概率都为 2^{-32}，那么 $P_1+P_2=2^{-64}$、$P_1=2^{-96}$。由此，可以得出 $P_1=2^{-96}$、$P_2=2^{-64}-2^{-96}$、$P_3=2^{-32}-2^{-64}$、$P_4=1-2^{-32}$。由此可见，$P_4 \gg P_3 \gg P_2 \gg P_1$。如果进行随机测试，则大部分的测试用例都只对程序段 4 进行重复测试，而难以对其它程序段进行测试。

```
1. void function(int a[4])
2. {
3.   if(a[0]== 3){
4.     if(a[1]== 4){
5.       if(a[2]== 5){程序段 1}
6.       else{程序段 2}
7.     }
8.     else{程序段 3}
9.   }
10.  else{程序段 4}
11. }
```

图 1 示例程序 1

2.2 符号执行

符号执行[5](Symbolic Execution)的基本思想是：以符号变量而不是具体的输入值来模拟程序的执行过程。

传统的软件安全性测试方法，大多采用一组测试用例对待测程序进行检测。测试时为待测程序提供具体的数值作为输入，然后得到的输出值也是具体数值。而符号执行则完全不同，它采用符号变量代替具体的数值作为待测程序的输入，得到的输出则是由这些符号变量所组成的表达式。

下面，以图 2 所示的程序段来说明符号执行和传统测试的不同。

```
1. int function(int a,int b,int c)
2. {
3.   int x=a * b;
4.   int y=b * c;
5.   int z=x * y+a;
6.   return z;
7. }
```

图 2 示例程序 2

图 2 所示的程序采用传统的测试方法，可以给 a、b、c 赋值来进行测试，例如，a、b、c 的取值分别为 2、3、4，则表 1 显示了按照图 2 所示程序的测试过程。

表 1 示例程序 2 随机测试过程

步骤	a	b	c	x	y	z
1	2	3	4	~	~	~
2	2	3	4	6	~	~
3	2	3	4	6	12	~
4	2	3	4	6	12	20
5	2	3	4	return z=20		

如果采用符号执行方法，利用符号变量作为程序的输入，假设 a、b、c 的取值分别为 α、β、γ，表 2 显示了图 2 所示程序的执行过程。

表 2 示例程序 2 符号执行过程

步骤	a	b	c	x	y	z
1	α	β	γ	~	~	~
2	α	β	γ	$\alpha\beta$	~	~
3	α	β	γ	$\alpha\beta$	$\beta\gamma$	~
4	α	β	γ	$\alpha\beta$	$\beta\gamma$	$\alpha(\beta^2\gamma+1)$
5	α	β	γ	return $z=\alpha(\beta^2\gamma+1)$		

通过两个表的对比可以发现：传统的测试方法只能利用有限个输入值对程序中的有限多条执行路径进行探测；而符号执行方法则依次解释程序中的每一条指令：对于非分支指令，求出其新的符号表达式；而对于分支指令，则将原路径条件与每个分支满足的分支条件进行合取，作为该分支路径的路径条件。因此，符号执行原则上可以遍历所有的执行路径，更有利于检测程序中的安全漏洞。但符号执行方法存在符号路径条件的约束求解、路径爆炸等诸多问题。

自从符号执行的思想提出以后，已经出现了很多种采用符号执行思想进行程序分析的工具，比如 EFFIGY[5]、SELECT[6]、DISSECT[7,8]、KLEE[9]、EXE[10]、PREfix[11]、IntScope[12] 等。EFFIGY 和 SELECT 是符号执行早期比较有代表性的工具，出现于 20 世纪 70 年代，两者几乎同时出现。EXE 和 KLEE 都是近期比较有名的符号执行工具，其中，KLEE 更是在 EXE 的基础上，取得了非常好的效果。

2.3 白盒模糊测试

白盒模糊测试(Whitebox fuzz testing)[13] 是一种结合了模糊测试和符号执行的方法。

白盒模糊测试的出现，得益于动态测试生成技术的发展[3,8]。采用动态测试生成技术对程序进行分析，不仅要给程序一个具体输入来执行待测程序，同时，还要存储程序的路径约束，采用符号分析的方法进行分析。程序执行过程中，要将具体值和符号值分别存储在 C 和 S 中，C 代表了程序执行过程中的具体值的集合，而 S 代表了符号值的集合。C 和 S 都与待测程序中的变量一一对应。

对一个程序进行白盒模糊测试，首先要从一个初始输入开始，以初始输入运行待测程序，得到一条执行路径。在运行程序的过程中，要记录程序的路径约束。然后，将路径约束的某个约束条件取反，得到一个新的输入。再以新的输入执行待测程序，得到一条新的执行路径。将这个过程重复下去，直到得到待测程序的所有执行路径。

图 3 所示的程序段中，如果 $a[0]$、$a[1]$、$a[2]$ 的值全为 0，则程序出现错误。用白盒模糊测试方法分析该程序段，假设

初始输入为 $a[0]=1$、$a[1]=2$、$a[2]=3$。图 4 显示程序段的所有执行路径。程序第一次执行时，会沿着图 4 中最左边的一条路径执行，此时，程序中 *sum* 的值为 0，约束条件为{$a[0]\neq0$，$a[1]\neq0$，$a[2]\neq0$}。程序第二次执行时，将{$a[0]\neq0$，$a[1]\neq0$，$a[2]\neq0$}中的一个取反，得到一个新的输入。比如，将 $a[2]\neq0$ 取反，成为 $a[2]=0$，则新的输入为 $a[0]=1$、$a[1]=2$、$a[2]=0$，以此输入执行该程序。此过程一直重复下去，直到图 4 中的所有执行路径都执行一遍。

```
1. int function(int a[3])
2. {
3.    int sum=0;
4.    if(a[0]== 0) {sum++;}
5.    if(a[1]== 0) {sum++;}
6.    if(a[2]== 0) {sum++;}
7.    if(sum == 3) {error;}
}
```

图 3　示例程序 3

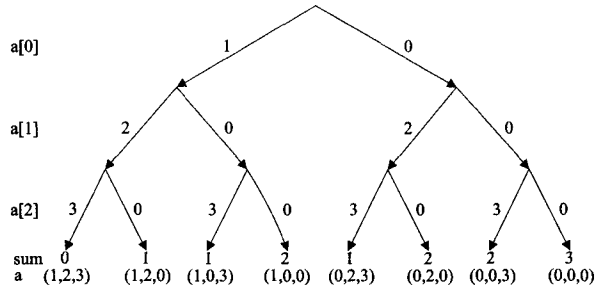

图 4　示例程序 3 执行路径

从理论上来讲，白盒模糊测试是一种比较完美的软件安全性测试方法，但是，还存在一些制约着其走向应用的问题，比如：路径爆炸问题。程序中的执行路径数量会随着程序中分支语句的数量成指数级增长，当程序的规模增长到一定程序时，执行路径的数量会远远超过计算机的处理能力。

除此之外，白盒模糊测试一般都会借助约束求解器来进行分析，比如 Disolver[14]、STP[15]、Z3[16]等。由于分支语句的增加，需要处理的情况也越来越复杂，而约束求解器的能力是有限的，其无法应对不断增加的执行路径。

目前，采用白盒模糊测试的测试工具主要有 DART[17]、CUTE[18]、SMART[19]、SAGE[13]等，但这些工具都不是开源的。目前开源的白盒模糊测试工具有 Catchconv[20] 和 Fuzzgrind[21]，两者均为轻量级的白盒模糊测试工具。

其中，由微软开发的 SAGE 性能非常出色。SAGE 由测试器、追踪器、覆盖率统计器、符号执行器 4 个部分组成。其中，测试器执行待测程序，并检验其是否存在漏洞；追踪器负责再次执行待测程序，记录执行日志，并在机器指令层面进行追踪；覆盖率统计器要统计在执行待测程序期间，测试了哪些程序块；符号执行器负责记录路径约束并生成新的输入。

由于采用了白盒模糊测试技术，SAGE 取得了非常好的测试效果，发现了一些隐藏很深的漏洞：

(1)MS07-017 漏洞。黑盒模糊测试和静态分析工具都没有发现 MS07-017 漏洞，但是 SAGE 在几小时内成功找到了这个漏洞。

(2)压缩文件。SAGE 在压缩文件发现了两个新的漏洞，一个是栈溢出漏洞，另一个是无限循环漏洞。

(3)媒体文件。SAGE 对“Media 1”、“Media 2”、“Media 3”、“Media 4” 4 种应用广泛的媒体文件格式进行了分析，并在所有的 4 种文件格式中都发现了漏洞。

(4)Office 2007。在对 Office 2007 进行分析时，SAGE 生成了 4548 个测试用例，其中 43 个测试用例导致了程序的崩溃。

SAGE 的优异表现，证明了白盒模糊测试的良好性能。

2.4　小结

模糊测试，是最流行的动态分析方法之一，主要通过向待测程序提供大量的随机输入来检测程序的性质。模糊测试的优点在于门槛较低，缺点是测试具有盲目性，可能有大量的测试用例实际上是重复的。

符号执行，是从语义或语法层面上对程序进行分析，甚至不需要执行待测程序。与模糊测试相比，符号执行更能揭示程序的内在缺陷，但是，在处理大型程序时，往往力不从心。

白盒模糊测试，则是一种基于动态分析的符号执行方法，结合了模糊测试和静态符号执行的优点。白盒模糊测试是一种有效的安全测试方法，但也存在符号约束求解、路径爆炸等诸多问题，一般只探测待测程序的部分执行路径。

3　白盒模糊测试分布式系统

白盒模糊测试过程中，约束求解器等工具的求解过程会占用大量的时间。因此，在白盒模糊测试过程中，待测程序的一次执行过程所需时间往往是该程序正常执行的几倍。另一方面，由于路径爆炸的问题，采用白盒模糊测试方法的工具一般仅探测程序的部分路径。

软件正向着并行化、分布式的方向发展。为提高白盒模糊测试的执行效率，我们设计并实现了一个分布式结构的白盒模糊测试系统，它可将测试任务分配到多个计算机上，以便在有限的时间内，探测更多的执行路径，有效提高执行效率和代码测试的覆盖率。

3.1　系统结构

白盒模糊测试分布式系统主要由一个控制单元和多个执行单元组成，其结构如图 5 所示。

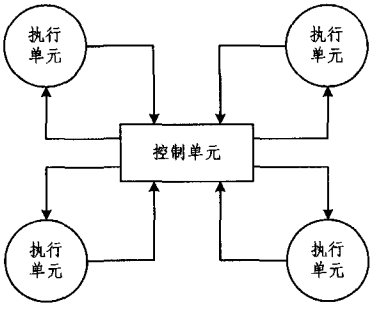

图 5　白盒模糊测试分布式系统结构

控制单元处于系统的核心地位，控制着整个系统的运行和测试任务的分配。系统的启动、停止，以及各个执行单元的工作都需要控制单元进行协调与分配。各个执行单元之间没有任何联系，独自工作。

3.2　系统各单元功能

控制单元负责将未执行过的测试任务分配给各个执行单

元，同时还负责从各个执行单元接收执行结果。当执行完待测程序的所有执行路径以后，控制单元要负责结束整个系统的运行。控制单元必须记录已执行过的测试任务和待执行的测试任务，以确保不重复执行。为便于管理各执行单元，控制单元必须保持两个队列，一个队列记录处于“工作”状态的执行单元，另一个队列记录处于“空闲”状态的执行单元。

执行单元有“工作”和“空闲”两个工作状态。当执行单元处于“空闲”状态时，可以从控制单元接收测试任务，然后对其进行安全性测试，此时该执行单元进入了“工作”状态。当测试任务完成后，将结果传输给控制单元，该执行单元进入“空闲”状态以等待下一步的测试任务。

3.3 系统工作流程

系统启动后，所有的执行单元都处于“空闲”状态，等待控制单元分配测试任务。控制单元在系统启动后，首先确定初始的测试任务(可以从一个给定的测试任务开始，比如对图2所示的程序，就可以从 $a[0]=1$、$a[1]=2$、$a[2]=3$ 开始)。

假设在控制单元中初始测试任务为 i_0，已执行的测试任务队列为 I_e，待执行的测试任务队列为 I_w，记录处于“工作”状态的执行单元的队列为 E_b，记录处于“空闲”状态的执行单元的队列为 E_w。在初始状态下，I_e 和 E_b 的内容都为空，$I_w=\{i_0\}$，E_w 中包含所有的执行单元。

系统运行以后，控制单元进行双线程操作，一个线程负责将测试任务分配给执行单元，另一个线程负责从执行单元接收执行结果。其中，第一个线程按照以下流程工作：

(1)控制单元首先判断 I_w 和 E_w 是否为空。如果 I_w 和 E_w 都不为空，则进入(2)；如果 I_w 或 E_w 中有一个为空，则要分成以下两种情况：①I_w 和 E_b 都为空，则说明所有的执行单元都处于“空闲”状态，并且没有新的测试任务，则整个系统结束；②I_w 和 E_b 都不为空，则等待一小段时间 t，重复(1)。

(2)控制单元从 I_w 中取出一个测试任务 i_k，并将其从 I_w 中删除，加入到 I_e；同时，控制单元从 E_w 中选定一个执行单元 e_k 加以删除，并将其加入到 E_b 中。控制单元和 e_k 建立连接，将 i_k 分配给 E_k 执行后，断开连接，返回到(1)。

与此同时，控制单元的第二个线程按照以下流程工作：

(1)控制单元监听是否有执行单元的通信请求，如果没有通信请求，则重复(1)；如果有通信请求，则进入(2)。

(2)控制单元根据执行单元的通信请求，与其建立连接，并且识别建立连接的执行单元的身份，假设其为 e_l。控制单元接收 e_l 的执行结果，并将 e_l 加入到 E_w 中，同时从 E_b 中删除 e_l。控制单元从 e_l 接收的结果中找到其执行的测试任务及其相应的约束条件，进行记录；同时，控制单元根据该约束条件生成新的测试任务集 I_l，进入(3)。

(3)控制单元判断 I_l 是否为空。如果 I_l 为空，则回到(1)，否则从 I_l 弹出一个测试任务 i，并将 I_e 和 I_w 进行对比，如果 i 所对应的执行路径与 I_e 及 I_w 中的任意一项都不相同，则将 i 加入到 I_w 中，否则将 i 丢弃，重复(3)。

系统运行以后，执行单元的工作流程如下：

(1)等待控制单元建立连接，如果建立连接，则进入(2)，否则继续等待。

(2)与控制单元进行通信，得到测试任务 i_k。执行单元使用 i_k 执行待测程序，并得到一条执行路径，进入(3)。

(3)执行单元将得到的执行路径的各个约束条件分别取反，生成若干个新的测试任务，得到一个新的测试任务集 I_k，进入(4)。

(4)请求与控制单元建立连接，将 i_k 以及得到的执行路径和新的测试任务集 I_k 传输给控制单元中，然后返回到(1)。

处于核心地位的控制单元中，两个线程并行执行，一个负责将测试任务分配给执行单元，当所有的执行单元都处于“空闲”状态并且没有未执行的输入时，则说明已经执行完待测程序的所有执行路径，系统结束运行；另一个线程负责从执行单元接收执行结果，并对新生成的测试任务进行取舍，丢弃那些重复的测试任务，避免对同一条执行路径的重复执行。

结束语 本文首先概述了软件安全性测试的基本概念；然后介绍了3种基于程序分析的软件安全性测试方法：模糊测试、符号分析和白盒模糊测试，并比较了这3种方法的优缺点；最后介绍了我们设计并实现的白盒模糊测试分布式系统的结构和工作流程。

从以上分析可以看出，软件安全测试的这几种方法都有其各自的优缺点，还在不断完善之中，更好的测试方法和测试工具将会不断涌现。

参 考 文 献

[1] The economic impacts of inadequate infrastructure for software testing[R]. National Institute of Standards and Technology, Planning Report 02-3. May 2002

[2] Takanen A, DeMott J, Miller C. Fuzzing for Softwair Security Testing and Quality Assurance[M]. USA: Aatech House Inc., 2008:22-32

[3] Duran J, Ntafos S. An Evaluation of Random Testing [J]. IEEE Transactions on Software Engineering, 1984, SE-10(4):438-444

[4] Peach[OL]. http://peachfuzzer.com/

[5] King J C. Symbolic Execution and Program Testing [J]. Journal of the ACM, 1976, 19(7):385-394

[6] Boyer R S, Elspas B, Levitt K N. SELECT-A formal system for testing and debugging programs by symbolic execution[C]// Proc. 1975 Int. Conf. Reliable Software. IEEE Computer Society, Long Beach, CA, 1975:234-245

[7] Howedn W E. Symbolic Testing and the DISSECT Symbolic Evaluation Systerm[J]. IEEE Transactions on Software Engineering, 1977, 7(4):266-278

[8] Howedn W E. DISSECT-A Symbolic Evaluation and Program Testing System[J]. IEEE Transactions on Software Engineering, 1978, 1(4):70-73

[9] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs [C]//OSDI'08. Dec. 2008

[10] Cadar C, Ganesh V, Pawlowski P, et al. EXE: Automatically generating inputs of death[C]// Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006). 2006

[11] Bush W R, Pincus J D, Sielaff D J. A static analyzer for finding dynamic programming errors[J]. Software-Practice and Experience, 2000, 30(7):775-802

(下转第22页)

的任务并行化，于是 MPI+OpenMP 混合编程的优势就逐渐显露出来，无论从运行时间还是从加速比来看，它都要明显优于 MPI 并行程序。

另外，从图 5 中 MPI 与 MPI+OpenMP 运行加速比曲线可看出，当矩阵规模＞1600＊1600 时 MPI 程序的加速比趋势呈现缓慢下移，而 MPI+OpenMP 混合程序也趋于平缓的走势，这是因为程序在运行时受到了硬件、通信和节点数目等因素的限制所致，但从图 6 所示的 MPI 和 MPI+OpenMP 程序运行的平均墙钟时间上可以看出，MPI+OpenMP 模式程序的运算速度依然要明显高于单一 MPI 模式。

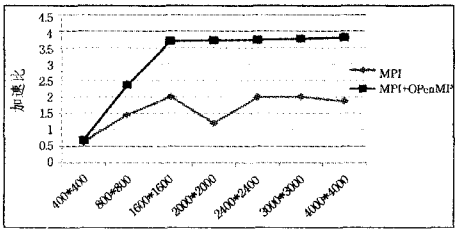

图 5　MPI 与 MPI+OpenMP 运行加速比曲线

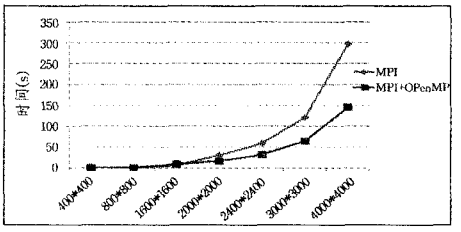

图 6　MPI 与 MPI+OpenMP 程序运行时间曲线

结束语　笔者的体会是，对于多核集群系统而言，目前在并行应用程序设计中，使用 MPI+OpenMP 混合编程模式更接近系统硬件的结构性能，因此它不失为一个可行的途径。

毕竟条件有限，实验所用的矩阵规模和节点数还远远不够，实验结果也难于以点带全。我们只是把实验结果和体会写出来，以飨读者。

参 考 文 献

[1] Hwang K. Advanced Computer Architecture: Parallelism Scalability Programmability[M]. New York: McGraw-Hill Inc., 1993

[2] Group W, Skjellum E L A. Using MPI——Portable Parallel Programming with the Massage Interface(Second Edition)[M]. CambridgeMassachusetts, London, England: The MITPress, 1999

[3] 都志辉. 高性能计算之并行编程技术 MPI 并行程序设计[M]. 北京：清华大学出版社，2001

[4] Robert S A J. Multi-core Programming: Increasing performance Through Software Multi-threading [M]. 李宝峰，富弘毅，李韬，译. 北京：电子工业出版社，2007：145-283

[5] Chandra R, Dagum L, Kohr D, et al. Menon: Parallel programming in OpenMP[M]. Morgan Kaufmann Publisher, Inc., San Francisco, CA, USA, 2001

[6] OpenMP C and C++ Application Program Interface[OL]. version3. May 2008. http://www.openmp.org

[7] Brown R. Performance and Productivity Comparison Between OpenMP and MPI[J]. Int Parallel Prog, 2007, 35: 441-458

[8] 章隆兵，吴少刚，蔡飞. 适合集群 OpenMP 系统的制导扩展[J]. 计算机学报，2004，27(8)：1129-1135

[9] 陈永健. OpenMP 编译与优化技术研究[D]. 北京：清华大学，2004

[10] Core i7 QPI 技术解密[OL]. http://wenku.baidu.com/view/63e77d160b4e767 f5acfcea -e. html. Wang D T. The CELL microprocessor. Real World Technologies, 2005

[11] Smith L, Bull M. Development of mixed mode MPI+OpenMP applications [J]. Scientific Programming, 2001, 9: 83-98

[12] Lusk E, Chan A. Early Experiments with the OpenMP/MPI Hybrid Programming Model [C]// IWOMP'08 Proceedings of the 4th International Conference on Open MP in a New era of Parallelism. Springer, 2008, 5004: 36-47

[13] 叶晓敏. 基于多核处理器并行 EDA 算法研究[D]. 复旦大学

(上接第 10 页)

[12] Wang T, Wei T, Zou W. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution [C]// Network and Distributed System Security Symposium. USA: Internet Society, 2009

[13] Godefroid P, Levin M, Molnar D. Automated whitebox fuzz testing[C]// NDSS. 2008

[14] Hamadi Y. Disolver: A Distributed Constraint Solver[R]. Technical Report MSR-TR-2003-91. Microsoft Research, December 2003

[15] Ganesh V, Dill D. A Decision Procedure for Bit-vectors and Arrays [M]. Computer Aided Verification. Berlin: Springer-verlag, 2007: 524-536

[16] Moura L, Bjorer N. Z3: An Efficient SMT solver. Tools and Algorithms for the Construction and Analysis of Systems[M]. Berlin: Springer-Verlag, 2008: 337-340

[17] Godefroid P, Klarlund N, Sen K. DART: Directed Automated Random Testing[C]// Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation). Chicago, June 2005: 213-223

[18] Sen K, Marinov D, Agha G. CUTE: A Concolic Unit Testing Engine for C[C]// European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering. USA: ACM Press, 2005: 263-272

[19] Godefroid P. Compositional Dynamic Test Generation[C]// Proceedings of POPL'2007 (34th ACM Symposium on Principles of Programming Languages). Nice, January 2007: 47-54

[20] Molnar D, Wagner D. Catchconv: Symbolic Execution and Runtime Type Inference for Integer Conversion Errors[R]. USA: Unversity of California Berkeley, 2007

[21] Fuzzgrind[OL]. http://esec-lab.sogeti.com/pages/Fuzzgrind